# Chapter 2

# Records and Random Binary Search Trees

In this chapter we examine the use of *records* in analyzing the running times of operations on randomized data structures. The records we are talking about are not records in a database, but rather best results so far, like Olympic records. In particular, let $A_1, \ldots A_n$ be a random permutation of the integers $1, \ldots, n$. Then $A_k$ is a *record* if Line 4 of the following pseudocode executes when $i = k$.

```
1: $m \leftarrow -\infty$
2: for $i = 1, \ldots, n$ do
3:   if $A_i > m$ then
4:     $m \leftarrow A_i$
5:   end if
6: end for
```

More precisely, $A_i$ is a record if and only $A_i = \max\{A_1, \ldots, A_i\}$. As an example, the records in the sequence

$$S = \underline{2}, \underline{7}, 5, 4, \underline{9}, 8, 3, 1, 6$$

are underlined.

The probability that $A_i$ is a record is exactly $1/i$ since $A_1, \ldots, A_i$ has only one maximum and it is equally likely to occur at any of the $i$ positions. What is the expected number of records in a random permuation of $n$ elements? Define the *indicator variable*

$$I_i = \begin{cases} 1 & \text{if } A_i \text{ is a record} \\ 0 & \text{otherwise} \end{cases} .$$

Then the expected value of $I_i$ is $\mathbf{E}[I_i] = 0 \cdot \Pr\{A_i \text{ is not a record}\} + 1 \cdot \Pr\{A_i \text{ is a record}\} = 1/i$. So the expected number of records is given by

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=1}^{n} I_i\right] &= \sum_{i=1}^{n} \mathbf{E}[I_i] && (2.1)\\
&= \sum_{i=1}^{n} 1/i && (2.2)\\
&= H_n \ , && (2.3)
\end{aligned}$$

Where $H_n = \sum_{i=1}^{n} 1/i$ is called the $n$th *harmonic number*. The value of $H_n$ has received considerable attention. Bounding $H_n$ using integrals [4, Section 3.2] shows that $\ln n \le H_n \le \ln n + 1$ for all $n > 1$.

Thus, the expected number of records in a random sequence is $O(\lg n)$. This very simple fact can be used to give very short proofs about the running times of many randomized algorithms.

## 2.1 Random Binary Search Trees

A *binary search tree* T is either the symbol **nil**, or consists of an integer *key*, key(T), and two *children*, left(T) and right(T), that are both binary search trees. Additionally, a binary search tree satisfies the following *search property*

**Property 1** (Search Property)**.** *If* $\mathrm{left}(T) \neq \mathrm{nil}$ *then* $\mathrm{key}(\mathrm{left}(T)) \le \mathrm{key}(T)$ *and if* $\mathrm{right}(T) \neq \mathrm{nil}$ *then* $\mathrm{key}(T) \le \mathrm{key}(\mathrm{right}(T))$

For an integer k, the *search path* of k in T, denoted path(k, T) is given the following recursive search procedure

$$\mathrm{path}(k,T) = \begin{cases} \mathbf{nil} & \text{if } T = \mathbf{nil} \\ T & \text{if } k = \mathrm{key}(T) \\ T, \mathrm{path}(k, \mathrm{left}(T)) & \text{if } k < \mathrm{key}(T) \\ T, \mathrm{path}(k, \mathrm{right}(T)) & \text{if } k > \mathrm{key}(T) \end{cases} .$$

A *random binary search tree* T of size $n$ is a binary search tree constructed by inserting the elements $\{1, \ldots, n\}$ into T in random order. Assuming that the key $k$ is not already contained in the tree T, we insert $k$ into T by making a new tree $T'$ with $\mathrm{key}(T') = k$ and making $T'$ a child of the last non-**nil** tree in path(k, T). This results in a sequence of trees $T_0, \ldots, T_n = T$ where $T_i$ is the tree resulting from the insertion of the first $i$ elements. See Figure 2.1 for an example of a binary tree constructed by a sequence of insertions.

One question we might ask about a random binary search tree is how much it costs to construct one. Most of us will have seen bad examples where constructing a binary search tree on $n$ elements by repeated insertion takes $\Omega(n^2)$ time. One obvious example of this occurs when the elements form an increasing sequence, in which case the tree is just a sorted list.

Since we are assuming that the elements of T are inserted in random order, what we would really like to know is the expected cost of creating T. One way to tackle this is to find the expected cost of inserting the $i$th element and then summing this over all $i$. In other words, if $C_i$ is the cost of the $i$th

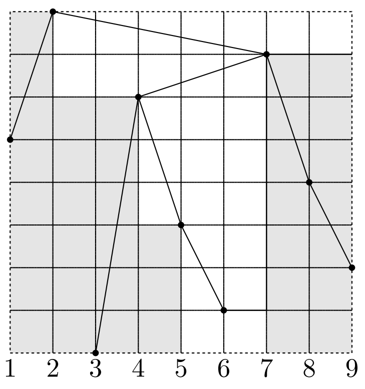

Figure 2.1: A binary search tree constructed by inserting the elements 2, 7, 4, 1, 8, 5, 9, 6, and 3 in that order.

insertion then

$$\mathbf{E}[\text{Cost of building } T] = \mathbf{E}\left[\sum_{i=1}^{n} C_i\right] = \sum_{i=1}^{n} \mathbf{E}[C_i] \enspace .$$

It is clear that $C_i$ is proportional to the length of $\mathrm{path}(k_i, T)$, where $k_i$ is the ith value inserted. To bring things to an even finer level of detail, let $\mathrm{path}(k_i, T) = U_1, \ldots, U_m$. We call $U_j$ a *left turn* if $U_{j+1} = \mathrm{left}(U_j)$, otherwise we call $U_j$ a *right turn*. Let $L_i$ and $R_i$ denote the number of left and right turns, respectively, in $\mathrm{path}(k_i, T)$. We will analyze the expected values of $L_i$ and $R_i$ seperately.

To get some intuition about $L_i$ and $R_i$ we should look at an example from Figure 2.1. In this example, $\mathrm{path}(6, T) = 2, 7, 4, 5, 6$. This search path takes a left turn at node 7 and right turns at nodes 2, 4 and 5. The insertion sequence at the time node 6 is inserted is

$$S = 2, 7, 4, 1, 8, 5, 9, 6 \enspace .$$

Consider the sequence containing only those elements that are less than 6, i.e.,

$$S' = \underline{2}, \underline{4}, 1, \underline{5} \enspace .$$

Note that the records in the sequence $S'$ are underlined and that these records correspond to the right turns in $\mathrm{path}(6, T)$. We might also consider the sequence of elements larger than 6, i.e.,

$$S'' = \underline{7}, 8, 9 \enspace .$$

If we modify the definition of records to use the minimum instead of the maximum, then the left turns in $\mathrm{path}(6, T)$ all correspond to records in the sequence $S''$.

Is this always the case? The answer to this question is yes. To see this, let $T_j$ denote the tree $T$ after insertion of the first $j$ elements. The $j$ keys inserted into $T_j$ divide the real line up into $j + 1$ intervals, and $k_i$ lies in one of these intervals $[a, b]$. When we insert the $(j + 1)$-st element, one of two things happens, either $\mathrm{path}(k_i, T_j) = \mathrm{path}(k_i, T_{j+1})$ or not, in which case the path of $k_i$ increases by one. The key observation is that the latter case only happens if the newly inserted key lies in the interval $[a, b]$. If the newly inserted key is less than $k_i$ then it is a record in $S'$. If it is greater than $k_i$ then it is a record in $S''$.

By splitting the insertion sequence into two sequences $S'$ and $S''$ we get two new sequences, which are both random. That is, the elements of $S'$ (and $S''$) are equally likely to occur in any order. Therefore, the expected number of records in $S'$ is

$$\mathbf{E}[R_i] = H_{|S'|} \le H_i \ ,$$

and a symmetric argument shows that $\mathbf{E}[L_i] \le H_i$. Therefore, the expected cost of building a random binary search tree by repeated insertions is

$$\sum_{i=1}^{n} \mathbf{E}[C_i] = c \sum_{i=1}^{n} (\mathbf{E}[L_i] + \mathbf{E}[R_i]) \le c \sum_{i=1}^{n} 2H_i = O(n \log n) \ ,$$

where $c$ is a positive constant.

Thus, given $n$ distinct keys, we can randomly permute them and then insert them into a binary search tree. An in-order traversal of the resulting tree can then be used to output the elements in sorted order. This gives us a sorting algorithm whose expected running time is $O(n \log n)$. In fact, the famous Quicksort algorithm is actually just an efficient implementation of this procedure.

Once we have constructed a random binary search tree we can use it as a data structure for searching. The expected cost of searching for some key $k$ is proportional to the length of $\mathrm{path}(k, T)$. There are two cases to consider. If $k$ is not stored in $T$ then the above argument shows that the expected cost of searching for $k$ is

$$\mathbf{E}[|\mathrm{path}(k, T)|] = H_i + H_{n-i} \ , \tag{2.4}$$

where $i$ is the *rank*, i.e., the number of elements less than $k$, of $k$ in $T$.

If $k$ is stored in $T$ then the upper bound

$$\mathbf{E}[|\mathrm{path}(k, T)|] \le H_i + H_{n-i} \ ,$$

still holds, since $\mathrm{path}(k, T)$ only visits the records in $S'$ and $S''$. However, it does not visit all the records in $S'$ and $S''$ since it never visits any node that was inserted after the insertion of $k$. Therefore, if we want a more exact result we must account for the time at which $k$ was inserted into $T$.

The expected number of elements of $S'$ visited while searching for $k$ is

$$\begin{aligned}
\mathbf{E}[R] &= \sum_{j=1}^{|S'|} H_j \times \Pr\{k \text{ appears at position } j \text{ in } S'\} \\
&= \frac{1}{|S'|} \sum_{j=1}^{|S'|} H_j \\
&= \frac{1}{|S'|} \times \left( \begin{array}{ccccccccccc}
1 & + & & & & & & & & & \\
1 & + & 1/2 & + & & & & & & & \\
1 & + & 1/2 & + & 1/3 & + & & & & & \\
\vdots & + & \vdots & + & \vdots & + & \ddots & & & & \\
1 & + & 1/2 & + & 1/3 & + & 1/4 & + & \cdots & + & 1/|S'|
\end{array} \right) \\
&= \frac{1}{|S'|} \times \left( |S'| + \frac{|S'|-1}{2} + \frac{|S'|-2}{3} + \frac{|S'|-3}{4} + \cdots + 1/|S'| \right) \\
&= 1 + 1/2 + 1/3 + 1/4 + \cdots + 1/|S'| - \frac{1}{2|S'|} - \frac{2}{3|S'|} - \frac{3}{4|S'|} - \cdots - \frac{|S'|-1}{|S'|^2} \\
&\ge H_{|S'|} - 1 \ .
\end{aligned}$$

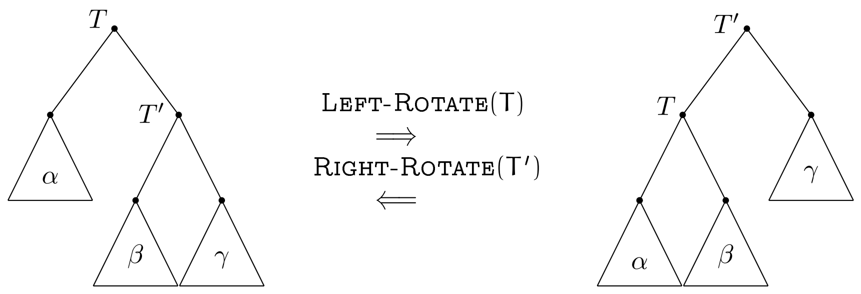

Figure 2.2: Left and right rotations.

A symmetric argument shows that the expected number of elements of $S''$ visited while searching for $k$ is

$$\mathbf{E}[L] \geq H_{|S''|} - 1 \ .$$

Thus, the expected length of the search path for $k$ when $k$ is stored in $T$ is

$$H_i + H_{n-i} \geq \mathrm{E}[|\mathrm{path}(k,T)|] \geq H_i + H_{n-i} - 2 \ , \tag{2.5}$$

where $i$ is the rank of $k$ in $T$.

**Theorem 1.** *The expected cost of constructing a random binary search tree on $n$ elements is $O(n \log n)$. Once such a tree is constructed it can be used to search for any key in $O(\log n)$ expected time.*

## 2.2 Treaps

Next we show how randomization can be used to implement *dictionary* operations. A *dictionary* $D$ is an abstract data type supporting the following operations on a set of real-valued keys.

1. INSERT$(k, D)$. Insert the key $k$ into $D$. This assumes that $k$ is not already stored in $D$.
2. DELETE$(k, D)$. Delete the key $k$ from $D$. This assumes that $k$ is already stored in $D$.
3. SEARCH$(k, D)$. Return the smallest key $k'$ contained in $D$ such that $k' \geq k$. If no such $k'$ exists then return $\infty$.

The reason we can not just use random binary trees to implement dictionary operations is that they are a *static* data structure. Once we build a random binary search tree on a set of keys we can not perform insertion or deletions into them and still maintain the expected operation times of $O(\log n)$. Traditionally, this problem is overcome by introducing a *balancing scheme* like those used in red-black [7] or AVL [1] trees. These balancing schemes use *rotations* to ensure that every root to leaf path in $T$ has length in $O(\log n)$. A rotation swaps a tree and one of its children in such a way that the order in which keys appear in an in-order traversal is maintained. An example of a rotation is given in Figure 2.2.

In this section we present a simple balancing scheme that uses randomization. A *treap* is a search tree in which each subtree $T$ also has a unique *priority* $\mathrm{priority}(T) \in [0, 1]$. In addition to having the sorted property (Property 1), every non-nil treap also has the following *heap property*.

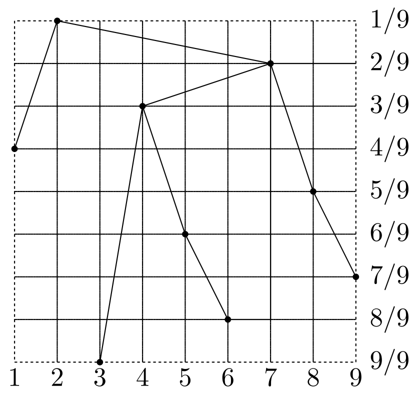

Figure 2.3: We obtain a treap by setting priority$(T^i) = i/n$.

**Property 2** (Heap Property). *If* left$(T) \neq$ **nil** *then* priority$(T) <$ priority(left$(T)$) *and if* right$(T) \neq$ **nil** *then* priority$(T) <$ priority(right$(T)$)

The link between treaps and random binary search trees is the following: If $k_i$ was the key that created the subtree $T^i$ in a random binary search tree, then we obtain a treap by setting priority$(T^i) \leftarrow i/n$. Essentially, we are setting priority$(T^i)$ to be the time at which $k_i$ was inserted. See Figure 2.3.

The interesting thing about this relationship is that it works both ways. If we assign priorities uniformly and independently at random from $[0, 1]$ to all keys then the resulting treap is a random binary search tree on the keys. It's as if we've inserted the keys in random order. We call such a treap a *random treap*. Thus, the nice properties of search paths on random binary search trees also apply to random treaps. It follows immediately that the expected cost of searching for a key $k$ in a random treap is $O(\log n)$.

To insert the key $k$ into a random treap, we first insert it the way we normally would into a binary search tree. We then assign the key a priority $p$ selected uniformly at random from $[0, 1]$ and apply left and right rotations (as appropriate) to move the new subtree upwards until the heap property (Property 2) is restored. See Figure 2.4 for an example.

What is the expected cost of inserting the key $k$ into a random treap $T$? Since a random treap has all the properties of a random binary search tree, the expected cost of performing the basic insertion is proportional to the expected length of path$(k, T)$. From (2.4) we see that $\mathbf{E}[|\text{path}(k, T)|] = H_i + H_{n-i}$. Next, the algorithm performs rotations to restore the heap property. Equation (2.5) show that after performing these rotations, $\mathbf{E}[|\text{path}(k, T)|] \geq H_i + H_{n-i} - 2$. Since each rotation reduces $|\text{path}(k, T)|$ by one, it follows that the expected number of rotations is at most 2.

To perform deletion of the key $k$ from a treap, we find the subtree $T'$ having $k$ as its key and repeatedly perform a rotation that moves $T'$ towards the child of $T'$ that has minimum priority. That is, we perform a left or right rotation (as appropriate) on the child of $T'$ that has smaller priority. We repeat this until $T'$ becomes a leaf, at which point it is deleted by setting a pointer in its parent to **nil**. For an example of deletion, look at Figure 2.4 starting at the bottom right.

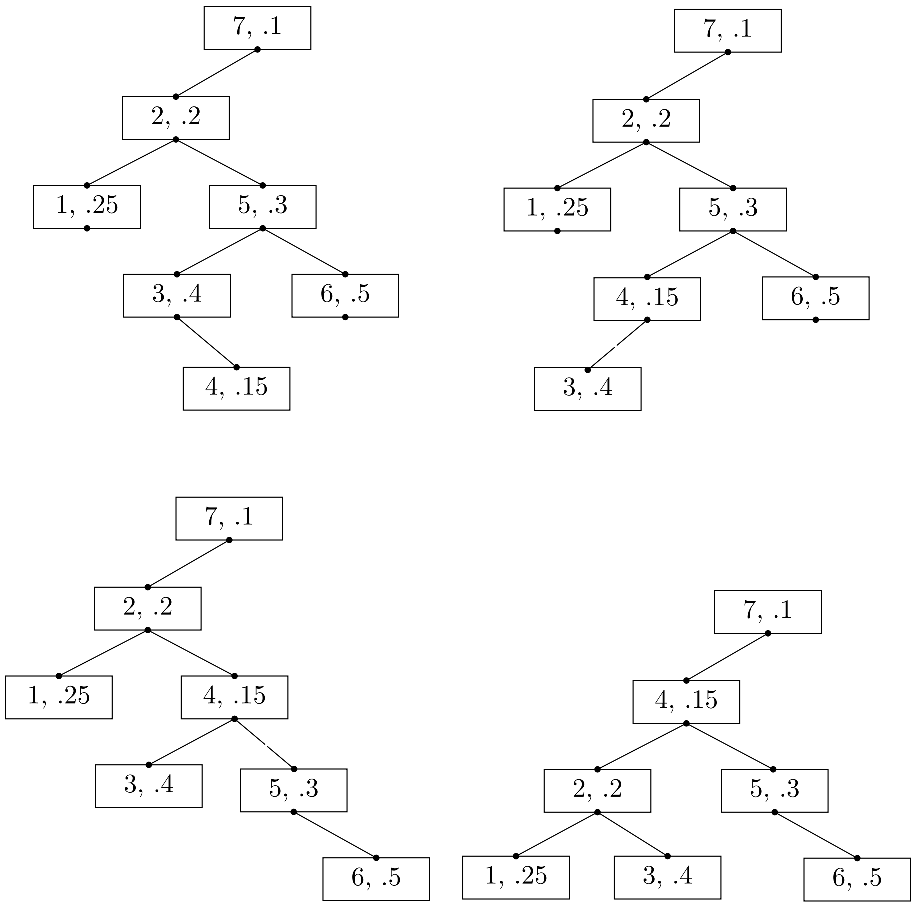

Figure 2.4: Inserting the key $k = 4$ with priority $p = .15$ into a treap.

At first the analysis of deletion seems somewhat more complicated than that of insertion. However, the following observation makes it trivial. Let $T^*$ be the treap that would have existed if the key $k$ had never been inserted. Then $T^*$ is a random treap with $n-1$ nodes and the expected cost of inserting $k$ into $T^*$ is $O(\log n)$. Now, note that the rotations performed while deleting $k$ from $T$ are exactly the same rotations performed while inserting $k$ into $T^*$, only they are done in reverse order. Therefore, the expected number of rotations performed while deleting $k$ is $\Theta(1)$, so the overall cost of deletion is $O(\log n)$. (Note that, if a pointer to the node containing $k$ is given then the expected cost of deletion is $O(1)$.)

**Theorem 2.** *Random treaps support the operations* INSERT*,* DELETE *and* SEARCH *in* $O(\log n)$ *expected time per operation, where* $n$ *is the number of keys stored in the treap at the time of the operation. The expected number of rotations performed during an insert or delete operation is* $O(1)$.

Another class of operations that treaps support very well are the SPLIT and MERGE operations. A SPLIT operation takes a value $k$ not in the treap and splits the treap into two treap $T_1$ and $T_2$ such that $T_1$ contains all keys less than $k$ and $T_2$ contains all keys greater than $k$. To implement a split operation we simply observe that we can insert $k$ and then perform rotations until $k$ becomes the root of the treap. At this point, the two treaps $T_1$ and $T_2$ that we want are the left and right children of the root. The cost of this is proportional to $|\text{path}(k, T)|$, and so the split operation can be done in $O(\log n)$ time.

The inverse of a SPLIT operation is a MERGE operation. This is where we take two treaps $T_1$ and $T_2$ such that all keys in $T_1$ are less than $k$ and all keys in $T_2$ are greater than $k$ and we merge $T_1$ and $T_2$ into a single treap. The implementation of merge is the exact inverse of split: We make a new root $k$ whose left and right children are $T_1$ and $T_2$, respectively, and then we delete $k$ from the treap. Again, the expected cost of this is $O(\log n)$.

**Theorem 3.** *Random treaps support the operations* SPLIT *and* MERGE *in* $O(\log n)$ *expected time per operation.*

## 2.3 Heaters

In the previous section we showed that by assigning random priorities to the nodes of a binary search tree we get a balanced binary search tree. In this section we will see that this trick also works the other way. If we assign random keys to the nodes of a priority queue, we get a balanced priority queue.

A *priority queue* $Q$ is an abstract data type supporting the following operations on a set of real-valued priorities.

1. INSERT$(p, Q)$. Insert the priority $p$ into $Q$. This assumes that $p$ is not already stored in $Q$.
2. FINDMIN$(Q)$. Return the smallest priority $p^*$ stored in $Q$. This assumes $Q$ is not empty.
3. DELETEMIN$(Q)$. Delete the smallest priority $p^*$ stored in $Q$ from $Q$. This assume $Q$ is not empty.

To implement a priority queue we can use a randomized binary tree as in the previous section. When we use a randomized binary tree in this way, we call it a *heater*. The difference from the previous section is that now instead of using random priorities, we will use random keys to obtain a *random heater*. Note that switching from random priorities to random keys makes no difference in how the shape of the underlying tree is distributed. In either case, the shape of the tree is distributed in the same way as the shape of a random binary search tree.

To insert the priority $p$ into a heater $H$, we choose a random real number $k \in [0, 1]$ and insert the key/value pair $(k, p)$ as described in the previous section. The cost of this insertion is proportional to the length of $\text{path}(k, H)$. Since the expected length of $\text{path}(k, H)$ is $O(\log n)$, the expected cost of insertion is $O(\log n)$.

Finding the minimum priority $p^*$ stored in a heater $H$ is trivial since the heap property ensures that the $p^*$ is stored at the root of $H$. Thus, FindMin can be implemented in constant time using a heater.

To delete the minum priority $p^*$ from a heater $H$, we simply delete the root of $H$ using the deletion algorithm described in the previous section. Let $k^*$ be the key associated with $p^*$ and let $H^*$ be the heater obtained after deletion of $p^*$. Then, as with treaps, the cost of deletion is proportional to the length of $\text{path}(k^*, H^*)$. Since $H^*$ is a random treap, the expected length of $\text{path}(k^*, H^*)$ is $O(\log n)$ and the expected cost of deletion is $O(\log n)$.

**Theorem 4.** *Random heaters support the operations* Insert *and* DeleteMin *in* $O(\log n)$ *expected time and* FindMin *in constant time per operation, where* $n$ *is the number of priorities stored in the heater at the time of the operation.*

## 2.4 Discussion and References

The use of records to analyze random binary search trees was introduced by Devroye [5]. His lecture notes [6] use records to give ultra-short proofs about many random phenomena. Randomized trees (treaps and heaters) are introduced by Vuillemin [8]. Aragon and Seidel [2] reinvestigate randomized trees and are responsible for the name treap. Basch *et al* [3] study the use of heaters in the context of kinetic priority queues. These are priorities queues in which the priorities are changing continuously over time.

## Bibliography

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Soviet Mathematics. Doklady*, 3:1259–1263, 1962.

[2] C. R. Aragon and R. Seidel. Randomized search trees. *Algorithmica*, 16(4):464–497, 1996.

[3] J. Basch, L. Guibas, and G. D. Ramkumar. Reporting red-blue intersections between two sets of connected line segments. *Lecture Notes in Computer Science*, 1136:302–314, 1996.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge MA, 1990.

[5] L. Devroye. Applications of the theory of records in the study of random trees. *Acta Informatica*, 26:123–130, 1988.

[6] L. Devroye. Probabilistic analysis of algorithms and data structures (lecture notes). Manuscript, 2001.

[7] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS'78)*, pages 8–21, 1978.

[8] J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.