Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 225 (2009) 99–113
www.elsevier.com/locate/entcs

Randomness Preserving Deletions on Special Binary Search Trees

Michaela Heyer[1,2]

Department of Computer Science
National University of Ireland
Cork
Ireland

Abstract

Deletions in binary search trees are difficult to analyse as they are not randomness preserving. We will present a new kind of tree which differs slightly from the standard binary search tree. It will be referred to as an ordered binary search tree as it stores a history element in its nodes, which provides information about the order in which the nodes were inserted. Using this extra information it is possible to design a new randomness preserving and order preserving deletion algorithm.

Keywords: Binary Search Trees, Randomness Preservation, Knott's Paradox

1 Introduction

Average-case analysis is the most challenging aspect of algorithm analysis and it is complicated further by the lack of randomness preservation as many have pointed out in the past ([5,6,11,1,10]). But what exactly do we mean by randomness preservation? In the context of binary search trees (BSTs) it is defined as follows: an operation is considered to be randomness preserving if, when applied to a random tree it will produce a random tree, i.e each possible tree structure is equally likely to arise. While BST insertions possess this special property, deletions in BSTs are not randomness preserving ([5,6,11,1]), a fact which was first discovered by Gary D. Knott ([4]). When an algorithm is randomness preserving the general consensus is that average-case time analysis is feasible. That this is indeed the case has been formally demonstrated in [10] where Schellekens introduces a new programming language MOQA for which all programs are guaranteed to be randomness preserv-

[1] Partially supported by the Centre for Efficiency Oriented Languages, Cork, Ireland
[2] Email: mh4@cs.ucc.ie

doi:10.1016/j.entcs.2008.12.069

ing and as a result give rise to recurrence equations for the average-case time in a (semi-) automated fashion.

Binary search trees and in particular the deletion on BSTs have been an active topic of research for a long time. One of the main researchers in this area was T.N. Hibbard and when he started his work in this field in 1962 he believed that BST deletions could be considered to be randomness preserving ([2]). It was only a few years later in 1975 that he was contradicted by Knott, who came up with what is now known as 'Knott's paradox' ([4]). Knott was the first to discover the pitfall with deletions in BSTs: they appear to be randomness preserving at first sight, as any number of deletions indeed preserve the property. However, once we have a number of deletions followed by a number of insertions, followed by more deletions the preservation of randomness is not necessarily guaranteed any longer (see [3,6]). Since this discovery the question of the existence of a truly randomness preserving deletion algorithm for BSTs has been posed by Knuth ([6]) and many have tried to find an answer. Seidel and Aragon in particular claim to be the first ones to have discovered such an algorithm, which when applied to their special tree, the *randomized search treap*, preserves randomness ([12]). Treaps are very similar to BSTs but they store a priority value in addition to the key value. The items are arranged in the tree with the keys adhering to the BST property and the priorities adhering to heap-order, i.e. the priority of any given node in the tree is smaller than or equal to that of its parent. For a randomized treap it is assumed that all priorities are independent, identically distributed random variables through which the randomness is introduced. The insertion algorithm is quite similar to the normal BST insertion algorithm but it requires a left/right rotation to ensure the heap-order priority of the nodes. Deleting an element is basically a backwards version of the insertion and both algorithms take $O(\log\, n)$ as do all update operations for this data structure. Even though treaps provide good performance, they do not actually answer Knuth's long standing question, as randomness is created on each insertion rather than truly preserved.

We will here present another variant of BST, the *ordered binary search tree* (OBST) and provide an insertion and deletion algorithm with guaranteed logarithmic performance. This new variant is quite similar to treaps, in that it stores two kinds of information at each node with one set being heap-ordered and the other set being ordered adhering to the BST property. However instead of using random variables as the set of priorities we simply time stamp each element on insertion and store this timestamp as the node's history value. This has the obvious disadvantage that the original input has to be randomized unlike treaps, where the randomness is introduced automatically as part of the insertion process. However randomizing the input is easily done and does not affect the overall performance. There is no need for rotations as part of the insertion algorithm in OBSTs, as the heap order property is automatically achieved by way of assigning the history values. The main advantages of OBSTs over treaps and other existing BST variants are a) all OBST operations, including the deletion, are truly randomness preserving and b) the deletion in OBSTs is also order preserving. By order preservation we mean

that deleting an element will create the same tree as would have been produced had the element deleted never been inserted. This property has several advantages: for example, we can predict the shape of a tree by examining the input sequence and the operations that are to be performed on it. As well as that we can easily determine an element's relative time of insertion from its position in the tree. This can be useful when analysing the order and kind of operations performed on the data structure. As mentioned above, OBSTs form the first data structure which provides a truly randomness preserving deletion algorithm, whereas randomness creating deletion algorithms have been described before ([7], [9]).

The remainder of this paper is organised as follows: in section 2 we describe this new kind of tree and provide an explanation of the deletion algorithm as well as its pseudo code. In section 3 we compare the probabilities of the different tree structures after a normal deletion with those using the new deletion algorithm on an example of a tree of size three. We then move on to provide a general proof of randomness preservation by proving that there is a bijection between the set of OBSTs and the set of permutations which respects the corresponding deletion and insertion operations. The average-case analysis of the algorithms can be found in section 4 where we show that both the insertion and the deletion algorithm have an expected performance of $O(\log n)$. Finally we provide a conclusion and some ideas for future work in section 5.

2 The Ordered Binary Search Tree

2.1 Introduction

We introduce a new notion of BST which stores information about the order in which the elements were inserted. From here on we will refer to these trees as *ordered binary search trees* (OBSTs). A similar notion is mentioned in [8] where Mishna describes the transformation between BSTs and heap ordered trees in order to create a bijection between permutations and BSTs. Any normal BST will create the same shape for the two distinct permutations $(y\,x\,z)$ and $(y\,z\,x)$. In an OBST we will store an item at each node consisting of the usual key and its position in the permutation thus yielding two distinct trees. This difference between BSTs and OBSTs is depicted in figure 1. Any OBST is therefore uniquely defined by the elements it contains and the order in which they were inserted unlike ordinary BSTs which are defined only by the elements they contain. When classifying the OBSTs, the underlying permutation will be taken into account, meaning that the two OBSTs depicted in figure 1 are classified as two different trees even if their shape is the same. In traditional BSTs it suffices to look at the shape alone when classifying the trees, as the deletion algorithm is based purely on this structural information. For our new kind of tree however, the deletion algorithm will use history information as well as structural information, which is the reason that this distinction is required.

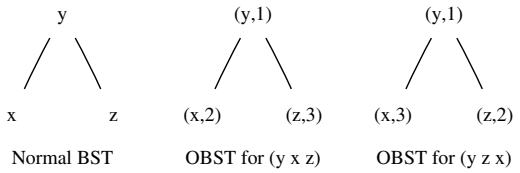


Fig. 1. The difference between OBSTs and ordinary BSTs

2.2 The Algorithm

To find/insert an element in an OBST, we proceed the same as with normal BSTs [11,5]. For simplicity we will assume that all keys are distinct.

The deletion operation uses information about the structure of the tree as well as information about the history of the tree which is in contrast to BSTs which rely on structural information only. Deleting an element from an OBST follows algorithm 1 and takes $O(\log n)$. Given a random tree as input, this operation will produce a random tree also, which is due to the fact that the tree is re-structured in such a way as to re-establish the heap order of the history values as well as the BST order of the key values. An item is deleted by replacing it with the smallest history value and then recursively re-inserting the remaining subtree at the correct historic level of the tree. The deletion process is shown in figure 2(a) to 4(a). As we are guaranteed that each new item has a different history value, there is no need to relabel the existing values after a deletion. This is a very important observation as relabeling would cause the algorithm to run in linear time, thus making it unacceptable in practice.

3 Randomness Preservation

In [3], Jonassen and Knuth show the deletion paradox on the example of BSTs of size 2 and 3 and we would ask the reader to familiarise himself with this example, as we will follow it very closely to show that our new BST variant solves the paradox. Knuth begins by defining the different shapes possible for BSTs of size 2 and 3. So let us consider the possible OBSTs of size 2 and 3 as depicted in figure 5. It can be easily seen that any list of integers with the same inter-relationship will yield one of the generic structures depicted and we can therefore restrict ourselves to values between 1 and n. There are two different trees of size 2, namely F and G and six different trees of size 3, which are S1, S2, S3, S4, S5, S6. OBSTs are defined by the elements they contain and the order of those elements hence S3 and S4 have to be considered 2 different trees. For normal BSTs we also have two trees of size 2, F and G but only five trees of size 3 which we will refer to as A, B, C, D and E and which are depicted in figure 6. The OBST trees of size 3 map to ordinary BSTs of size 3 as follows:

S1 → A
S2 → B
S3 → C

Algorithm 1 *The Deletion in OBST*

```
Delete( T, k )
x ← Find( T, k )
if ( x has no children )
  remove x
else if ( x has one child c )
  replace( x, c )
else
  if ( x.leftChild.getHistory < x.rightChild.getHistory )
    l ← x.leftChild
    if (l.rightChild.isExternal)
      replace( x, l )
    else
      s ← l.rightChild
      replace( x, l )
      ReInsert( l.rightChild, left, s )
  else
    //equivalently ( change right to left and v.v. )
```

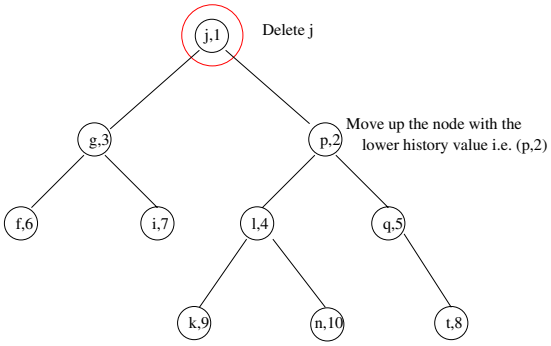
Algorithm 2 *ReInsert in OBST*

```
ReInsert( correctPos, direction, subtree )
testPos ← correctPos.parent
while ( correctPos.getHistory < subtree.getHistory && !testPos.isExternal )
  set Flag
  if ( direction = left )
    correctPos ← correctPos.leftChild
  else
    correctPost ← correctPos.rightChild
  testPos ← correctPos
if ( Flag is not set )
  newSubtree ← correctPos
  replace( correctPos, subtree )
  ReInsert( subtree, - direction, newSubtree )
else
  if ( !correctPos.isExternal )
    replace( correctPos, subtree )
    ReInsert( subtree, - direction, correctPos )
  else
    replace( correctPos, subtree )
```

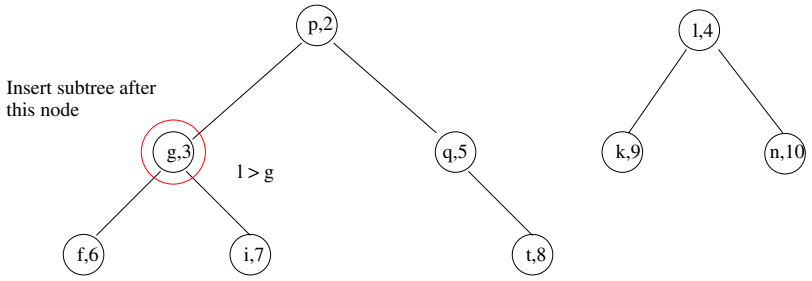
S4 → C
S5 → D
S6 → E

It is easy to see that we do not have a bijection, as there are two OBSTs which map to only one BST. However we do have a bijection between permutations and OBSTs which will help us in achieving randomness preservation.

We will now look at the deletion process on an OBST of size 3: there are three possible labels that can be deleted: x, y or z. If we use the deletion algorithm described in algorithm 1, then we get the following distribution of 2-element trees:

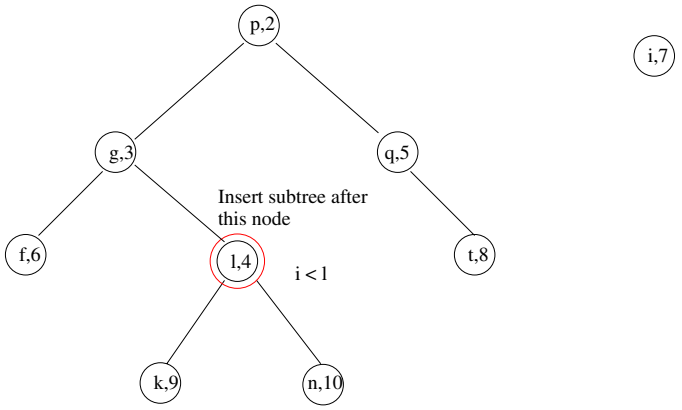


(a) OBST from permutation ($j\ p\ g\ l\ q\ f\ i\ t\ k\ n$)

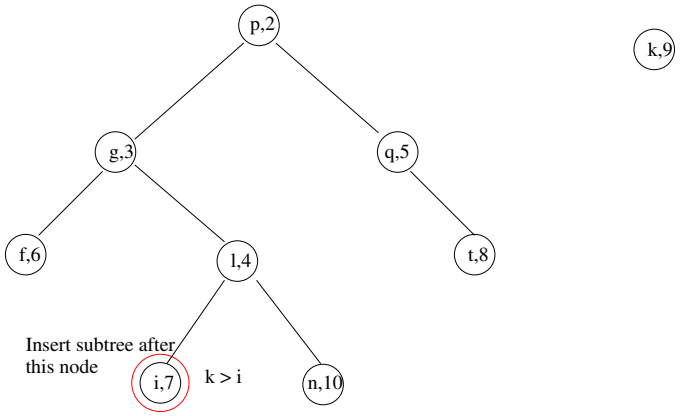


(b) The tree before ReInsert and its separate subtree

Fig. 2.



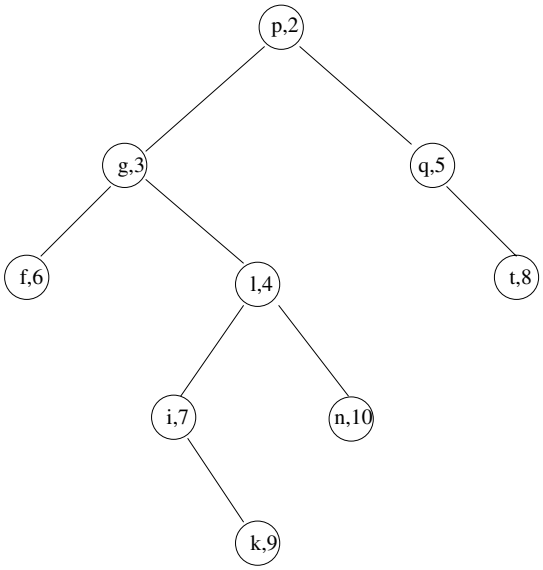
(a) The tree after the first call to ReInsert with its separate subtree



(b) The tree after the second call to ReInsert with its separate subtree

Fig. 3.

Initial Structure	Delete x	Delete y	Delete z
S1	F	F	F
S2	F	F	G
S3	G	G	F
S4	G	F	F
S5	F	G	G
S6	G	G	G



(a) Tree after deletion of j

Fig. 4.

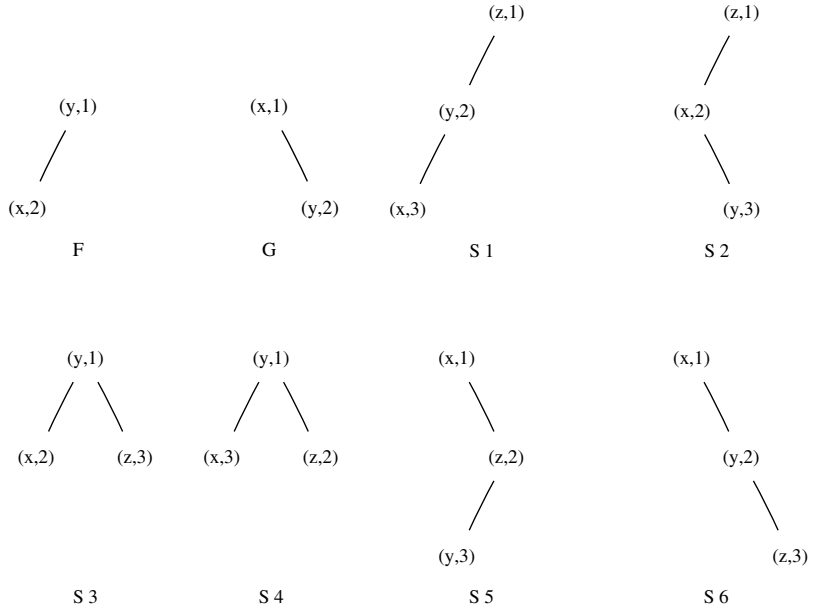


Fig. 5. OBST of size 2 and 3

The overall distribution of F and G is $\frac{9}{18} = \frac{1}{2}$, which is the same for deletion on normal BSTs and is as we would expect. If we distinguish between the different F and G structures according to the elements they contain however, then we get

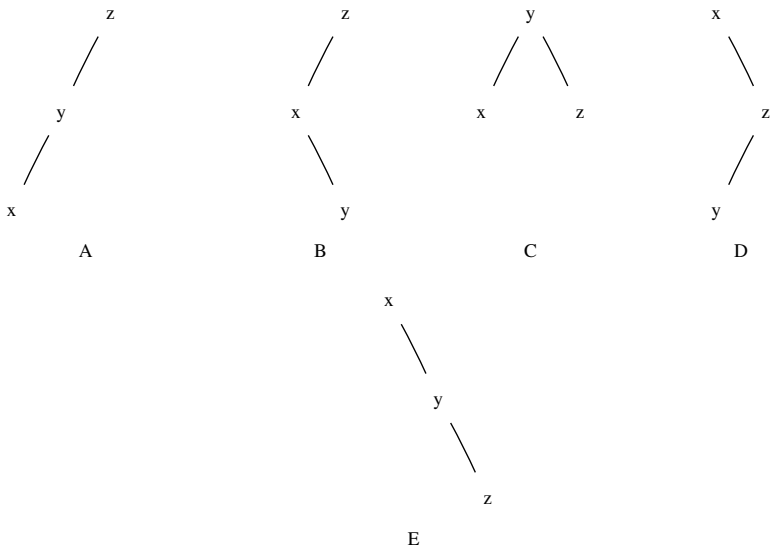


Fig. 6. BST of size 3

different values for deletion in OBSTs and deletion in normal BSTs:

	F(x,y)	F(x,z)	F(y,z)	G(x,y)	G(x,z)	G(y,z)
BST	3/18	4/18	2/18	3/18	2/18	4/18
OBST	3/18	3/18	3/18	3/18	3/18	3/18

Now assume a random insertion into these trees. The newly inserted element can have the following inter-relationship to the elements that were originally in the tree, before the first deletion took place: *a)* be smaller than all of them, *b)* be bigger than one of the items but smaller than the other two, *c)* be bigger than two of the items but smaller than the other one, *d)* be bigger than all of them. For example the structure F(x,z) yields the structure S1 in case *a)*, the structure S2 in cases *b)* and *c)* and the structure S3 in case *d)*. If we do the same for all 2-element structures and then compute the probabilities we get the following values for OBSTs:

P(S1) = $\frac{3+3+6}{72} = \frac{1}{6}$, P(S2) = $\frac{3+6+3}{72} = \frac{1}{6}$, P(S3) = $\frac{6+3+3}{72} = \frac{1}{6}$,
P(S4) = $\frac{3+3+6}{72} = \frac{1}{6}$, P(S5) = $\frac{3+6+3}{72} = \frac{1}{6}$, P(S6) = $\frac{6+3+3}{72} = \frac{1}{6}$.

In comparison, the values of the five possible shapes for normal BSTs are as follows:

P(A) = $\frac{11}{72}$, P(B) = $\frac{13}{72}$, P(C) = $\frac{25}{72}$, P(D) = $\frac{11}{72}$, P(E) = $\frac{12}{72}$.

If we now perform an additional random deletion and compute the probabilities we see that structures F and G still arise with a probability of $\frac{1}{2}$ each, in comparison to normal BSTs where this final deletion causes structure F to arise with probability $\frac{109}{216} > \frac{1}{2}$.

From the above results it seems that this new deletion is randomness preserving. However we have only looked at trees of a very small size and cannot infer from these examples that it will hold for all trees. To prove that it does, we will first prove the correctness of the operations described in section 2.2. In the following we will show

that if we are given an OBST which was created from a specific permutation, then inserting an element into this OBST is equivalent to adding a new element to the permutation and then using this new permutation to create the OBST. Similarly deleting an element from the OBST is equivalent to removing the element from the permutation and then creating the OBST from the modified permutation. In particular we will show the following three things:

a) there is a one-to-one mapping between permutations and OBSTs

b) a random insertion preserves this mapping

c) a random deletion also preserves this mapping.

We will use the following notation.

Let $P = (e_1\, e_2\, \ldots\, e_n)$ denote a permutation of the set X of keys e_1 to e_n and let $T = [e_1, e_2 \ldots e_n]$ denote the OBST which was created by inserting the keys e_1 to e_n in exactly that order.

Lemma 3.1

$$P = (e_1 e_2 \ldots e_n) \Leftrightarrow T = [e_1, e_2 \ldots e_n]$$

Proof. First we show

$$P = (e_1\, e_2\, \ldots\, e_n) \Rightarrow T = [e_1, e_2 \ldots e_n]$$

$T = [e_1, e_2 \ldots e_n]$ is defined above to be the OBST created by inserting the elements e_1 to e_n in that order, so the equation is true by definition.

Now let us assume that

$$P = (e_1 e_2 \ldots e_n) \Leftarrow T = [e_1, e_2 \ldots e_n]$$

does not hold, i.e. we have two different permutations for one OBST. This would mean that we have at least one key at a different position which would result in a node with a different key-history value pair, hence a different tree. So we have found a contradiction and thus we know that

$$P = (e_1 e_2 \ldots e_n) \Leftarrow T = [e_1, e_2 \ldots e_n]$$

□

Lemma 3.2

$$P = (e_1 e_2 \ldots e_n e_{n+1}) \Leftrightarrow \textit{Insert } e_{n+1} \textit{ into } T = [e_1, e_2 \ldots e_n]$$

Proof. To show $P = (e_1\, e_2\, \ldots\, e_n e_{n+1}) \Rightarrow \textit{Insert } e_{n+1} \textit{ into } T = [e_1, e_2 \ldots e_n]$ consider the permutation $P = (e_1\, e_2\, \ldots\, e_n e_{n+1})$. We have already shown in a) that this creates exactly one OBST which is: $T = [e_1, e_2 \ldots e_n, e_{n+1}]$. This in turn describes the OBST which contains the elements e_1 to e_{n+1} and as e_{n+1} is the element with the highest subscript, it is the node in T with the highest history value, i.e. the node that was inserted last. No changes have been made to the position or history values of any other nodes.

Now consider the initial situation *Insert* e_{n+1} *into* $T = [e_1, e_2 \dots e_n]$, i.e. the case $P = (e_1\, e_2\, \dots\, e_n e_{n+1}) \Leftarrow$ *Insert* e_{n+1} *into* $T = [e_1, e_2 \dots e_n]$. Can we show that the resulting tree could have only been created by $P = (e_1\, e_2\, \dots\, e_n e_{n+1})$? From a) we know that $T = [e_1, e_2 \dots e_n]$ could have only been created by $P = (e_1 e_2 \dots e_n)$ and we also know that our insertion algorithm will give e_{n+1} the highest history value and add it at its correct position in T without actually modifying the rest of the tree structure. Therefore the only permutation that this tree could have been created from is the permutation $(e_1 e_2 \dots e_n)$ plus e_{n+1} added to the end of it which is equal to $P = (e_1\, e_2\, \dots\, e_n e_{n+1})$.

□

Lemma 3.3

$$P = (e_1\, e_2\, \dots\, e_{j-1}\, e_{j+1}\, \dots\, e_n) \Leftrightarrow \textit{Delete } e_j \textit{ from } T = [e_1, e_2 \dots e_n]$$

Proof. First we will look at $P = (e_1\, e_2\, \dots\, e_{j-1}\, e_{j+1}\, \dots\, e_n) \Rightarrow$ *Delete* e_j *from* $T = [e_1, e_2 \dots e_n]$.

We already know from a) that the permutation $P = (e_1\, e_2\, \dots\, e_{j-1}\, e_{j+1}\, \dots\, e_n)$ creates exactly one OBST, which is $T = [e_1, e_2, \dots e_{j-1}, e_{j+1} \dots e_n]$. Is this the same tree as $[e_1, e_2 \dots e_n]$ with e_j deleted? For two OBSTs to be the same, all nodes have to have the same key value, same history value and same position in the tree, where the position is characterized by a) the level and b) the left/right relationships. No key or history values are being modified as part of the deletion, so we do not have to worry about that aspect. It is easy to see that the position in the tree is certainly the same for all nodes up to e_{j-1}, as the deletion only modifies the part of the tree which contains nodes with history values greater than that of the deleted node. Both trees do not contain e_j, so we still have a match. What about the nodes e_{j+1} to e_n? The deletion may move these nodes, but if it does it takes care of two things: moving them to the correct level according to its history value and following the proper key comparison while moving them down the tree. Thus the new position of those nodes is the same as if they had been inserted after e_{j-1}, which proves that the two trees are the same.

To prove $P = (e_1\, e_2\, \dots\, e_{j-1}\, e_{j+1}\, \dots\, e_n) \Leftarrow$ *Delete* e_j *from* $T = [e_1, e_2 \dots e_n]$ we need to show that the deletion algorithm does not violate any of the following two properties: the relationship between the element e_j and any of its predecessors e_i, where $1 \leq i < j$ and the original relative ordering of the remaining elements. When deleting an element from an OBST, the algorithm does not modify the relative ordering between the elements. The structure of T is modified but the subtrees are reattached to the tree in such a manner as to preserve the original left and right subtree relationships. Consider an arbitrary element e_x in the tree: after the deletion of e_j the position of e_x might be different. However if e_x was in the left (right) subtree of any e_j, where $1 \leq j < x$, then it will still be there afterwards. Therefore the new tree could have only been created from the permutation $P = (e_1\, e_2\, \dots\, e_{j-1}\, e_{j+1}\, \dots\, e_n)$. □

Lemmas 1-3 immediately imply the following Theorem:

Theorem 3.4 *There is a bijection between the set of OBSTs and the set of permutations (of the same size) which respects deletion and insertion.*

Proof. The Theorem is an immediate corollary from the Lemmas. □

Corollary 3.5 *Deletions and Insertions on Ordered binary search trees are randomness preserving.*

Proof. Permutations are randomness preserving ([11]) and we have proved that there exists a bijection between ordered binary search trees and permutations. □

4 Average-Case Analysis

As both the insertion and the deletion algorithm are randomness preserving it is very easy to compute the expected values for internal path-length and height. The internal path-length is equivalent to the construction cost of a random BST and is described by the following recurrence which can be found in [11]:

$$C_n = n - 1 + \frac{1}{n} \sum_{1 \le k \le n} (C_{k-1} + C_{n-k}) \text{ for } n > 0 \text{ with } C_0 = 0 \tag{1}$$

We can solve this recurrence using generating functions yielding the following formula for the expected internal path-length:

$$2(n+1)(H_{n+1} - 1) - 2n \tag{2}$$

where H_n denotes the *harmonic number* which is defined as $H_n = \sum_{k=1}^{n} \frac{1}{k}$. We also know that the sum of depths of all nodes in the tree yields the internal path-length:

$$\text{Internal path-length } = \sum_{i=1}^{n} d_i \tag{3}$$

Now we can easily calculate the expected depth of an item from (2) and (3):

$$\text{Expected depth: } D_e = 2H_{n+1} + 2\frac{H_{n+1} - 1}{n} - 4 \tag{4}$$

We will also require a value for the expected height $Height_e$ in a BST. An approximation for this value can be found in [11]:

$$\text{Expected height: } Height_e = c \log n, \text{ where c } \approx 4.31107 \tag{5}$$

These are the main values required to analyse both algorithms.

Insertion: To insert a new item into the tree, we essentially perform a search for the correct external node and it is not hard to see that the expected cost is directly related to the number of nodes that are visited along this search. In fact, the insertion of a new node is equal to the expected depth plus one to reach the external node.

$$I_e = D_e + 1 = 2H_{n+1} + 2\frac{H_{n+1} - 1}{n} - 3 \tag{6}$$

The approximation for the harmonic number H_n is given in [11] as: $H_n \approx \log n + .57721\cdots$. Using this approximate and the fact that $\frac{H_{n+1}}{n}$ goes to zero as n becomes

large, we can see that the average cost for insertion is $O(\log\ n)$ which is as we would expect for random BSTs.

Deletion: To analyse the deletion algorithm, we divide it into two main parts. In the first part we perform a search for the node to be deleted, let us refer to the cost for this as S_e. The second part of the algorithm is concerned with re-structuring the tree to make sure it is still an OBST. The cost for this part will be referred to as R_e. Finding the cost S_e involves arguments similar to the ones used for the insertion algorithm. Again, the cost is directly related to the number of nodes visited along our search path, only this time we are looking for an internal node rather than an external node. Therefore we can simply take the value for the expected depth of a node: $S_e = D_e$.

Finding the value for R_e is slightly more complex. The cost for re-structuring is essentially nothing but a recursive call to ReInsert, so the cost can be expressed by the following recurrence:

$$R_n = I_n + R_s, \text{with } 0 < s < n \tag{7}$$

The cost of ReInsert on a tree with n nodes is made up of the cost of inserting a tree of arbitrary size into a tree of size n plus the cost of calling ReInsert on a smaller tree T_{sub} with only s nodes. We stop when we reach an external node, i.e. when $s = 0$. Inserting a tree is not very different from inserting a single node: we consider only the root of the tree and try and find the correct position in the tree, insert the root and through its children pointers, the rest of the tree gets automatically inserted. As part of ReInsert we do not necessarily need to find an external node to insert the tree and so we use the cost of finding a node in a tree of size n:

$$I_n = D_e = 2H_{n+1} + 2\frac{H_{n+1} - 1}{n} - 4$$

Now we need to get a handle on the average size s of T_{sub}. We already know the expected depth of an element in the tree and we also know the expected height $Height_e$ of the tree. Therefore we can compute the expected height h of T_{sub} as follows:

$$h = Height_e - D_e \tag{8}$$

Filling the values from (5) and (4) into (8) we get:

$$h = c\ \log\ n - 2H_{n+1} - 2\frac{H_{n+1} - 1}{n} + 4$$

Now we can fill in the approximation for the harmonic number to give:

$$h = c\ \log\ n - 2\ \log\ (n+1) + 2c_1 - 2\frac{\log\ (n+1) + c_1 - 1}{n} + 4$$

At this point the equation looks like it is not going to be of any help, as it is far too complicated to plug into the recurrence. However, for this analysis we are not interested in constant factors, so we can ignore them. Also we are only interested in the upper bound, so using O() notation we get the following:

$$h = O(\log n) - O(\log n) + O(\frac{\log n}{n})$$

This simplifies to:

$$h = O(\frac{\log n}{n}) \tag{9}$$

We also know that for the average height of T_{sub} the normal formula applies, so:

$$h = c \log s \Leftrightarrow s = 2^{\frac{h}{c}} \tag{10}$$

We fill in the value we got for h in (9) into (10) yielding the following for the cardinality s of T_{sub}:

$$s = \sqrt[n]{n}$$

Now we can finally plug the values into (7):

$$R_n = O(\log n) + O(R_{\sqrt[n]{n}})$$

After the first recursive step the recurrence will look as follows:

$$O(\log n) + O(\log \sqrt[n]{n}) + R_{\sqrt[n]{n}-k}$$

It is easy to see that as this recurrence unfolds further, all additional terms are bounded by $O(\log n)$ and we can now use this value in our original equation for the average cost of deletion:

$$Del_e = S_e + R_e = O(\log n) + O(\log n) = O(\log n)$$

Thus we get a logarithmic average running time for the deletion algorithm.

5 Conclusion and Future Work

We have presented a new kind of BST, which provides an answer to Knuth's open question regarding the existence of a randomness preserving deletion algorithm for BSTs. The insertion and deletion algorithms are very straightforward, easy to implement and have an expected performance of $O(\log n)$. The deletion algorithm is the first ever to be randomness preserving. On top of that it is also order preserving which can be very useful in practice. Clearly OBSTs are very usable as they have good average-case performance and the only extra information needed is the history value in the form of an integer.

To further investigate the usability of these trees, one might be interested to show how the actual tree structure and the corresponding internal path-length change after multiple insertions and deletions. It would also be of interest to compare experimentally the (average-, best- and worst-case) performance of the OBST deletion with the deletion of other random BSTs to see which runs faster on average.

References

[1] S. Edelkamp. Weak-heapsort, ein schnelles sortierverfahren. Diplomarbeit, Universität Dortmund, 1996.

[2] T. N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM*, 9, 1962.

[3] A. T. Jonassen and D.Knuth. A trivial algorithm whose analysis isn't. *J.Comput. Syst. Sci.Journal of the ACM*, 16(3):301–322, 1978.

[4] G. D. Knott. *Deletion in Binary Storage Trees.* PhD thesis, Stanford University, 1975.

[5] D. Knuth. *Sorting and Searching - Volume 3 of The Art Of Computer Programming.* Reading, Massachusetts: Addison-Wesley, 1973.

[6] D. Knuth. Deletions that preserve randomness. *IEEE Trans. Software Engineering*, 3:351–359, 1977.

[7] C. Martinez and S.Roura. Randomized binary search trees. *Journal of the ACM*, 45(2), 1998.

[8] M. Mishna. Attribute grammars and automatic complexity analysis. *INRIA*, 2000.

[9] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[10] M. P. Schellekens. *A Modular Calculus for the Average Cost of Data Structuring, Efficiency-Oriented Programming in MOQA.* Springer, to appear.

[11] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 1996.

[12] R. Seidel and C. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.