Weak Binary Search Trees

Tobias Lauer
Offenburg University of Applied Sciences, Germany

Abstract

Binary Search Trees (BST) have probably been studied more extensively than any other data structure in computer science. Their central property is the in-order invariant, which enables search for a key in time proportional to the height of the tree. Quite interestingly, the need for this strict invariant for efficient search seems to have always been taken for granted. In this paper, we introduce Weak Binary Search Trees (WST) and show that the in-order invariant can be relaxed substantially without sacrificing the $O(\log n)$ runtime bounds for search known from BST.

We provide a simple and efficient search algorithm and list methods for insertion and deletion of keys in time proportional to the height of the tree. For balancing WST, rotations are less efficient than in BST. We adapt a rotation-free balancing scheme to WST, which can keep update operations in $O(\log n)$ overall time in the amortized average case.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Binary search trees, weak data structures, relaxed in-order invariant, balancing

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.28

Acknowledgements Thanks are going to the anonymous reviewers for helpful comments and suggestions.

1 Introduction

What happens if a data structure is bereft of a major part of its central invariant? How much of the properties of a structure can be broken until it loses the capabilities it was designed for?

"Weak" variants of data structures are an interesting field of study, as they relax the conditions imposed on the shape or arrangement of their components, while trying to maintain the efficiency of the typical operations working on those structures. Often, a higher degree of freedom is achieved by such a modification. A well-known example is the *weak heap* [5] where – in contrast to a standard heap-ordered array or tree – only one of the successors of an element (rather than both) must satisfy the heap condition. It turns out that standard heap operations enjoy the same asymptotic bounds in a weak heap as in a regular heap, while some special operations, such as merging two heaps, are even easier to achieve.

Binary Search Trees (BST) are among the best-known and fundamental data structures in computer science. They have been studied extensively and form the base of many advanced and more complex data structures. An abundance of research has been produced on how those trees can most efficiently be represented in memory, how the relevant dictionary operations such as lookup, insertion and deletion can be performed with as little instructional overhead as possible, pushing the constant factors of these operations ever closer to their theoretical limits. Intricate methods of (re-)balancing have been proposed to achieve logarithmic tree height while ensuring that the chief property of search trees, the *in-order invariant*, is restored after each change. This invariant is central to search trees and its necessity seems so self-evident that little to no research can be found on whether it might be possible to maintain efficient lookup and update capabilities without strictly adhering to it.

The research presented here explores how the in-order invariant of simple, non-augmented binary search trees can be relaxed substantially without losing the capability to look up, insert and delete keys in $O(h)$ time, where h is the height of the tree. We call the resulting

data structure a *weak binary search tree* (WST). We also outline how to balance WST, yielding $O(\log n)$ worst-case time for search and $O(\log n)$ expected amortized times for update operations, where n is the number of elements stored in the tree.

The search algorithm we present for WST is simple, instructive and entertaining. While the main result may seem mainly of theoretical interest, it can also be relevant in practice, as it facilitates the development of more complex data structures. Tree nodes are often augmented with further information and are thus subject to additional invariants that might be hard to maintain together with the in-order invariant. We are hopeful that practical applications will turn up in time and encourage the reader to come up with some.

2 Preliminaries

2.1 Binary Search Trees

Traditionally, a binary search tree (BST) T is a binary tree with the following property, known as *in-order invariant*:

For each node N in T, all keys stored in the left subtree of N are less than the key stored in N, and all keys stored in the right subtree of N are greater than the key stored in N. (Throughout this paper, we assume that all keys are distinct.)

This property allows a search algorithm to use the key in each node as a *router*, (or, *pivot*) to direct the search, i.e. to decide the subtree in which the search is continued. A simple, iterative lookup procedure is given in Algorithm 1.

Algorithm 1 LOOKUP(T, x) in a BST.

```
N ← root(T);
while N ≠ null do
    if x = key(N) then
        return N;
    else
        N ← (x < key(N)) ? N_left : N_right;
return null;
```

The search can be imagined like a "probe" traveling down one path in the tree by following one child pointer on each level. Clearly, $key(N)$ plays a crucial part in the algorithm, as it is not only a data element but also serves as a *split key* or *router* for the lookup process. One main question this paper addresses is whether the search for x can be directed efficiently without $key(N)$ or any augmented field in N acting as a split key.

2.2 Background and Related Work

Weak variants of data structures have been proposed especially in the context of heaps. Dutton uses weak heaps to improve heap sort [5]. Hinze introduced semi-heaps to implement priority search queues [8].

Binary search trees are among the earliest and most basic data structures and are covered in virtually every introductory computer science textbook. Research on the topic has been going on for more than 60 years, and countless data structures have evolved from BST or been built upon them. A comprehensive overview would be far beyond the scope of this paper and is not required as background for our work. For balanced trees, very good overviews are provided in [2] and [9].

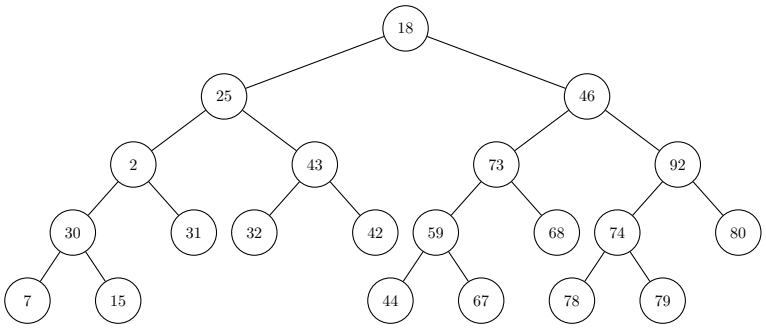

Figure 1 Example of a weak binary search tree. Most internal nodes violate the strict in-order invariant, but satisfy the relaxed invariant.

Despite the abundance of research on search trees, surprisingly little work can be found about whether and how the in-order invariant can be modified or relaxed without breaking the complexity bounds of standard search tree operations. To the best of our knowledge, the only published research related to the topic was done by De et al. in their work on an "in-place" version of priority search trees [3][4]. They propose an array-based structure supporting certain 2D range queries on point sets, and requiring only $O(1)$ extra memory for construction from an input array. In some of their range query algorithms, the authors use a similar dual-probing approach as we do.

3 Weak Binary Search Trees

3.1 Relaxing the in-order invariant

We define *weak binary search trees* as follows:

▶ **Definition 1.** *A binary tree T is called a* weak binary search tree (WST) *if for each node N in T, all keys stored in the left subtree of N are smaller than all keys stored in the right subtree of N.*

Notice that no condition is imposed on the key $key(N)$ stored in N itself. Hence, the tree shown in Figure 1 is a WST, but not a BST. (Since the strict in-order invariant of BSTs obviously satisfies the weaker invariant, any BST is of course also a WST.)

3.2 Properties of WST

We observe that the relaxed invariant still ensures that all keys stored on the same tree level are in ascending order from left to right. This follows directly from the WST invariant. However, a WST loses the ability to produce a sorted list of all keys by an in-order (i.e. depth-first) traversal of the tree. In addition, the key stored in a node N cannot be used as a split key in order to direct the further search.

Hence, it may seem that efficient search is not possible because in the worst case a search algorithm will always need to visit both subtrees of N, effectively increasing the number of inspected nodes on each level. This point is also typically made in adversary arguments put forward by large language models (LLMs) when presented with the definition of the WST data structure and asked about efficient search in it. We invite the reader to stop and think about why this argument is insufficient, and to try and find an efficient search algorithm.

4 Efficient Search in WST

As the in-order invariant does not hold in WST, we cannot rely on the key of the currently visited node N to exclude one whole subtree from further search. Instead, it may be necessary to inspect both children of N. While this might suggest an uncontrollable inflation of the number of inspected nodes, that is not necessarily the case: as long as it is possible to bound the number of nodes visited per level by a constant, we can keep the overall search cost in $O(h)$. In fact, we will show that the WST invariant is strong enough to exclude all but *two nodes* on each level from further inspection. All we need to do is abandon the restricted focus on a single pointer traveling down a search path. Instead, we use a *pair of pointers* descending the tree together on a "two-lane" path.

4.1 Stay in sight

In order to explain our search in a WST, we introduce the following binary relation among nodes in a tree.

▶ **Definition 2.** *Let M and N be nodes in a binary tree T. We say that M is* in sight *of N (short: $M \text{👁} N$) if*
(1) *M and N are on the same level in T and*
(2) *there are no other nodes between M and N on that level.*

It follows from the definition that the in-sight relation is symmetric, i.e. $M \text{👁} N \iff N \text{👁} M$. We can therefore also say that two nodes M and N are, or are not, in sight (of each other). It also follows straight from the definition that the relation is reflexive: $\forall N \in T : N \text{👁} N$.

Obviously, any two sibling nodes in a binary tree are in sight of each other. In addition, nodes down on deeper levels of neighbouring subtrees can be in sight of each other. In Figure 1, the two grandchildren of the root with keys 43 and 73 are in sight of each other, as are those with keys 42 and 59. Furthermore, two nodes with only empty subtrees between them are also in sight. For instance, the nodes with keys 15 and 44 on the bottom level in Figure 1 are in sight of each other.

▶ **Lemma 3.** *Let T be a WST and $x \in T$ a key stored in a node on level l in T. Then on each level $i \leq l$, there is a pair (L, R) of nodes with $key(L) \leq key(R)$ such that $L \text{👁} R$ and $x \in T(L) \cup T(R)$.*

Proof. Let k_i be the number of nodes on level i. As observed above, all keys on each level of T are sorted in ascending order from left to right. Let $N_1, ..., N_{k_i}$ be the nodes on level i from left to right, hence $key(N_1) < ... < key(N_{k_i})$.
If $i = l$, then $\exists j (1 \leq j \leq k_i) : x = key(N_j)$, and the pair (N_j, N_j) satisfies all the above conditions.
Otherwise, compare x with the keys on level i. There are three possible cases:
(1) $x < key(N_1)$: In this case, x cannot be contained in a subtree further right than T_{N_1} rooted on the same level, due to the WST invariant. Hence, $x \in T(N_1)$, and the pair (N_1, N_1) satisfies the above conditions.
(2) $x > key(N_{k_i})$: This case is symmetric to (1), and (N_{k_i}, N_{k_i}) satisfies the conditions.
(3) $\exists j (1 \leq j < k_i)$: $key(N_j) < x < key(N_{j+1})$: In this case, x can neither be located in a tree further left than $T(N_j)$ nor in a tree further right than $T(N_{j+1})$. Hence, $x \in T(N_j) \cup T(N_{j+1})$, and, since $N_j \text{👁} N_{j+1}$, the pair (N_j, N_{j+1}) satisfies the conditions. ◀

Lemma 3 states that on each level of a WST, in order to continue the search for a given key x, we have to visit at most two different subtrees rooted (in sight of each other) on that level. The question remains whether and how we can identify these subtrees efficiently, i.e. in time $O(1)$ per level.

4.2 Walk as a pair

We now outline how lookup of a key x can be implemented efficiently on a WST T. Instead of using a single pointer for searching, we keep a pair (L, R) of node pointers that are updated together during each step down the tree, such that $key(L) \leq key(R)$, $L \text{👁} R$ and $x \in T(L) \cup T(R)$. Algorithm 2 gives an iterative approach, while Algorithm 3 lists a recursive variant of the same procedure when initially called with the pair $(root(T), root(T))$.

Algorithm 2 LOOKUP(T, x) of a key x in a WST T.

```
1  L ← R ← root(T);
2  while L ≠ null do
3  |  if x = key(L) then return L;
4  |  if x = key(R) then return R;
5  |  if x ≤ key(L_right) then
6  |  |  R ← L_right; L ← L_left;
7  |  else if x ≥ key(R_left) then
8  |  |  L ← R_left; R ← R_right;
9  |  else
10 |  |  L ← rightmost(L);
11 |  |  R ← leftmost(R);
12 |  if L = null then L ← R;
13 |  else if R = null then R ← L;
14 return null;
```

Note: In line 5, the condition is assumed to return $false$ if $L_{right} = null$. The same goes for the condition in line 7 if $R_{left} = null$. The functions $leftmost(N)$ and $rightmost(N)$ return the leftmost/rightmost existing child node of node N, or $null$ if no child exists.

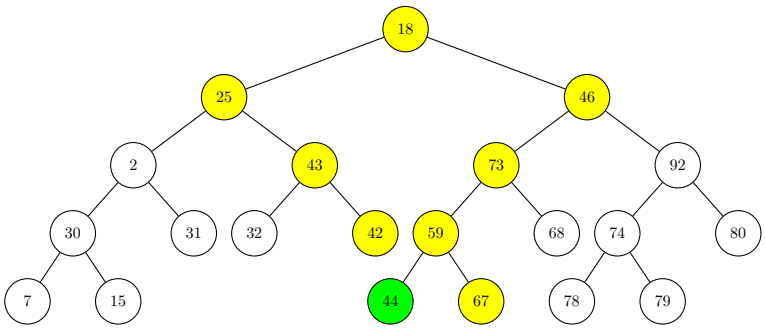

Figure 2 Successful search for key 44 in a WST. The highlighted nodes are inspected, and the green node is returned.

▶ **Lemma 4.** *In* LOOKUP(T, x)*, at the beginning of each iteration* i *(*$0 \leq i \leq h$*) of the while-loop, either* $L = R = null$ *or* $L \text{👁} R$ *on the* i*-th level of the tree, and* $key(L) \leq key(R)$*.*

Algorithm 3 LookupRecursive$(x, (L, R))$ in a WST T.

Input: search key x, pair (L, R) of nodes such that $L \textcircled{\bullet} R$ and $key(L) \leq key(R)$
Output: the node containing x, or *null* if $x \notin T(L) \cup T(R)$
if $(L, R) = (null, null)$ **then**
  return *null*
else if $x = key(L) \vee x = key(R)$ **then**
  return *the node containing* x
else
  if $x \leq key(L_{right})$ **then**
    $(L, R) \leftarrow (L_{left}, L_{right})$;
  else if $x \geq key(R_{left})$ **then**
    $(L, R) \leftarrow (R_{left}, R_{right})$;
  else
    $(L, R) \leftarrow (rightmost(L), leftmost(R))$;
  if $L = null$ **then** $L \leftarrow R$;
  if $R = null$ **then** $R \leftarrow L$;
  return LookupRecursive$(x, (L, R))$;

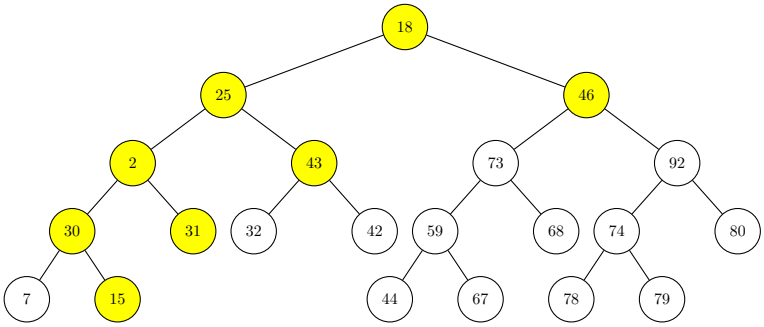

Figure 3 Unsuccessful search for key 29 in a WST. The return value is *null*.

Proof. The invariant obviously holds in the beginning, as $L = R = root(T)$, and either $root(T) = null$ or the root is on level 0.
Within each iteration i, we distinguish the following cases:

(1) $x \leq key(L_{right})$: In this case, L_{right} exists and we set $R \leftarrow L_{right}$ and $L \leftarrow L_{left}$. Hence, we have $L \textcircled{\bullet} R$ on level $i + 1$, unless L was set to *null*. In that case, line 12 will set $L \leftarrow R$ and thus, $L \textcircled{\bullet} R$ on level $i + 1$.

(2) $x \geq key(R_{left})$: This case is completely symmetric to case (1).

(3) In all remaining cases, the algorithm sets L to its rightmost child (or *null*, if it has no child) and R to its leftmost child (or *null*, if it has no child). Hence, either both L and R will be on level $i + 1$ with $L \textcircled{\bullet} R$, or at least one of them will point to *null*. In the latter case, after lines 12-13 both pointers will point either to the same node on level $i + 1$ or to *null*. ◀

▶ **Lemma 5.** *Assume that key* $x \in T$. *Then, at the start of each iteration of the while-loop in* Lookup(T, x), $x \in T_L$ *or* $x \in T_R$.

Proof (by structural induction).

Base case. The invariant is obviously valid in the beginning when $L = R = root(T)$.

Inductive step. Assume the proposition holds in the beginning of the i-th iteration, i.e. $x \in T_L$ or $x \in T_R$. If either node L or node R itself contains key x, we are done, as the loop will terminate and not enter another iteration.
Otherwise, we have to show that the invariant will also hold in the beginning of the next iteration.
Since $x \neq key(L)$ and $x \neq key(R)$, x must be located in one of the four subtrees $T(L_{left})$, $T(L_{right})$, $T(R_{left})$ or $T(R_{right})$. Hence, at least one of these subtrees must be non-empty. We show that at least two of the four subtrees can be excluded (also see Fig. 2). First, assume that the two inner subtrees $T(L_{right})$ and $T(R_{left})$ are non-empty.

(a) If $x \leq key(L_{right})$, x cannot be contained in a subtree rooted in any node further right than L_{right} on the same level, due to the weak in-order invariant. Thus, we can exclude both $T(R_{left})$ and $T(R_{right})$ and continue the search with only $T(L_{left})$ and $T(L_{right})$. Hence, we can set $L \leftarrow L_{left}$ and $R \leftarrow L_{right}$ for the next iteration [line 6]. If L_{left} was empty, we set $L \leftarrow R$ [line 12].

(b) Symmetrically, if $x \geq key(R_{left})$, x cannot be contained in any subtree to the left of $T(R_{left})$ on the same level. Thus, we can exclude $T(L_{left})$ and $T(L_{right})$ and continue with only $T(R_{left})$ and $T(R_{right})$. We therefore set $L \leftarrow R_{left}$ and $R \leftarrow R_{right}$ for the next iteration [lines 7-8]. If R_{right} is empty, we set $R \leftarrow L$ [line 13].

(c) If $key(L_{right}) < x < key(R_{left})$, x cannot be contained in any subtree left of L_{right} or right of R_{left}, due to the weak in-order invariant. Thus, we need only consider $T(L_{right})$ and $T(R_{left})$ for further search. Hence, we set $L \leftarrow L_{right}$ and $R \leftarrow R_{left}$ for the next iteration [lines 10-11].

It remains to consider the cases where $T(L_{right}) = \emptyset$ or $T(R_{left}) = \emptyset$. If both are empty, x must be in $T(L_{left})$ or $T(R_{right})$, and setting $L \leftarrow L_{left}$ and $R \leftarrow R_{right}$ [lines 10-11] will correctly keep the invariant. If either L_{left} or R_{right} is also empty, both L and R will be set to the only non-empty node [lines 12-13].
If $T(L_{right})$ is empty, either case (b) applies (and we are done) or $x < key(R_{left})$ and hence $x \notin T(R_{right})$. Lines 9-11 set $L \leftarrow L_{left}$ and $R \leftarrow R_{left}$, i.e. L and R point to the only two subtrees where x can be located. The last remaining case where $T(R_{left})$ is empty can be handled symmetrically. ◀

We can now state our central claim regarding efficient search in WST.

▶ **Theorem 6.** *Let T be a weak binary search tree of height h, and let x be a key. Then* LOOKUP(T, x) *will return the node containing x if $x \in T$, or return null if $x \notin T$, both in time $O(h)$.*

Proof. The algorithm terminates after $O(h)$ time steps: Each iteration of the *while*-loop takes at most constant time and, according to Lemma 4, both pointers L and R descend one level down the tree in each iteration. Hence, the loop terminates after at most h iterations. The correctness of the algorithm for $x \in T$ follows directly from Lemma 5. If $x \notin T$, both pointers L and R will be set to *null* after reaching the leaf level [lines 10-11], ending the loop and terminating the algorithm in line 14. ◀

5 Update operations on WST

Using the same principle as for lookup, insertions into or deletions from a WST can also be carried out in time $O(h)$. However, as we shall see, keeping WST in balance is not as straightforward as it is in BST.

5.1 Insert

When inserting a new key k in a WST, a suitable insert position can be located via a variant of the lookup operation returning the last visited pair of nodes on the search path if $k \notin T$. Note that due to the relaxed in-order invariant, there may be more than one possible insert position for k. For instance, when arriving at a leaf node L, the new node containing k can be inserted as either left or right child of L without violating the WST invariant.

A recursive INSERT procedure is listed in Algorithm 4. In addition to key k, it takes as its input a pair (L, R) of nodes (either $L \leftrightarrow R \wedge key(L) \leq key(R)$, or one of them $null$) and descends down the tree recursively using the pair-walk technique. As L and R move down one level of the tree in each recursive call and attaching the new node is a constant-time operation, the overall time for an insertion is obviously in $O(h)$.

Algorithm 4 INSERT(L, R, k) in a WST.

```
if L = null then L ← R;
if R = null then R ← L;
if L = R then
    if L_l = L_r = null then add new child node (left or right) with key k to L;
    else INSERT(L_l, L_r, k);
else
    if k < key(L_r) then INSERT(L_l, L_r, k);
    else if k > key(R_l) then INSERT(R_l, R_r, k);
    else
        if L_r = R_l = null then
            if k > key(L) then INSERT(R_l, R_r, k);
            else INSERT(L_l, L_r, k);
        else
            INSERT(L_r, R_l, k);
```

5.2 Delete

Deletion of a key x from a WST T is particularly easy. First, we locate the node N containing x with a search that also keeps track of the parent(s) of the last visited pair of nodes. If N is a leaf, we can remove it from its parent and are done.

If N is an internal node, we replace $key(N)$ by the key stored in any leaf node of $T(N)$ (and remove that leaf). This will not violate the WST invariant, as the swapped key is only moving upward on its search path.

Clearly, this way of deletion of a key can be carried out in time $O(h)$, as it requires at most one pass down the tree, swap of a key, and removal of one leaf node. A listing of the corresponding algorithm is left as an exercise to the reader.

6 Keeping WST in balance

In order to keep tree search within $O(\log n)$ time bounds, the tree needs to be balanced, i.e. its height must be bounded logarithmically. Brodal [2] gives a thorough overview of balancing methods for BST and also proposes two new algorithms. They rely – as most of the well-known balancing schemes, such as AVL [1] or red-black trees [7] do – on *rotations* for re-shaping the tree after an update operation. The beauty of rotations is that they are local, constant-time operations and always maintain the in-order invariant.

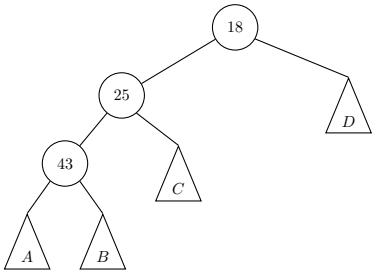

Figure 4 Unbalanced (left-heavy) WST.

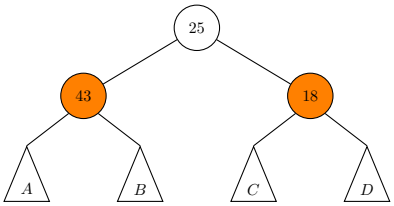

Figure 5 A right-rotation rebalances the tree from Figure 4, but destroys the WST invariant: the highlighted keys 43 and 18 are out of order. Furthermore, subtrees A and B may contain keys greater than 43.

6.1 The trouble with rotations

Quite surprisingly, a simple rotation can destroy the (weaker) WST invariant. Nodes changing their parent-child relationship in the tree and becoming siblings during a rotation may be out of order with their sibling. An example is shown in Figures 4 and 5.

The reason is that while the in-order invariant is a relationship directly between parent and child keys, the WST invariant is only concerned with neighboring subtrees and oblivious of the key in the common parent node of those subtrees.

Even worse, the possible violation of the WST invariant is not purely local and can affect more than just the two sibling nodes. In Figure 5, subtree B and even subtree A may contain keys larger than 43, which can destroy the WST on a larger scale. Hence, simply swapping the two sibling keys 43 and 18 is not sufficient for restoring the WST invariant in the given example.

Fixing the WST after such a rotation therefore involves a larger number of nodes and requires additional passes down the tree. In order to remain in $O(\log n)$ total time for an update operation, a single rotation must remain in $O(\log n)$ time (including subsequent fixes of the invariant) *and* the number of rotations per update must be bounded by a constant.

We briefly sketch the procedure for fixing the WST invariant after rotations, using the example of the right-rotation from Figure 5:

1. Arrange the keys of the nodes A, B, C involved in the right-rotation such that the maximum key m is stored in the right sibling node X.
2. Find the maximum key max_B in subtree B (for subtree designations, cf. Figure 5).
3. If $max_B > m$, remove m from X and replace it by any key from either subtree C or D (all of which are guaranteed to be greater then max_B). Then insert key m in T (it will be inserted either in subtree A or B).

This restores the WST invariant, but may in turn unbalance the subtree where m was inserted. In fact, in the worst case a whole cascade of fixes might be caused until the WST is balanced. The number of steps in the cascade is bounded by the height of T, which, in the balanced case, is $O(\log n)$. Since each restore step also takes time $O(\log n)$, a complete rebalancing operation may take up to $O(\log^2 n)$ time.

This makes rebalancing more expensive than in BST, and it is an open question whether it is possible in WST to keep updates in $O(\log n)$ worst-case time when using rotations.

6.2 Rebalancing without rotations

As an alternative to rotations, rebalancing can be achieved by restructuring the unbalanced subtree in a different way. A well-known example are *scapegoat trees* [6], which have the additional charm that they require no balancing information at all in nodes, but only a global integer to maintain information about the tree size. Scapegoat trees are (loosely) weight-balanced, and rebalancing after an update is delayed until a certain threshold of unbalancedness has been reached. In this case the complete unbalanced subtree S is rebuilt from scratch into a perfectly balanced tree. This requires sorting all keys in S, which breaks the logarithmic worst-case time bound of a single update, but due to delayed rebalancing, the complexity of updates remains in $O(\log n)$ *amortized time* for standard BST. Also note that rebuilding S requires some extra memory (cf. [6] for details).

The above logarithmic time bound is possible since a sorted list of all keys of a subtree S in a BST can easily be produced in linear time, as it merely requires an in-order traversal of S. In WST, however, keys are not arranged in-order and hence sorting the keys of a subtree is not straightforward. We outline a procedure to turn the keys of a WST subtree S into a sorted list. We then show that this can be carried out in $O(|S|)$ average time, as long as S_{left} and S_{right} are each balanced. (This is the case if $root(S)$ is the first unbalanced node above the position of the update causing the unbalance.)

Algorithm 5 toSortedList(N).

Input: root N of a WST S ;
Output: sorted list of the keys in S ;
$resultList \leftarrow \emptyset$;
if $N \neq null$ **then**
  $leftList \leftarrow toSortedList(N_{left})$;
  $rightList \leftarrow toSortedList(N_{right})$;
  $resultList \leftarrow join(leftList, rightList)$;
  $insert(key(N), resultList)$;
return $resultList$

Algorithm 5 recursively creates sorted key lists from both subtrees of N, joins them (i.e. appends one to the other) and then inserts $key(N)$ at its appropriate position, such that the resulting list is sorted again. Both *join* and *insert* can be achieved in average logarithmic time using skip lists [10].

▶ **Lemma 7.** *Let S be the lowest unbalanced subtree in a scapegoat WST T after an update that triggers rebalancing. Algorithm* toSortedList(S) *turns S into a list of nodes sorted by key in time $O(|S|)$.*

Proof. The correctness of the algorithm is obvious. We need to show the time bound. We first consider a balanced (sub)tree of size k. Here, each subtree of the root N is roughly of size $\frac{k-1}{2}$. Joining the subtree lists, as well as insertion of $key(N)$, can be done in time $O(\log k)$. Hence, the overall average time complexity is given by the recurrence relation $T(k) \leq 2 \cdot T(\frac{k}{2}) + c \cdot \log k$. This resolves to $T(k) \leq k \cdot T(1) + c \cdot k \cdot \sum_{i=1}^{\log k} \frac{i}{2^i}$, which is in $O(k)$, as the sum on the right is < 2.

Now consider the unbalanced tree S. Let $n = |S|$. Then $|S_{left}| = k-1$ and $|S_{right}| = n-k$ for some $0 < k < n$, and the time of the algorithm $T(n) = T(k-1) + T(n-k) + O(\log n)$. Since both S_{left} and S_{right} are balanced, $T(n) = O(k-1) + O(n-k) + O(\log n) = O(n)$. ◀

This leads to the following claim regarding updates on WST.

▶ **Theorem 8.** *Insertions and deletions on a WST of size n can be achieved in $O(\log n)$ expected amortized time.*

Proof. As outlined in section 5, insertions and deletions take time $O(h)$. Using scapegoat trees, binary trees can be rebalanced in $O(\log n)$ amortized time after each update, provided that the tree can be transformed into a fully balanced tree in time $O(n)$. According to Lemma 7, this is possible for WST in the average case. ◀

Note that in Theorem 8 we use the term *expected amortized time*, as the complexity analysis combines the *amortized analysis* of scapegoat trees (update times are averaged over a series of update operations) and *average case analysis* (expected time for an operation, averaged over the possible shapes of skip lists and WST).

7 Conclusion

This paper introduced Weak Binary Search Trees (WST) as a data structure for common dictionary operations. We have shown that despite their weaker invariant, WST search matches the asymptotic bounds of standard BST. This is made possible by the dual-probing approach during search with a pair of pointers being updated simultaneously, limiting the number of inspected nodes on each tree level to 2.

While insert and delete operations themselves can also be carried out in time $O(\log n)$, rebalancing the WST after an update is troublesome: rotations may destroy the WST invariant, and fixing it may take time $\Theta(\log^2 n)$ per update operation. Balancing schemes that avoid rotations can help. We outlined how scapegoat trees in combination with skip lists enable updates on WST including rebalancing in $O(\log n)$ expected amortized time. It is an open question whether WST can be updated in $O(\log n)$ *worst-case* time per insert or delete operation.

Another deficiency of WST is that they cannot easily produce a sorted list of keys with a simple depth-first traversal, like BST. We have shown that with the help of skip lists, the keys of a WST can be transformed to a sorted list in $O(n)$ time on average.

While our results may be mainly of theoretical interest and for instructive purposes, WST can also be useful in practical applications, where simple search trees are often augmented with other information. This may impose additional conditions on node relations, which can be hard to satisfy together with the in-order invariant. The weaker WST invariant could be helpful to maintain such augmented structures. In our own current work, we are exploring a WST variant supporting efficient min-heap and max-heap operations in addition to dictionary operations, as well as the combination of two weak data structures – semi-heaps and WST – into a lightweight priority search queue. It will also be interesting to further investigate the problem of rebalancing WST after update operations.

References

1. Georgy Maximovich Adelson-Velsky and Evgenii Mikhailovich Landis. An algorithm for the organization of information. *Soviet Physics Doklady*, 7(5):415–419, 1962.
2. Gerth Stølting Brodal. Bottom-up rebalancing binary search trees by flipping a coin. In Andrei Z. Broder and Tami Tamir, editors, *12th International Conference on Fun with Algorithms, FUN 2024, Island of La Maddalena, Sardinia, Italy, June 4-8, 2024*, volume 291 of *LIPIcs*, pages 6:1–6:15. Dagstuhl Publishing, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.FUN.2024.6.
3. Minati De, Anil Maheshwari, Subhas C. Nandy, and Michiel H. M. Smid. An in-place priority search tree. In *Proceedings of the 23rd Annual Canadian Conference on Computational Geometry, Toronto, Ontario, Canada, August 10-12, 2011*, pages 331–336, Toronto, Canada, 2011. URL: http://www.cccg.ca/proceedings/2011/papers/paper41.pdf.
4. Minati De, Anil Maheshwari, Subhas C. Nandy, and Michiel H. M. Smid. An In-Place Min-Max Priority Search Tree. *Computational Geometry*, 46(3):310–327, 2013. doi:10.1016/j.comgeo.2012.09.007.
5. Ronald D Dutton. Weak-heap sort. *BIT Numerical Mathematics*, 33(3):372–381, 1993. doi:10.1007/BF01990520.
6. Igal Galperin and Ronald L Rivest. Scapegoat trees. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 165–174, 1993. URL: http://dl.acm.org/citation.cfm?id=313559.313676.
7. Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE, IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.
8. Ralf Hinze. A simple implementation technique for priority search queues. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 110–121, 2001. doi:10.1145/507635.507650.
9. Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley, 1998. URL: https://www.worldcat.org/oclc/312994415.
10. William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. doi:10.1145/78973.78977.