EFFICIENT MULTIPLE SEARCH TREE STRUCTURE

Mohammad Reza Ghaeini\textsuperscript{1} and Mohammad Reza Mirzababaei\textsuperscript{2}

\textsuperscript{1}Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran
\texttt{mr.ghaeini@aut.ac.ir}

\textsuperscript{2}Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran
\texttt{mirzababaei@aut.ac.ir}

ABSTRACT
This paper describes an efficient multiple search tree structure instead of binary search tree. The search tree is more robust against different sequential of numbers and more balance tree. Our tree height is less than BST (near to half); therefore, our search tree is faster in searching. In deletion, our search tree is more stable and doesn’t need reconstruction, so the algorithm done in $O(\log(n))$.

KEYWORDS
Tree, Binary search tree, Search Structure, Sigmoid function

1. INTRODUCTION
A good data structure searches quickly, uses less memory, and reasonable construction time. We have many either linear or non-linear data structures used in computer science. The tree is one of the non-linear fundamental. We use trees in compiler design, text processing and searching algorithms [1].

Trees made up from some nodes and any node has some child nodes. Tree has a main node as root its. Trees have many types and usages; one of those is the binary search tree [2]. In binary search tree, any node has zero, one or two child nodes. We have a policy in nodes, all of the left subtree keys are less than the parent’s key and all of the right subtree keys are bigger than the parent’s key. In best case, the height of the binary search tree is $\log_2 n$ and the search algorithm has $O(\log(n))$ time complexity. Tree making time complexity according to Knuth proof equals to $O(n\log(n))$ [3]:

$$\sum_{k=1}^{n} \log(k) = \log(n!) = n \log(n) - n \log(e) + \theta(\log(n)) \approx n \log(n) - 1.442695 n$$  \hspace{1cm} (1)

In worst case, the height of binary search tree is $n$ and $O(n)$ time is the complexity for searching. So the tree making time complexity is equal to $O(n^2)$:

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \frac{n^2 + n}{2}$$  \hspace{1cm} (2)

Despite the wide popularity of binary search tree, unfortunately it has few serious problems. First, its shape depends on the nature of the input. Second, its shape depends on the deletes and inserts too [4].

According to the importance and usage of binary search tree we concluded that finding an efficient search tree is so valuable.
2. **Related Works**

The shape of the binary search tree depends on the nature of the input but some researches have been done to warranty the shape regular tree for any input, e.g.:

- AVL Trees with Relaxed Balance [5].
- The Suffix Binary Search Tree and Suffix AVL Tree [6].
- An Algorithm for the Organization of Information, (AVL Tree) [7].
- Optimally Grown Binary Trees with a Sorting Algorithm [8].
- Self-Adjustable Binary Search Trees [9].

Another type of those trees is the chromatic trees, which studied by Nurmi and Soisalon-Soininen [10].

- Chromatic binary search trees: A structure for concurrent rebalancing [10].
- Amortization results for chromatic search trees, with an application to priority queues [12].
- Left-leaning Red-Black Trees [13].
- Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms [14].

Below papers describe non-blocking binary search tree in an asynchronous shared-memory system using single-word compare-and-swap operations.

- Non-blocking Binary Search Tree [15].
- Non-blocking k-ary Search Tree [16].

One of tree problems is its traversal, for binary search tree we have three common traversal methods: Inorder, Preorder, and Postorder. These methods commonly implement recursively. But in some situations we want to implement these methods non-recursively, below paper presents a non-recursive algorithm for reconstructing a binary tree.

- Modified Non-Recursive Algorithm for Reconstructing a Binary Tree [17].

The other papers try to present new tree or optimize binary search tree.

- On Dynamic Optimality for Binary Search Trees [18].
- A Forest of Hashed Binary Search Trees with Reduced Internal Path Length and better Compatibility with the Concurrent Environment [4].
- An insertion algorithm for a minimal internal path length binary search tree [19].
- An Empirical Study of Insertion and Deletion in Binary Search Trees [20].

Below paper is about one of the binary search tree usages.

- The research of quadtree search algorithms for anti-collision in radio frequency identification systems [21].

This paper mainly is about RFID (Radio Frequency Identification); the collision resolution is the most important issues in RFID system that affects the data integrity [21]. One of the anti-collision algorithms is binary search tree that is used in this paper.

This area is so wide, but we wouldn’t elaborate any further.
3. **MULTIPLE SEARCH TREE STRUCTURE**

In this structure every node store three keys and three Booleans. In every node, we have a sigmoid function that classifies numbers to blocks and buckets. This function is:

\[
y(x) = \frac{\text{size}}{1 + e^{-\left(\frac{x - \text{middle}}{w}\right)}}
\]

(3)

Node function determines bucket that searched key can be there. The tree policy is the node function, each key that classify in a bucket, stores in that bucket or in bucket subtree.

### 3.1. Algorithm Specification

First middle and size equal to zero and w equals to one. In first number addition, we store it in first position; size variable sets to one and its filling flag sets. In second number addition, we sort these data and store them in increasingly sorted order and set size variable to two, now we make middle equal to average of them. In third number addition, after sorting, storing and setting size, we make middle equal to average of maximum and minimum, next,

\[
w = \frac{(\text{node}[2] - \text{node}[0])}{1.37}
\]

(4)

We divide sigmoid function to three area, first area less than 1/3, second area between 1/3 and 2/3, and third area bigger than 2/3. Calculation shows that length of middle area is near to 1.38 (Equation 5) and for avoiding marginal position, we use 1.37 in our calculation. By w, we map middle data between maximum and minimum in middle bucket and middle area.
\[
y(x) = \frac{1}{1 + e^{-x}} \quad \& \quad x(y) = -\ln\left(\frac{1-y}{y}\right) \rightarrow x\left(\frac{1}{3}\right) \approx -0.68 \quad \& \quad x\left(\frac{2}{3}\right) \approx 0.68 \quad (5)
\]

When number addition in filled bucket occurred, we call subtree to do number addition request. For searching, in any node we find the bucket that our data can be in there, if our data not equals to node bucket data, we continue searching in bucket subtree. For deletion, first we find that number, after finding, we delete the data and continue traversal of that subtree arbitrarily to reach a leaf (a node that don’t have any child), and exchange an arbitrary leaf bucket number with the deleted bucket number and reset its flag, and if the deleted bucket haven’t subtree, we only delete that number and reset its flag.

3.2. Pseudo Code

middle ← 0
w ← 1
size ← 0
child[3] ← {null, null, null}
node[3]
isFill[3] ← {false, false, false}

add(number):
if (size == 0)
    then
        node[0] ← number
        isFill[0] ← true
        size ← 1
    else if (size == 1)
        then
            if (number ≥ node[0])
                then
                    node[1] ← number
                else
                    node[1] ← node[0]
                    node[0] ← number
                    middle ← (node[1]+node[0])/2
                    isFill[1] ← true
                    size ← 2
            else if (size == 2 node[1])
                then
                    node[2] ← number
                else (if number ≥ node[0])
                    then
                        node[1] ← number
                    else
                        node[1] ← node[0]
                        node[0] ← number
                        middle ← (node[2] + node[0])/2
                        w ← (node[2] – node[0])/1.37
                        isFill[3] ← true
                        size ← 3
            else
                bn ← getBucketNumber(number)
                if (isFill[bn])
                    then
                        child[bn].add(number)
                    else
                        node[bn] ← number

isFill[bn] ← true

---------------------------------------------------------------------
getBucketNumber(number):

\[
result \leftarrow \frac{1}{1 + e^{-\left(\frac{\text{number}-\text{middle}}{w}\right)}}
\]

return result

findNumber(number):

bn \leftarrow \text{getBucketNumber}(\text{number})
if (\text{node}[\text{bn}] == \text{number}) then
    \text{return true}
else
    if (\text{child}[\text{bn}] \neq \text{null}) then
        \text{return child[bn].findNumber(number)}
    else
        \text{return false}

delete(number):

\text{isDeleted} \leftarrow \text{false}
bn \leftarrow \text{getBucketNumber}(\text{number})
if (\text{node}[\text{bn}] \neq \text{number}) then
    if (\text{child}[\text{bn}] \neq \text{null}) then
        \text{return child[bn].delete(number)}
    else
        \text{force} \leftarrow \text{bn}
        \text{temp} \leftarrow \text{child}[\text{bn}]
        if (\text{temp} \neq \text{null}) then
            while (\text{temp.child}[0] \neq \text{null} || \text{temp.child}[1] \neq \text{null} || \text{temp.child}[2] \neq \text{null}) do
                if (\text{temp.child}[\text{force}] \neq \text{null}) then
                    \text{temp} \leftarrow \text{temp.child}[\text{force}]
                else if (\text{temp.child}[0] \neq \text{null}) then
                    \text{temp} \leftarrow \text{temp.child}[0]
                    \text{force} \leftarrow 0
                else if (\text{temp.child}[1] \neq \text{null}) then
                    \text{temp} \leftarrow \text{temp.child}[1]
                    \text{force} \leftarrow 1
                else if (\text{temp.child}[2] \neq \text{null}) then
                    \text{temp} \leftarrow \text{temp.child}[2]
                    \text{force} \leftarrow 2
                \text{temp.size} \leftarrow \text{temp.size} - 1
                \text{node}[\text{bn}] \leftarrow \text{temp.node}[\text{temp.size}]
            else
                \text{isFill} \leftarrow \text{false}
        \text{return true}
    return false

4. IMPLEMENTATION AND SIMULATION

We implement our data structure by JAVA. The system properties are:
Processor: Intel Core 2 Duo E4600, 2.40GHz / Memory: 2*1GB DDR2 800.

In this implementation we study two feathers, Tree making time and Tree Height. To decrease system errors and input situations, we tested each dataset test 10 times, and accepted the
minimum, we examined 10 different dataset in every step and declared average of them as result of that step. (Datasets make absolutely randomly)

In implementation, we can use some optimization, e.g., we can determine number of bucket for data without sigmoid function calculation, we can determine bucket with e power amount.

Time of making tree is important too, finding and allocating memory affect to this time, in binary search tree we have more extra and useless allocated memory, therefore, making that is slower (Figure 3).

In any node, probable bucket of numbers and keys determine in one action and while each node doesn’t fill, we stay in that node and don’t make a new node. We have local and flexible policy in each node; so, our tree is more robust against different natural of input and sequence of numbers.

5. COMPARISON

In best case, our tree height is $\log_3 n$ and $O(\log(n))$ time complexity for searching. Making time complexity according to Knuth proof equals to $O(n(\log(n)))$ [3]:

$$\sum_{k=1}^{n} \log(k) = \log(n!) = n \log(n) - n \log(e) + O(\log(n)) \approx n \log(n) - 1.442695 n$$ (6)

In worst case, our search tree height is $n/3$ and $O(n)$ time complexity for searching. Making time complexity equals to $O(n^2)$:

$$\sum_{k=1}^{n} k = \frac{1 + 2 + 3 + \ldots + n}{3} = \frac{n}{3} \left( \frac{n + 1}{2} \right) = \frac{n^2 + 3n}{6} = \frac{n^2 + 3n}{18}$$ (7)

In average case, generally, we can say it is similar to best case and worst-case occurrence possibility is too low.

Because of similarity between this search tree and binary search tree, we can design and implement some algorithms that belong to binary search tree for our search tree, E.g. traversal algorithms.

According to lower height of our tree, certainly, we can expect lower time and fewer cycle for searching in this search tree.

In this search tree we have some advantages including: lower searching time, lower deleting time, stable shape after deleting, lower tree height, lower tree making time, more robust in different natural of input (by local policy in any node), and efficient using of memory (lower useless allocated memory). So, our search tree is efficient substitute for binary search tree.
6. CONCLUSIONS

According to the points which mentioned in last of comparison section, we can seriously say that our search tree has many advantages in each usages and we can substitute that with binary search tree.

7. FUTURE WORKS

The most important issues and future works are using this search tree in problems and applications to achieve its optimization and its advantages.

REFERENCES


