Stanchev. Published in the Third International Conference on Advances in Databases, Knowledge, and Data Applications Proceedings: January 23-28, 2011, St. Maarten.

Efficient Access to Non-Sequential Elements of a Search Tree

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

***Abstract*—This article describes how a search tree can be extended in order to allow efficient access to predefined subsets of the stored elements. This is achieved by marking some of the elements of the search tree with marker bits. We show that our approach does not affect the asymptotic logarithmic complexity for existing operations. At the same time, it is beneficial because the modified search tree can now efficiently support requests on predefined subsets of the search elements that it previously could not.**

***Keywords*-marker bits; search trees; data structures**

I. Introduction

A balanced search trees, such as an AVL tree ([1]), an AA tree (see [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. For example, consider a database that stores information about company employees. A search tree may store information about the employees ordered by age. 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.

The technique that is 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.

Naïve solutions to the problem fail. For example, it is not enough to mark all the nodes of the search tree that contain data elements that belong to subsets of the data that we are interested in. This approach will not allow us to prune out any subtrees because it can be the case that the parent node does not belong to an interesting subset, but the child nodes do.

To the best of our knowledge, detailed explanation of how marker bits work have not been previously published. Our previous work [5] briefly introduces the concept of marker bits, but it does explain how marker bits can be maintained after insertion, deletion and 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.

Given a subset of the search elements S, our approach 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.

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

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 a total order 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 trivially extended to allow an MB-tree to have multiple data values in a node, as is the case for a B Tree, but this is beyond the scope of this paper.

Going back to our motivating example, consider the MB-tree $\langle\langle RICH\_EMPS\rangle, EMPS, \langle age\rangle\rangle$. This represents a search tree of the employees, where the ordering is relative to the attribute *age* 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.

Each node in the MB-tree contains the name of the employee, their age, and their salary. Above each node the value of the marker bit is denoted, where the bit is set

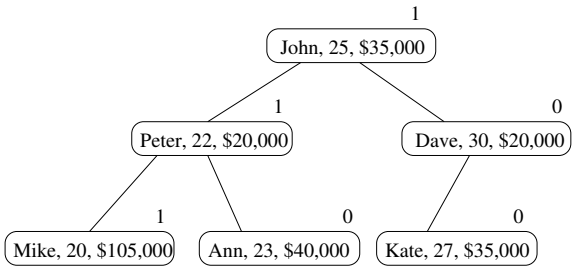

Figure 1. Example of an MB-tree

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

Table I
INTERFACE OF A NODE

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.

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. Non-binary trees will be considered in Section IV. We will assume that every node of the search tree supports the methods of the interface 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 search and update can be extended in the presence of marker bits.

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 ← 1 to s do
2:    if n.data() ∈ S_i then
3:       n.m[i] ← 1
4:    else
5:       n.m[i] ← 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 function `insert_parent_fix` fixes the marker bits of the ancestors of the inserted node, where the later is presented in Algorithm 2.

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

```
1: if n = null then
2:    return
3: end if
4: changed ← false
5: for i ← 1 to s do
6:    if m[i] = 1 and n.m[i] = 0 then
7:       n.m[i] ← 1
8:       changed ← 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. 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 changed. If the variable is not changed, 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 holds 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 bit 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 ← 1 to s do
2:    if n.data() ∈ S_i and (n is leaf node or n.child().m[i] = 0) then
3:       m[i] ← 1
4:    else
5:       m[i] ← 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 = null then
2:    return
3: end if
4: changed←false
5: for i ← 1 to s do
6:    if m[i] = 1 and n.data() ∉ S_i and (n has no other child or n.other_child().m[i] = 0) then
7:       n.m[i] ← 0
8:       changed ← 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^{th} marker bit of the other child node is set. If both conditions are false, then the only reason the i^{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 unset (Line 7) and the ancestors of n needs to be recursively checked (Line 12). Conversely, if one of the conditions is true or $m[i] = 0$, then the i^{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.

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

```
1: old ← n
2: for i = 1 to s do
3:    if n.data() ∈ S_i or (n.left() ≠ null and n.left().m[i] = 1) or (n.right() ≠ null and n.right().m[i] = 1) then
4:       n.m[i] ← 1
5:    else
6:       n.m[i] = 0
7:    end if
8:    if n.m[i] = 1 and old.m[i] = 0 then
9:       m[i] ← "insert"
10:   else if n.m[i] = 0 and old.m[i] = 1 then
11:      m[i] ← "delete"
12:   else
13:      m[i] ← "no change"
14:   end if
15: end for
16: update_parent_fix(n.parent(), m)
```

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.

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

```
1: if n = null then
2:    return
3: end if
4: changed ← false
5: for i = 1 to s do
6:    if m[i] = "insert" and n.m[i] = 0 then
7:       n.m[i] ← 1
8:       changed ← true
9:    end if
10:   if m[i] = "delete" and n.data() ∉ S_i and (n.other_child() = null or n.other_child().m[i] = 0)) then
11:      n.m[i] ← 0
12:      changed ← true
13:   end if
14: end for
15: if changed then
16:    update_parent_fix(n.parent(), m)
17: 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 updated node. The method `update_fix` preserves the marker bit

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 marking 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 for all descendants of n_2, $m[i] = 0$.

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

1: **if** $p = n$ **then**
2: `update_fix(`n, v`)`
3: **end if**
4: *changed* $\leftarrow$ **false**
5: **for** i=1 **to** s **do**
6: **if** $m[i] = 1$ **and** p.`data()` $\notin S_i$ **and** (p has no other child **or** p.`other_child()`$.m[i] = 0$) **then**
7: $p.m[i] \leftarrow 0$
8: *changed* $\leftarrow$ **true**
9: **end if**
10: **end for**
11: **if** *changed* **then**
12: `delete_fix_two_children(`n, p.`parent()`$, v, m$`)`
13: **else**
14: `update_fix`(n, v)
15: **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 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 right child node n_1 after the rotation around the two nodes was performed.

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**

The above 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 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. 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 that is n or that is after n, relative to the tree order, and that has data that belongs to the set S_i, where d is initially set to `false`.

The algorithm first checks if the data in the current node is in S_i. If it is, then 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

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

```
1: if (n.data() ∈ S_i) then
2:    return n
3: end if
4: if n.left() is not the last node visited and n.left() = 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() = 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)
```

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 starting with n that has data 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 search tree.

IV. Conclusion

We introduced MB-trees and showed how they are beneficial for accessing predefine 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.

References

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

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

[3] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3), 1972.

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

[5] L. Stanchev and G. Weddell. Saving Space and Time Using Index Merging. *Elsevier Data & Knowledge Engineering*, 69(10):1062–1080, 2010.