Efficient Non-Sequential Access and More Ordering Choices in a Search Tree

Lubomir Stanchev
Computer Science Department
Indiana University - Purdue University Fort Wayne
Fort Wayne, IN, USA
stanchel@ipfw.edu

***Abstract*—A traditional search tree allows for efficient sequential access to the elements of the tree. In addition, a search tree allows for efficient insertion of new elements and efficient deletion of existing elements. In this article we show how to extend the capabilities of a search tree by presenting an algorithm for efficient access to predefined subsets of the indexed elements. This is achieved by marking some of the elements of the search tree with marker bits. In addition, our algorithm allows us to efficiently retrieve the indexed elements in either ascending or descending direction relative to each of the ordering attributes.**

Index Terms*—*marker bits*; *search trees*; *ordering directions*; *data structures

I. Introduction

The paper extends a conference paper on the topic of efficient access to non-sequential elements of a search tree ([5]). The major contribution of this article is the introduction of an algorithm that extends the ordering choices for the elements that are returned. As a minor contribution, we show that retrieving multiple elements from a search tree with marker bits takes time that this proportional to the size of the tree.

A balanced search trees, such as an AVL tree ([1]), an AA tree ([2]), or a B^+ tree ([3]), allows efficient retrieval of elements that are consecutive relative to an in-order traversal of the tree. However, there is no obvious way to efficiently retrieve the elements that belong to a predefined subset of the stored elements if they are not sequential in the search tree. Similarly, there is no obvious way of retrieving the elements in an order that is different from the tree order (or the reverse of the tree order). For example, consider a database that stores information about company employees. A search tree may store information about the employees ordered first by age ascending and then by name ascending. This search tree can be used to retrieve all the employees sorted by age, but the search tree does not efficiently support the request of retrieving all rich employees (e.g., making more than $100,000 per year) sorted by age. In this paper, we will show how the example search tree can be extended with marker bits so that both requests can be efficiently supported. In addition, we will show how the search tree can be used to efficiently retrieve the elements in a different order, for example, ordered by age descending (rather than ascending) and then by name ascending.

The techniques that are proposed in this paper will increase the set of requests that can be efficiently supported by a search tree. This means that fewer search trees will need to be built. This approach will not only save space, but will also improve update performance. For example, [6] shows how our approach can be applied to perform index merging. Specifically, indices on the same elements that have different orderings can be merged when the orderings are on the same attributes. Similarly, indices with orderings on the same attributes that contain common elements can be merged together.

Naïve solutions to the problem of efficiently accessing a non-sequential subset of the elements that are indexed fails. For example, it is not enough to mark all the nodes of the search tree that contain data elements that belong to the subset. This approach will not allow us to prune out subtrees because it can be the case that the parent node does not belong to an interesting subset, but some of the descendent nodes do belong. Similarly, efficiently accessing the elements of a search tree where the ascending and descending direction of the attributes is changed is not trivial because this can require both forward and backward scanning.

To the best of our knowledge, detailed explanation of how marker bits work have not been previously published. Our previous work [6] briefly introduces the concept of marker bits, but it does not explain how marker bits can be maintained after insertion, deletion, or update. Other existing approaches handle requests on different subsets of the search tree elements by exhaustive search or by creating additional search trees. However, the second approach leads to not only unnecessary duplication of data, but also slower updates to multiple copies of the same data. Similarly, to the best of our knowledge, no previous research addresses the problem of efficiently retrieving the elements of a search tree in order that is different from the search order or its reverse.

Given a subset of the search elements S, our approach to efficiently retrieve these elements marks every node in the tree that contains an element of S or that has a descendant that contains an element of S. These additional marker bits will only slightly increase the size of the search tree (with one bit per tree node), but will allow efficient logarithmic execution of requests that ask for the elements of S in the tree order. It will take time proportional to the size of the tree to retrieve all the elements of S.

The algorithm that returns the elements of a search tree in an order that changes the ascending and descending direction of the attributes repeatedly calls the `next` method. The method tries to find the "next" element that has the same value for all but the last attribute as the current node, where next is defined relative to the direction of the last attribute. If such an element does not exist, then the method tries to find the next element that has the same value for all but the last two attributes and so on, where the `next` method is recursive.

In what follows, Section II presents core definitions, Section III describes how to efficiently perform different operations on a search tree with marker bits, and Section IV contains the conclusion and directions for future research.

II. Definitions

Definition 1 (MB-tree): An *MB-tree* has the following syntax: $\langle\langle S_1, \ldots, S_s\rangle, S, O\rangle$, where S and $\{S_i\}_{i=1}^{s}$ are sets over the same domain Δ, $S_i \subseteq S$ for $i \in [1..s]$, and O is an ordering over Δ. This represents a balanced search tree of the elements of S (every node of the tree stores a single element of S), where the in-order traversal of the tree produces the elements according to the order O. In addition, every node of the tree contains s marker bits and the i^{th} marker bit is set exactly when the node or one of its descendants stores an element that belongs to S_i - we will refer to this property as the *marker bit property*.

The above definition can be extended to allow an MB-tree to have multiple data values in a node, as is the case for a B Tree. However, this is area for future research.

Going back to our motivating example, consider the MB-tree $\langle\langle \mathit{RICH\_EMPS}\rangle, \mathit{EMPS}, \langle \mathit{age}\ \texttt{asc}, \mathit{name}\ \texttt{asc}\rangle\rangle$. This represents a search tree of the employees, where the ordering is first relative to the attribute *age* in ascending order. If two employees have the same age, then they are ordered relative to their name in ascending order. The *RICH_EMPS* set consists of the employees that make more than \$100,000 per year. Figure 1 shows an example instance of this MB-tree. Each node of the tree contains the name of the employee followed by their age and salary.

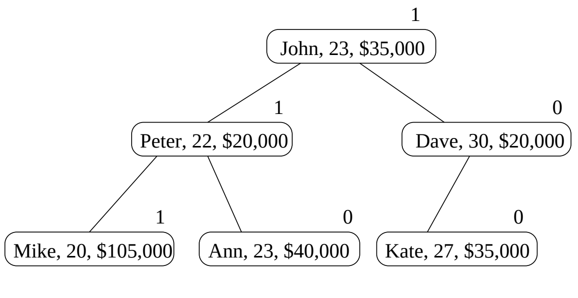

Fig. 1. Example of an MB-tree

Each node in the MB-tree contains the name of the employee, their age, and their salary. Note that Ann and John are the same age. However, Ann comes before John in the search order because "Ann" is lexicographically before "John".

TABLE I
INTERFACE OF A NODE

(*operation*)	(*return value*)
`left()`	left child
`right()`	right child
`parent()`	parent node
`data()`	stored data
`m[i]`	the i marker bit ($1 \leq i \leq s$)

Above each node, the value of the marker bit is denoted, where the bit is set exactly when the node or one of its descendants contains a rich employee. As the figure suggests, the subtree with root node that contains the name Dave can be pruned out when searching for rich employees because the marker bit of the root node is not set. We will show that this MB-tree can be used to efficiently find not only all employees sorted by age, but also all rich employees sorted by age.

The tree can also be used to efficiently find employee (or rich employees) ordered by, for example, age descending and then name ascending. The query that is asking for all employees will return the employees in order: Dave, Kate, Ann, John, Peter, and Mike, while the second query will return only Mike (the only rich employee). Note that throughout this paper *efficient* refers to logarithmic time relative to the size of the search tree.

III. Operations on an MB-tree

Although an MB-tree does not need to be binary, in the following section we will consider only binary trees. The presented algorithms can be extended to non-binary trees and this is area for future research. We will assume that every node of the search tree supports the methods of the interface that is shown in Table I in constant time, where $\{S_i\}_{i=1}^{s}$ are the marker bit sets.

Next, we describe how the algorithms for tree update and search can be extended in the presence of marker bits. Note that supporting more ordering choices only affects the search algorithm.

A. Element Insertion

After an algorithm has inserted a leaf node n, it should call the `insert_fix` method from Algorithm 1 to update the marker bits in the tree.

Algorithm 1 `insert_fix(Node` n`)`

```
1: for $i \leftarrow 1$ to $s$ do
2:     if $n$.data() $\in S_i$ then
3:         $n.m[i] \leftarrow 1$
4:     else
5:         $n.m[i] \leftarrow 0$
6:     end if
7: end for
8: insert_parent_fix($n$.parent(), $n.m$)
```

Lines 1-7 of the code set the marker bits for the new node. The call to the recursive procedure `insert_parent_fix` fixes the marker bits of the ancestors of the inserted node, where the later procedure is presented in Algorithm 2.

Algorithm 2 `insert_parent_fix(Node` n`, Bit[]` m`)`

```
1: if $n = \texttt{null}$ then
2:     return
3: end if
4: $changed \leftarrow$ false
5: for $i \leftarrow 1$ to $s$ do
6:     if $m[i] = 1$ and $n.m[i] = 0$ then
7:         $n.m[i] \leftarrow 1$
8:         $changed \leftarrow$ true
9:     end if
10: end for
11: if $changed$ then
12:     insert_parent_fix($n$.parent(),$n.m$)
13: end if
```

We claim that the resulting tree satisfies the marker bit property. In particular, note that only the marker bits of the inserted node and its ancestors can be potentially affected by the insertion. Lines 1-7 of the `insert_fix` method update the marker bits of the node that is inserted. If the i^{th} marker bit of the node is set, then we check the i^{th} marker bit of its parent node (Lines 6 of the `insert_parent_fix` method). If the i^{th} marker bit of the parent is set, then the i^{th} marker bit of all ancestors will be set because of the marker bit property and nothing more needs to be done for the i^{th} marker bit. Conversely, if the i^{th} marker bit of the parent is not set, then we need to set it and then check the i^{th} marker bit of the parent of the parent node and so on. This is done by Line 7 and the recursive call at Line 12, respectively. The variable *changed* is used to record whether any of the marker bits of the current node have been modified. If the variable is equal to `true`, then the marker bits of the ancestor nodes will not need to be updated. Therefore, the marker bits of the inserted node and its ancestors are updated correctly and the marker bit property is preserved for the updated search tree.

B. Deleting a Node with Less than Two Children

Deleting a node with two children from a binary tree cannot be performed by just connecting the parent of the deleted node to the children of the deleted node because the parent node may end up with three children. Therefore, we will consider two cases: when the deleted node has less than two non-null children and when the deleted node has two non-null children. The first case is explained next, while the second case is explained in Section III-D.

An implementation of Algorithm 3 should be called before a node n with less than two non-null children is deleted. In the algorithm, n`.child()` is used to denote the non-null child of n and $m[i]$ is set when the i^{th} marker bits of the ancestor nodes need to be checked. The algorithm for the method `delete_parent_fix` that updates the marker bits of n's ancestors in the search tree is shown in Algorithm 4.

Algorithm 3 `delete_fix_simple(Node` n`)`

```
1: for $i \leftarrow 1$ to $s$ do
2:     if $n$.data() $\in S_i$ and ($n$ is leaf node or $n$.child().$m[i] = 0$) then
3:         $m[i] \leftarrow 1$
4:     else
5:         $m[i] \leftarrow 0$
6:     end if
7: end for
8: delete_parent_fix($n$.parent(), $m$)
```

Algorithm 4 `delete_parent_fix(Node` n`, Bit[]` m`)`

```
1: if $n = \texttt{null}$ then
2:     return
3: end if
4: $changed \leftarrow$false
5: for $i \leftarrow 1$ to $s$ do
6:     if $m[i] = 1$ and $n$.data()$\notin S_i$ and (n has no other child or $n$.other_child().$m[i] = 0$) then
7:         $n.m[i] \leftarrow 0$
8:         $changed \leftarrow$ true;
9:     end if
10: end for
11: if $changed$ then
12:     delete_parent_fix($n$.parent(), $m$)
13: end if
```

Note that we have used n`.other_child` to denote the child node of n that is not on the path to the deleted node. We claim that the deletion algorithm preserves the marker bit property. In particular, note that only the ancestors of the deleted node can be affected. If $m[i] = 1$ (Line 6 of the `delete_parent_fix` method), then we check whether the data in the node belongs to S_i and whether the $i^{\texttt{th}}$ marker bit of the other child node is set. If both conditions are false, then the only reason the $i^{\texttt{th}}$ marker bit of n is set is because the data in the deleted node belonged to S_i and now this marker bit needs to be reset (Line 7) and the ancestors of n need to be recursively checked (Line 12). Conversely, if one of the conditions is true or $m[i] = 0$, then the $i^{\texttt{th}}$ marker bit of n and its ancestors will not be affected by the node deletion. Therefore, the marker bits of the ancestors of the deleted node are updated correctly and the marker bit property holds for the updated search tree.

C. Element Update

Algorithm 5 should be executed after the data in a node n is modified, where v is the old data value of n. The pseudo-code updates the marker bits of the node n and then calls the `update_parent_fix` method, which is presented in Algorithm 6.

Note that we have used n`.other_child()` to denote the child node of n that is not on the path to the updated node. The method `update_fix` preserves the marker bit

Algorithm 5 `update_fix(Node` n`, Value` v`)`

```
old ← n
for $i = 1$ to $s$ do
    if $n$.data() $\in S_i$ or ($n$.left() $\neq$ null and $n$.left().$m[i] = 1$) or ($n$.right() $\neq$ null and $n$.right().$m[i] = 1$) then
        $n.m[i] \leftarrow 1$
    else
        $n.m[i] = 0$
    end if
    if $n.m[i] = 1$ and $old.m[i] = 0$ then
        $m[i] \leftarrow$ "insert"
    else if $n.m[i] = 0$ and old.$m[i] = 1$ then
        $m[i] \leftarrow$ "delete"
    else
        $m[i] \leftarrow$ "no change"
    end if
end for
update_parent_fix($n$.parent(), $m$)
```

Algorithm 6 `update_parent_fix(Node` n`, Value[]` m`)`

```
if $n =$ null then
    return
end if
changed ← false
for $i = 1$ to $s$ do
    if $m[i] =$ "insert" and $n.m[i] = 0$ then
        $n.m[i] \leftarrow 1$
        changed ← true
    end if
    if $m[i] =$ "delete" and $n$.data() $\notin S_i$ and ($n$.other_child() = null or $n$.other_child().$m[i] = 0$)) then
        $n.m[i] \leftarrow 0$
        changed ← true
    end if
end for
if changed then
    update_parent_fix($n$.parent(), $m$)
end if
```

property because it is a combination of the `insert_fix` and `delete_fix_simple` methods. In particular, $m[i]$ in the method `update_fix` is set to `insert` when the i^{th} marker bit of the updated node was changed from 0 to 1 and to `delete` when this marker bit was updated from 1 to 0. The first case is equivalent to a node with the i^{th} marker bit set being inserted, while the second case is equivalent to a node with the i^{th} marker bit set being deleted.

D. Deleting a Node with Two Children

As it is usually the case ([4]), we assume that the deletion of a node n_1 with two non-null children is handled by first deleting the node after n_1 relative to the tree order, which we will denote as n_2, followed by changing the data value of n_1 to that of n_2. The pseudo-code in Algorithm 7, which implementation should be called after a node is deleted from the tree, shows how the marker bits can be updated, where initially $n = n_1$, p is the parent of n_2, v is the value of the data that was stored in n_2, and $m[i] = 1$ exactly when $n_2.m[i] = 1$ and $n'.m[i] = 0$ for all descendants n' of n_2.

Algorithm 7 `delete_fix_two_children(`n`,`p`,`v`,`m`)`

```
if $p = n$ then
    update_fix($n, v$)
end if
changed ← false
for $i$=1 to $s$ do
    if $m[i] = 1$ and $p$.data() $\notin S_i$ and (p has no other child or $p$.other_child().$m[i] = 0$) then
        $p.m[i] \leftarrow 0$
        changed ← true
    end if
end for
if changed then
    delete_fix_two_children($n$,$p$.parent(), $v, m$)
else
    update_fix($n, v$)
end if
```

In the above code "p has no other child" refers to the condition that p has no other child than the child that it is on the path to the deleted node n_2. Similarly, p`.other_child()` is used to denote the child of p that is not on the path to the deleted node n_2. Note that the above algorithm changes the nodes on the path from n_2 to n_1 using the deletion algorithm from the method `delete_parent_fix` and the nodes on the path from n_1 to the root of the tree using the update algorithm from the method `update_fix` and is therefore correct.

E. Tree Rotation

Most balancing algorithms (e.g., the ones for AVL, red-black, or AA trees) perform a sequence of left and/or right rotations whenever the tree is not balanced as the result of some operation. Here, we will describe how a right rotation can be performed, where the code for a left rotation is symmetric. The pseudo-code in Algorithm 8 should be called with a parent node n_2 and a right child node n_1 after the rotation around the two nodes was performed. The pseudo-code only fixes the marker bits of n_1 and n_2. The descendants of all other nodes will not change and therefore their marker bits do not need to be updated.

F. Time Analysis for the Modification Methods

Obviously, the pseudo-code for the rotation takes constant time. The other methods for updating marker bits visit the updated node and possibly some of its ancestors and perform constant number of work on each node and therefore take order logarithmic time relative to the number of nodes in the tree.

Algorithm 8 `rotate_right_fix`(n_1, n_2)

```
1: for $i \leftarrow 1$ to $s$ do
2:     if $n_1$.data() $\in S_i$ or ($n_1$ has left child and $n_1$.left().$m[i] = 1$) then
3:         $n1.m[i] \leftarrow 1$
4:     end if
5:     if $n_2$.data() $\in S_i$ or ($n_2$ has left child and $n_2$.left().$m[i] = 1$) or ($n_2$ has right child and $n_2$.right().$m[i] = 1$) then
6:         $n2.m[i] \leftarrow 1$
7:     end if
8: end for
```

Therefore, the extra overhead of maintaining the marker bits will not change the asymptotic complexity of the modification operations.

G. Search

Let us go back to our motivating example from Figure 1. Our desire is to efficiently retrieve all rich employees in the tree order. This can be done by repeatedly calling the implementation of the `next` method from Algorithm 9. The terminating condition is when the method returns `null`. The algorithm finds the first node after n, relative to the tree order, that has data that belongs to the set S_i, where d is initially set to `false`.

Algorithm 9 `next`(n,i,d)

```
1: if $n$.data() $\in S_i$ and $d$ then
2:     return $n$
3: end if
4: if $n$.left() is not the last node visited and $n$.left() $\neq$ null and $n$.left().$m[i] = 1$ and $d$ then
5:     return next($n$.left(), $i$, true)
6: end if
7: if $n$.right() is not the last node visited and $n$.right() $\neq$ null and $n$.right().$m[i] = 1$ then
8:     return next($n$.right(), $i$, true)
9: end if
10: if $n$.parent() = null then
11:     return null
12: end if
13: return next($n$.parent(), $i$, $d$)
```

The algorithm first checks if the data in the current node is in S_i and d is true. Note that d becomes true when n is a node that is after the initial node in the search tree. When both conditions are true, we have found the resulting node and we just need to return it. Next, we check the left child node. If we did not just visit it and its i^{th} bit is marked and it is after the start node relative to the in-order tree traversal order, then the subtree with root this node will contain a node with data in S_i that will be the resulting node. Next, we check if the right child has its i^{th} bit marked. This condition and the condition that we have not visited it before guarantees that this subtree will contain the resulting node. Finally, if nighter of the child subtrees contain the node we are looking for, we start checking the ancestor nodes in order until we find an ancestor that has a right child node that we have not visited and its i^{th} marker bit for this child is set. We then visit this subtree because we are guaranteed that it will contain the resulting node. Therefore, the algorithm finds the first node after n that has data that is in S_i. Since, in the worst case, we go up a path in the search tree and then down a path in the search tree, our worst-case asymptotic complexity for finding the next node with data in S_i is logarithmic relative to the size of the tree, which is the same as the asymptotic complexity of the traditional method for finding a next element in a balanced search tree.

Next, we will consider a method that finds all the elements in the search tree without using the `next` method and we will show that this method runs in time that is proportional to the size of the tree. Note that, in the worst case all nodes belong to the query result and therefore we cannot do any better. The algorithm is presented in Algorithm 10. In the method, n is initially the root node of the search tree. Since the method visits every node once, it runs in time that is proportional to the size of the tree. Note that the nodes that are visited by calling the `next` method multiple times are the same as the nodes that are visited by calling the `find_all` method. In both cases, the nodes in the tree are visited relative a in-order traversal of the tree, where subtrees that have root nodes that are unmarked are pruned out.

Algorithm 10 `find_all`(n,i)

```
1: result $\leftarrow \emptyset$
2: if $n$.left() $\neq$ null and $n$.left().$m[i] = 1$ then
3:     result $\leftarrow$ result $\cup$ find_all($n$.left(),$i$)
4: end if
5: if $n.m[i] = 1$ then
6:     result $\leftarrow$ result $\cup \{n\}$
7: end if
8: if $n$.right() $\neq$ null and $n$.right().$m[i] = 1$ then
9:     result $\leftarrow$ result $\cup$ find_all($n$.right(),$i$)
10: end if
11: return result
```

Next, we will describe a search algorithm that can be used to efficiently retrieve the elements of the search tree in an order that this different from the search order. For starters, we present the method `previous` that returns the previous element that belongs to the set S_i. The method is presented in Algorithm 11, where d is initially set to `false`. Note that the method `previous` is analogous to the method `next`. To only difference is that it searches for an element to the left (rather than to the right) of the current element.

Next, we present a method `search` that is also need in order to retrieve the elements of the search tree in an order that is different from the search tree order. The method has the following properties, where we assume that the search tree contains elements with attributes $\{A_i\}_{i=1}^{a}$ and that the ordering of the tree is $\langle A_1$ `asc`, ..., A_a `asc`$\rangle$.

Algorithm 11 `previous`(n, i, d)

```
1: if $n$.data() $\in S_i$ and $d$ then
2:   return $n$
3: end if
4: if $n$.right() is not the last node visited and $n$.right()$\neq$ null and $n$.right().$m[i] = 1$ and $d$ then
5:   return previous($n$.right(), $i$, true)
6: end if
7: if $n$.left() is not the last node visited and $n$.left()$\neq$ null and $n$.left().$m[i] = 1$ then
8:   return previous($n$.left(), $i$, true)
9: end if
10: if $n$.parent() $=$ null then
11:   return null
12: end if
13: return previous($n$.parent(), $i$, $d$)
```

`search`$(A_1, P_1, \ldots, A_l, P_l, dir, r, i)$:

- **pre-conditions:** $l \leq a$ and $r \in [l-1, l]$.
- **return value:** If $dir =$ `asc` ($dir =$ `desc`), then this method returns the the first (last) node n in S_i, relative to the tree order, for which:
  1) $\bigwedge_{i=1}^{r} (n.\mathtt{data}.A_i = P_i)$ and
  2) if $r < l$, then $n.\mathtt{data}.A_l > P_l$ when $dir =$ `asc` and $n.\mathtt{data}.A_l < P_l$ when $dir =$ `desc`.

  The method returns `null` when such a node does not exist.

The method search is used to search for the node that has the same value for some of the attributes as the current node, which allows us to go forward and backwards in the tree and return the elements in the desired order. The pseudo-code for the method when $r = l$ is presented in Algorithm 12. Note that we have added a node n as a parameter, which is initially the root of the search tree. The code when $dir =$ `asc` and $dir =$ `desc` are symmetric. The expression $\langle P_1, \ldots, P_l \rangle \prec n$ returns `true` when $P_j \leq n.\mathtt{data}().A_j$ for $j \in [1 \ldots l]$, but not all inequalities are equalities. Similarly, the expression $n = \langle P_1, \ldots, P_l \rangle$ returns `true` when $P_j = n.\mathtt{data}().A_j$ for $j \in [1 \ldots l]$.

The code first checks if the element that we are searching for is strictly in the left subtree (Lines 2-4) or in the right subtree (Lines 5-7). Of course, the subtrees are only considered if the appropriate marker bit is set. If both if statements fail (on Lines 2 and 5), then the current node has values $P_1, \ldots, P_l$ for the attributes $A_1, \ldots, A_l$, respectively. If there is a node in the left subtrees with these values, then we recursively call the method on the left subtree. Otherwise, we simply return the current root node n. The algorithm finds the first node with the desired property and therefore is correct. At each step, the algorithm considers only the left or the right subtree and therefore it runs in logarithmic time relative to the size of the tree.

Algorithm 12 `search1`$(n, A_1, P_1, \ldots, A_l, P_l, dir, l, i)$

```
1: if $dir =$ asc then
2:   if $n$.left() $\neq$ null and $n$.left().$m[i] = 1$ and $\langle P_1, \ldots, P_l \rangle \prec n$ then
3:     return search1($n$.left(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)
4:   end if
5:   if $n$.right() $\neq$ $null$ and $n$.right().$m[i] = 1$ and $n \prec \langle P_1, \ldots, P_l \rangle$ then
6:     return search1($n$.right(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)
7:   end if
8:   if $n \neq \langle P_1, \ldots, P_l \rangle$ then
9:     return null
10:  end if
11:  if search1($n$.left(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)=null then
12:    return $n$
13:  end if
14:  return search1($n$.left(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)
15: end if
16: if $n$.right() $\neq$ null and $n$.right().$m[i] = 1$ and $n \prec \langle P_1, \ldots, P_l \rangle$ then
17:   return search1($n$.right(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)
18: end if
19: if $n$.left() $\neq$ $null$ and $n$.left().$m[i] = 1$ and $\langle P_1, \ldots, P_l \rangle \prec n$ then
20:   return search1($n$.left(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)
21: end if
22: if $n \neq \langle P_1, \ldots, P_l \rangle$ then
23:   return null
24: end if
25: if search1($n$.right(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)=null then
26:   return $n$
27: end if
28: return search1($n$.right(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $l$, $i$)
```

We will next consider the `search` method when $r = l - 1$. We will only show the code for when $dir_l = asc$, where the other case is symmetric. The pseudo-code is presented in Algorithm 13. Again n is initially the root node of the tree. The expression $\langle P_1, \ldots, P_l \rangle \preceq n$ returns `true` when $P_j \leq n.\mathtt{data}().A_j$ for $j \in [1 \ldots l]$. The algorithm first checks to see if the resulting node is the right subtree (Lines 2-4). If not, then it checks the left subtree (Lines 5-7). If both options fail, then the algorithm checks if the current root node passes the condition. If it does, then it returns it (Line 11). If it does not, then it returns `null` (Line 9). Therefore, the algorithm finds the correct node. The algorithm runs in logarithmic time on a balanced tree because it calls itself recursively on either the left or right subtree, but not both.

Algorithm 13 `search2`(n, A_1, P_1, ..., A_l, P_l, dir, r, i)

```
1: if $dir$ = asc then
2:     if $n$.right() $\neq$ $null$ and $n$.right().$m[i] = 1$ and $n \preceq \langle P_1, \ldots, P_l \rangle$ then
3:         return search2($n$.right(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $r$, $i$)
4:     end if
5:     if $n$.left() $\neq$ null and $n$.left().$m[i] = 1$ and search2($n$.left(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $r$, $i$) $\neq$ $null$ then
6:         return search2($n$.left(), $A_1$, $P_1$, ..., $A_l$, $P_l$, $dir$, $r$, $i$)
7:     end if
8:     if $n \neq \langle P_1, \ldots, P_{l-1} \rangle$ or $n$.data().$A_l \not\geq P_l$ then
9:         return null
10:    end if
11:    return $n$
12: end if
13: ...
```

We will next show how the elements of the search tree can be efficiently retrieved in the order $\langle A_1\ dir[1], \ldots, A_a\ dir[a] \rangle$ where $dir[j] \in \{\texttt{asc}, \texttt{desc}\}$ for $j \in [1 \ldots a]$. The algorithm is presented in Algorithm 14, where the method will return only elements that belong to the set S_i. Note that the variables $\{dir[i]\}_{i=1}^{a}$ are parameters to the method. However, they have been omitted from the algorithm in order to keep the code simpler and more understandable.

Algorithm 14 `ordered_next`(n,i)

```
1: if $dir[a]$=asc then
2:     $n' \leftarrow$ next($n, i,$ false)
3: end if
4: if $dir[a]$=desc then
5:     $n' \leftarrow$ previous($n, i,$ false)
6: end if
7: if $n' \neq$ null and $\bigwedge_{i=1}^{a-1} (n.A_i = n'.A_i)$ then
8:     return $n'$
9: end if
10: if $a = 1$ then
11:     return null
12: end if
13: $n' \leftarrow$ search($A_1, n.A_1, \ldots, A_{a-1}, n.A_{a-1}, dir[a-1], a-2, i$)
14: return next_up($n, n', i, a-1$)
```

Note that we have used $n.A_j$ to denote the value of the j^{th} attribute of the element that is stored in node n. The method calls the `next_up` method, which in tern can call the `next_down` method. The methods are presented in Algorithms 15 and 16, respectively.

Algorithm 15 `next_up`(n,n',i,l)

```
1: if $n'$ = null then
2:     if $l = 1$ then
3:         return null
4:     end if
5:     $n' \leftarrow$ search($A_1, n.A_1, \ldots, A_{l-1}, n.A_{l-1}, dir[l-1], l-2, i$)
6:     return next_up($n, n', i, l-1$)
7: end if
8: return next_down($n', i, l$)
```

Algorithm 16 `next_down`(n,i,l)

```
1: if $n$ = null then
2:     return null
3: end if
4: if $l = a$ then
5:     return $n$
6: end if
7: $n' \leftarrow$ search($A_1, n.A_1, \ldots, A_l, n.A_l, dir[l+1], l, i$)
8: return next_down($n', i, l+1$)
```

Consider first Lines 1-6 of the method `ordered_next`. The code first checks whether the next node n' relative to the order $\langle A_1\ dir[1], \ldots, A_{a-1}\ dir[a-1] \rangle$ has the same values for the attributed $A_1, \ldots, A_{a-1}$ as n. If this is the case, then only this node needs to be returned. If this is not the case and $a = 1$ (Lines 10-12), then a "next" node does not exist and the method returns `null`. If this is not the case and $a > 1$, then Line 13 of the code looks for the first node that has the same value as n for the attributes $\{A_i\}_{i=1}^{a-2}$ and a value for the attribute A_{a-1} that is right after the value of the attribute A_{a-1} for n. In Line 14 of the code the `next_up` method is called with the element that is found in the previous line, where the value for n' is `null` when such an element does not exist.

The method `next_up` looks for a node n' that has the property that $\bigwedge_{i=1}^{l-1} (n.A_i = n'.A_i)$ and the value for A_l of n' is right after the value for A_l of n. When this is the case, then the `next_down` method is executed. It finds the first element in the search tree for which $\bigwedge_{i=1}^{l-1} (n.A_i = n'.A_i)$ holds. This is indeed the node that needs to be returned. When n' does not have the desired property, l is decremented by 1 and `next_up` is called recursively. If l becomes equal to 0, then a "next" tuple does not exist and the method `next_up` returns `null`. Since the *next* method goes up and down a path in the three, its running time is logarithmic.

IV. Conclusion and Future Research

We introduced MB-trees and showed how they are beneficial for accessing predefined subsets of the tree elements. MB-trees use marker bits, which add only light overhead to the different operations and do not change the asymptotic complexity of the operations. An obvious application of MB-trees is merging search trees by removing redundant data, which can result in faster updates because fewer copies of the redundant data need to be updated. In addition, we showed how elements in different orders can be retrieved from a search tree. Again, the application is index merging because fewer indices can efficiently answer the same set of queries.

One obvious area for future research is showing that the algorithm for retrieving all the elements of a search tree that belong to a set S_i in a non-trivial order takes time that is proportional to the size of the tree. Another contribution would be to present algorithms that extend our algorithms to secondary storage structures, such as B and $B+$ trees.

References

[1] G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Soviet Math. Doklady*, vol. 3, pp. 1259–1263, 1962.

[2] A. Andersson, "Balanced search trees made simple," *Workshop on Algorithms and Data Structures*, pp. 60–71, 1993.

[3] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, vol. 1, no. 3, 1972.

[4] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. McGraw Hill, 2002.

[5] L. Stanchev, "Efficient Access to Non-Sequential Elements of a Search Tree," *The Third International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA 2011*, pp. 181–185, January 2011.

[6] L. Stanchev and G. Weddell, "Saving Space and Time Using Index Merging," *Elsevier Data & Knowledge Engineering*, vol. 69, no. 10, pp. 1062–1080, 2010.