SN Computer Science (2022) 3: 367
https://doi.org/10.1007/s42979-022-01285-9

ORIGINAL RESEARCH

A Fast Parallelizable Algorithm for Constructing Balanced Binary Search Trees

Pavel S. Ruzankin[1]

Received: 28 August 2021 / Accepted: 1 July 2022 / Published online: 14 July 2022

Abstract
We suggest a new non-recursive algorithm for constructing a balanced binary search tree given an array of numbers. The algorithm has $O(N)$ time and $O(1)$ auxiliary memory complexity if the given array of N numbers is sorted. The resulting tree is of minimal height and can be transformed into a complete binary search tree while retaining minimal height with $O(\log N)$ time and $O(1)$ auxiliary memory. The algorithm allows simple and effective parallelization resulting in time complexity $O((N/s) + s + \log N)$, where s is the number of parallel threads.

Keywords Binary search tree · Parallel algorithm

A binary search tree (BST) is a fundamental data structure which is widely used in applications. There is a large variety of algorithms for constructing BSTs. A first approach is based on sequentially adding nodes to the tree. The nodes may be added to leaves [5] or to the root of the tree [7]. A second approach consists in reconstructing a BST from preorder or postorder traversals (e.g., see [1, 2] and references therein). A third approach is based on halving the given sorted array and recursively building the left and the right subtrees [9]. There are also algorithms that account for the probabilities of hitting specific nodes and try to build optimal BSTs (e.g., see [4] and references therein).

There also exist algorithms that do not adhere to any of the approaches above. The recursive algorithm in [8] constructs the tree by sequentially constructing perfect BSTs. After a perfect BST is constructed, it is incorporated into the new BST as the left subtree of the root. Then the new BST is transformed into a perfect BST, and so on.

We present a new non-recursive algorithm for constructing a binary search tree. The algorithm has $O(N)$ time and $O(1)$ auxiliary memory complexity if the given array of N numbers is sorted. We use an array-based representation of the BST. The $O(1)$ auxiliary memory complexity means that, except for the resulting arrays used to store the tree, we need $O(1)$ auxiliary memory. If a link-based representation is needed, then the algorithm will additionally need $O(N)$ auxiliary memory. The resulting BST has minimal height. However, it may fail to be balanced in the sense of AVL trees, i.e., the trees where the heights of the two child subtrees of each node differ by at most one. The new algorithm, though being non-recursive, somewhat resembles the recursive algorithm in [8]. Moreover, we can use the rotations algorithm from [8] to make the BST complete (i.e., make all the levels, except possibly the lowest one, completely filled) while retaining the minimal height, which needs $O(\log N)$ time and $O(1)$ auxiliary memory.

As mentioned above, we assume that the given array of N numbers is sorted. To simplify notations, we will build a BST for the numbers $0, \dots, N-1$. This will not limit generality even when the keys are not pairwise distinct.

Our algorithm is substantially based on the binary representation of a number. We will mark the binary numbers with a leading zero to distinguish them from decimal ones; e.g., $2 = 010$.

First, let us consider the case when $N = 2^K - 1$ for some integer $K \geq 1$. In this case, the minimal-height BST is perfect. For example, for $K = 4$, the tree is shown in Fig. 1.

Let the level of a node be the distance from the node to a nearest leaf in the perfect BST. That is, in a perfect BST, the leaves lie on level 0, the parents of the leaves lie on level 1, and so on. Note that in our case the level of a node depends on the number of the node only and does not depend on the height of the tree.

✉ Pavel S. Ruzankin
ruzankin@math.nsc.ru

[1] Sobolev Institute of Mathematics, Novosibirsk, Russia

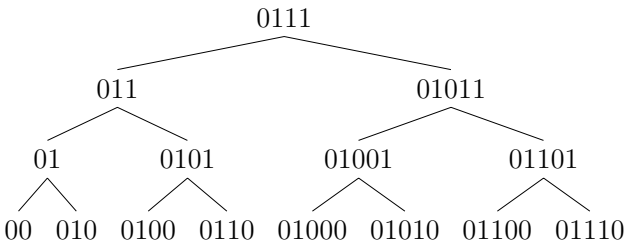

Fig. 1 The perfect binary search tree for the binary numbers from 0 to $N-1=2^4-2$

We see that the binary representations of nodes of level k end with a zero and plus k ones, since subsequent nodes on level k differ by 2^{k+1}. Thus, the level $L(j)$ of a node j can be calculated as the location of the least significant zero in the binary representation of j.

To calculate $L(j)$, one may use the operation of the least significant one in a binary number, which is implemented in many modern processor architectures [3]. There are also built-in functions for the operation in popular compilers. For instance, in GCC, $L(j)$ can be computed as `__builtin_ffs(~j)-1` (see [6]). We assume that the binary representation of $N-1$

contains at least one zero, which is the case when N can be represented as the number of the same unsigned integer type as is used for indexing the cells of the given sorted array of numbers.

However, Algorithms 1 and 2 below utilize not $L(j)$ itself but $2^{L(j)}$, and Algorithm 3 below can be obviously modified to use $2^{L(j)}$ instead of $L(j)$ if needed. It is well known [3] that

$$2^{L(j)} = (j+1)\&(-(j+1))$$

or

$$2^{L(j)} = (\sim j)\&(-(\sim j)),$$

where & is the bitwise AND operator, $\sim$ is the bitwise NOT operator, $-j$ is the negative of j treating j as a signed integer in two's complement arithmetic, which is common in modern processors. For instance, in R, $2^{L(j)}$ can be computed as `bitwAnd((j+1L),-(j+1L))`.

A node j on level $k \geq 1$ has the left child $j-2^{k-1}$ and the right child $j+2^{k-1}$. Moreover, a node j on level $k \geq 0$ has the parent $j+2^k$ if the binary representation of j ends with "$001\cdots1$" (ending with k ones) and the parent $j-2^k$ if the binary representation of j ends with "$101\cdots1$" (ending with k ones).

Thus, for the case $N=2^K-1$, we can write down the algorithm as the following pseudocode. Below, p, l, r denote the resulting arrays of parents, left children, and right children, respectively. The algorithm constructs the BST as these three arrays. t stands for the number of the root node. $M(j)$ denotes the location of the most significant one in the binary representation of j, e.g., $M(01001)=3$. For instance, in GCC, `__builtin_clz()` function may be used for computing $M(j)$ [6]. Unlike $L(j)$, the function $M(j)$ has no known representations in terms of common bitwise operations.

Algorithm 1.

```
for (j in 0,...,N-1)
    if ((j & 2^{L(j)+1}) = 0)
        p[j] := j + 2^{L(j)}
    else
        p[j] := j - 2^{L(j)}
    end if
    if (L(j) > 0)
        l[j] := j - 2^{L(j)-1}
        r[j] := j + 2^{L(j)-1}
    else
        l[j] := NULL
        r[j] := NULL
    end if
end for
t := 2^{M(N)} - 1
p[t] := NULL
```

Now it remains to modify Algorithm 1 for the case of arbitrary N. If we try to build a binary tree with Algorithm 1, then some edges may point to missing nodes that are greater than $N-1$.

Lemma . *All the edges pointing to missing nodes in the "tree" built by Algorithm* 1, *except the down-right edge of the last node if any, are located on the ascending path from the node* $(N-1)$ *to the root in the perfect BST of the same height.*

Proof Let us have the "tree" constructed by Algorithm 1. Let a node j, $j \neq N-1$, of the "tree" have its down-right edge pointing to a missing node i. We have $j < N-1 < i$. Let k be the level of the node j, and let m be the ancestor of the node $(N-1)$ on level k in the corresponding perfect BST. Then we have $m \geq j$, since $j < N-1$. Besides, we cannot have $m > j$ since it would imply $N-1 > i$. Hence, $m=j$, and the ancestor of the node $(N-1)$ at level $k-1$ in the corresponding perfect BST is i since $j < N-1 < i$.

Let now a node j of the "tree" constructed by Algorithm 1 have its up edge pointing to a missing node i, let k be the level of the node j, and let m be the ancestor of the node $(N-1)$ on level k in the corresponding perfect BST. Then again $j < N-1 < i$ and $m \geq j$, and again we cannot have $m > j$ since it would imply $N-1 > i$. Hence, $m=j$.

The lemma is proved.

To correct the "tree" built by Algorithm 1, it remains to follow the descending path from the root to the node $(N-1)$ in the corresponding perfect BST and merge edges pointing to missing nodes. Finally, the algorithm, which will be explained below, is as follows. Below, / denotes integer division, e.g., $1/2 = 0$.

Algorithm 2.

```
function P(j) :=
    if ((j & 2^{L(j)+1}) = 0) then j + 2^{L(j)}
    else j − 2^{L(j)}

for (j = 0 to N − 1 by 2)
    p[j] := P(j)
    l[j] := NULL
    r[j] := NULL
end for
for (j = 1 to N − 1 by 2)
    p[j] := P(j)
    l[j] := j − 2^{L(j)−1}
    r[j] := j + 2^{L(j)−1}
end for
r[N − 1] := NULL
t := 2^{M(N)} − 1
p[t] := NULL

k := 2^{L(t)}
j := t
while (k > 2^{L(N−1)})
    k := k/2
    if (((N − 1 − t) & k) = 0)
        k := k/2
        while (((N − 1 − t) & k) = 0)
            k := k/2
        end while
        r[j] := j + k
        p[j + k] := j
    end if
    j := j + k
end while
```

The time complexity is still $O(N)$, since merging edges after the `for` loops takes $O(\log N)$ time and $O(1)$ auxiliary memory.

The `while` loops for merging edges are explained as follows. Traveling by the descending path from the root node t to the node $(N-1)$, we move by $\pm 2^{L(j)-1}$ when we go from the node j to its right/left child. Thus, $N-1-t = A-B$, where A is the binary number with 1's on locations $m = L(j)-1$ such that the path contains the edge from j to its right child; analogously, B is the binary number that has 1's on locations m such that the path contains the edge from j to its left child, $m = L(j)-1$. The path goes through the nodes $> N-1$ when it contains a subpath with the edges down right–left–left–⋯–left with the next edge being down-right or with the last edge of the subpath being the last edge of the path. Only the first and the last node of the subpath are $\le N-1$. So we must merge each such subpath into one edge. Let m and n be the levels of the first and the last node of the subpath. Then $N-1-t$ will contain the following binary digits at the locations $m-1,\dots,n$: $100\cdots0-011\cdots1 = 00\cdots01$. The `while` loops just search for all such subpaths (all such patterns in $N-1-t$) and connect their first and last nodes with an edge.

Remark 1 Algorithm 2 allows effective straightforward vectorization and/or parallelization. The only loop that cannot be vectorized or parallelized is the loop correcting edges pointing to nodes $> N-1$. That loop has complexity $O(\log N)$. Hence, the time complexity of the parallelized algorithm is $O((N/s)+s+\log N)$, where s is the number of parallel threads. Note that the possible parallelization is much simpler than that for recursive algorithms.

Remark 2 If the user does not need the array of parents p, then the array can be excluded from Algorithm 2 as well as from Algorithm 3 below, since those algorithms never read the values from p.

To make the tree complete while retaining the minimal height, we can use the rotations algorithm from [8] as follows.

In the algorithm below, $R(j)$ stands for the height of the right subtree of a node j in the BST built by Algorithm 2 when the node j is reachable from the root node by descending via down-right edges only, h is the level of the current node, and x is the current node.

Algorithm 3.

```
function R(j):=
    if (j = N − 1) then 0
    else M(N − 1 − j) + 1

x:=t
h:=L(t)
if (R(x) < h and h > 1) then
    y:=l[x]
    t:=y
    p[y]:=NULL
    l[x]:=r[y]
    p[l[x]]:=x
    r[y]:=x
    p[x]:=y
    z:=y
else
    z:=x
    x:=r[x]
end if
h:=h − 1

while (h > 1)
    if (R(x) < h) then
        y:=l[x]
        p[y]:=z
        r[z]:=y
        l[x]:=r[y]
        p[l[x]]:=x
        r[y]:=x
        p[x]:=y
        z:=y
    else
        z:=x
        x:=r[x]
    end if
    h:=h − 1
end while
```

Algorithm 3 needs $O(\log N)$ time since h decreases by 1 in each iteration of the `while` loop.

Performance of the new algorithm

In order to test performance of the new algorithm, we compared it with the classical Wirth's algorithm [9] which recursively builds left and right subtree for each node, and with Vaucher's algorithm [8]. Rotations balancing the BST were included into the implementation of Vaucher's algorithm, as well as into the implementation of the new algorithm. Recall that all the three algorithms have linear time complexity and build complete BSTs. All the algorithms were implemented in R language and used array-based representations of BSTs. We ran the three tests: for all $N = 10{,}001, \ldots, 20{,}000$ randomly ordered; for 1000 values of N randomly selected without replacement from the range 100,001, … , 200,000; and for 1000 values of N randomly selected without replacement from the range 1,000,001, … , 2,000,000. All the three algorithms were run for the same choices of values of N. No parallelization was implemented for the algorithms. The tests were run on an AMD 3950X processor. It is to be noted that, in the tests, the time was measured for constructing a BST for the numbers $0, \ldots, N-1$, while, in practice, the tree may be constructed for an unsorted array of numbers, in which case sorting of the array must be done before applying each of the algorithms. The results of the tests are reported in Table 1 as mean time ± standard deviation.

We see that in all the tests the new algorithm performed at least 8 times faster than each of the other two considered algorithms. That can be explained by the fact that the new algorithm does not use recursive function calls which are time-expensive. Besides, a vectorized version of the new algorithm was tested (We might note that the vectorized version has $O(N)$ auxiliary memory complexity unlike the version with loops that needs $O(1)$ auxiliary memory).

Conclusion

We have suggested a new fast algorithm for constructing complete binary search trees. The new algorithm has linear time complexity like previously known algorithms, but, in contrast to those algorithms, the new algorithm does not use recursive function calls. The new algorithm allows simple vectorization and parallelization. When comparing the performance of the new algorithm with that of Wirth's

Table 1 Execution times of the algorithms, presented as mean ± standard deviation

	Mean execution time, milliseconds		
$N \in$	10,001, … , 20,000	100,001, … , 200,000	1,000,001, … , 2,000,000
Vaucher's algorithm	22.47 ± 5.72	228.5 ± 47.2	2343 ± 458
Wirth's algorithm	17.14 ± 4.94	172.6 ± 35.6	1769 ± 355
Algorithm 2 + Algorithm 3	0.852 ± 1.835	14.15 ± 8.42	217.8 ± 70.8

algorithm and Vaucher's algorithm, the new algorithm appeared to be at least eight times faster than its competitors in the considered tests.

Funding The study was supported by the program for fundamental scientific research of the Siberian Branch of the Russian Academy of Sciences, project FWNF-2022-0009.

Declarations

Conflict of interest The author declares that has no conflict of interest.

References

1. Aghaieabiane N, Koppelaar H, Nasehpour P. An improved algorithm to reconstruct a binary tree from its inorder and postorder traversals. J Algorithms Comput. 2017;49(1):93–113.
2. Das VV. A new non-recursive algorithm for reconstructing a binary tree from its traversals. In: 2010 International Conference on Advances in Recent Technologies in Communication and Computing, Kottayam, 2010; pp 261–263. https://doi.org/10.1109/ARTCom.2010.88.
3. Find first set. Wikipedia article. https://en.wikipedia.org/wiki/Find_first_set. Retrieved 1 Jun 2022.
4. Gagie T. New ways to construct binary search trees. In: Ibaraki T, Katoh N, Ono H (eds) Algorithms and computation. ISAAC 2003. Lecture Notes in Computer Science, vol. 2906, Springer, Berlin, Heidelberg, 2003.
5. Knuth DE. The art of computer programming: sorting and searching, vol. 3. Reading: Addison-Wesley Pub. Co; 1973.
6. Other built-in functions provided by GCC. https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html. Retrieved 27 Aug 2021.
7. Stephenson CJ. A method for constructing binary search trees by making insertions at the root. Int J Comput Inf Sci. 1980;9:15–29.
8. Vaucher JG. Building optimal binary search trees from sorted values in O(N) time. In: Essays in Memory of Ole-Johan Dahl, 2004; pp 376–388.
9. Wirth N. Algorithms + data structures = programs. Englewood Cliffs: Prentice-Hall; 1976.