A Lecture on Cartesian Trees

Written by: Kaylee Kutschera, Pavel Kondratyev, Ralph Sarkis
Corrected by: Simran Awadia
Corrected by: Anna Brandenberger, Luc Devroye

March 8, 2022

This is the augmented transcript of a lecture given by Luc Devroye on the 23rd of February 2017 for a Data Structures and Algorithms class (COMP 252). The subject was Cartesian trees and quicksort.

Cartesian Trees

Definition 1. A **Cartesian tree**[1] is a binary tree with the following properties:

1. The data are points in $\mathbb{R}^2$: $(x_1, y_1), \ldots, (x_n, y_n)$. We will refer to the x_i's as keys and the y_i's as time stamps.
2. Each node contains a single pair of coordinates.
3. It is a binary search tree with respect to the x-coordinates.
4. All paths from the root down have increasing y-coordinate.

The Cartesian tree was introduced in 1980 by Jean Vuillemin in his paper "A Unifying Look at Data Structures." It is easy to see that if all x_i's and y_i's are distinct, then the Cartestian tree is unique — its form is completely determined by the data (in any order).

[1] Vuillemin [1980]

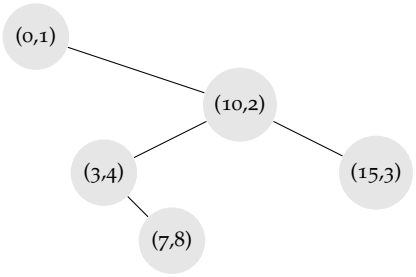

Figure 1: Example of a Cartesian tree with 5 nodes. No other configuration for these nodes will satisfy the properties of the Cartesian tree.

Ordinary Binary Search Tree

An ordinary binary search tree for $x_1, \ldots, x_n$ can be obtained by using the values 1 to n in the y-coordinate in the order that data was given: $(x_1, 1), \ldots, (x_n, n)$. Unfortunately, this can result in a very unbalanced search tree with $height = n - 1$.

Random Binary Search Tree (RBST)

Let $(\sigma_1, \ldots, \sigma_n)$ be a random uniform permutation of $(1, \ldots, n)$. Then the data of a RBST is $(x_{\sigma_1}, 1), \ldots, (x_{\sigma_n}, n)$ or equivalently $(x_1, \tau_1), \ldots, (x_n, \tau_n)$, where $(\tau_1, \ldots, \tau_n)$ is the reverse permutation of $(\sigma_1, \ldots, \sigma_n)$, i.e., $\sigma_i = j$ if and only if $\tau_j = i$. This inverse permutation will also be random and uniform as there is a 1-to-1 relationship with the original permutation.

Treap

Suppose we have data $x_1, \ldots, x_n$ that we want to store. To build a treap, we generate n independent random numbers $T_1, \ldots, T_n$ called random time stamps and pair each x_i with a T_i, thus giving data pairs (x_i, T_i). The T_i's can be considered as uniform [0,1] random numbers. A treap is a combination of a binary search tree and a heap where the x_i's are keys to the binary search tree and T_i's are the keys of the heap. Although treaps are not fully balanced trees, they have expected height $O(\ln(n))$ and form a typical Cartesian tree. The treap was first described in Aragon and Seidel [1989]. Treaps are random binary search trees that are easy to maintain as data are added and deleted.

Operations on Cartesian Trees

Compared to other data structures and especially other search trees that we have seen in class, a Cartesian tree has a nice structure that lets us define complex operations in terms of two atomic operations. We will first see the two FIX operations and then look at how they are used to construct the others.[2]

[2] In definition 1, the fourth property comes from the heap data structure. We give two simpler formulations for this property (henceforth called the heap property):

- For any node, the parent has a smaller y coordinate.
- For any node, the children have a greater y coordinate.

We will use these simpler formulations because they correspond more to the operations we will describe.

Atomic operations

Firstly, we informally define what we mean by a fix. The input is a Cartesian tree in which only one node, v, does not satisfy the heap property. We fix the Cartesian tree by moving v inside the tree, keeping the invariant that only v does not satisfy the heap property. During this procedure, the node v can only move one way, either upwards (FIX-UP) or downwards (FIX-DOWN). Since a Cartesian tree is finite, the procedure must halt and v must be fixed when this happens (it is not clear why at the moment but it will be explained in the following paragraphs).

When we say that we are moving a node in the tree, we cannot do this in an arbitrary manner. Each change we make in the tree should keep the invariant mentioned above. We will use a simple move called a tree rotation[3] in order to either push a node above its parent or below its children. The particularity of the rotation is that it maintains all the properties of the Cartesian tree. It is easier to describe this pictorially so we refer to Figure 2.

[3] More information on rotations in Sleator et al. [1988]

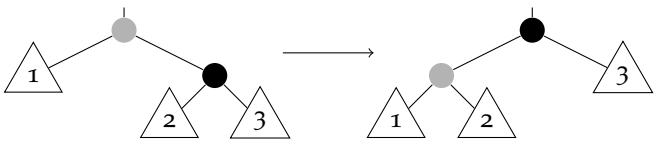

Figure 2: Example of a tree rotation that would be done during a FIX-UP operation. The black node is the misplaced and should be above the gray node.

In this diagram representing one iteration of a FIX-UP operation, we have drawn two nodes and three subtrees which would be part of a bigger tree. The tagged node (in black) is misplaced, namely, the y-coordinate of its parent is bigger, however, all of its descendants are well-sorted. We will use a rotation to try to fix it without messing the x-wise order. We put the black node above the gray one by replacing the left child of the tagged node with its parent. Then, we put sub-tree 2 as the right child of the gray node. Hence, sub-tree 2 is to the left of the black node and to the right of the gray node and the previous ordering is maintained. This is called a right rotation because the tagged node is the right child of its parent. Nevertheless, by changing the direction of the arrow and the colors of the node in the diagram above, we obtain the completely symmetric left rotation.

Secondly, we will describe the FIX-UP operation in more depth and argue about the correctness of the operation. As we mentioned above, the input is a node v that has a parent with a larger y-coordinate. Moreover, it is the only unsorted node in the sense that if you list all y-coordinates from the root to a descendant leaf of v, you obtain a list in increasing order with one unsorted element. Viewing it as a list of y coordinates, the procedure is very similar to an insertion sort. We will loop and halt when the y-coordinate of v is greater than that of its parent or when v is the root of the tree. At each iteration, we either do a right rotation or a left rotation to make the node v go up.

In terms of a Cartesian tree, the y-coordinate represents the *timestamp* of a node. Hence, there are two possibilities for the loop to stop.

1. If $timestamp[v] > timestamp[parent[v]]$; then v must satisfy the heap property because $timestamp[v]$ is smaller than all the *timestamps* of its descendants (which form a well ordered Cartesian tree) and it is greater than all the *timestamps* of its ancestors[4] (well ordered as well)

2. If v is at the root; then this means its *timestamp* is smaller than every other node in the tree and hence v also satisfies the heap property.

[4] Recall the insertion sort comparison

We will demonstrate its implementation below: u represents the node that is out of order and t is the Cartesian tree.

The following notation will be used: $key[u]$, $left[u]$, $right[u]$, $time[u]$, $parent[u]$.

```
FIX UP(u, t):
 1  while u ≠ root && time[parent[u]] > time[u]
 2      w = parent[u]
 3      if w ≠ root then
 4          z = parent[w]
 5          if right[z] == w then right[z] = u else left[z] = u
 6      if right[w] == u then
 7          right[w] = left[u]
 8          left[u] = w
 9          parent[u] = parent[w]
10          parent[right[w]] = w
11          parent[w] = u
12      else
13          left[w] = right[u]
14          right[u] = w
15          parent[u] = parent[w]
16          parent[left[w]] = w
17          parent[w] = u
```

For the FIX-DOWN operation, we will use the same rotation as described above but we want to make the tagged node move down, as shown in Figure 3.

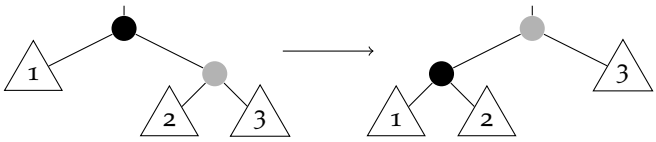

Figure 3: Example of a tree rotation that would be done during a FIX-DOWN operation. Note that the only difference is that the tagged node is the parent at first and goes down in the tree.

The algorithm for this operation is quite similar but it loops until v becomes a leaf or until its y-coordinate (*time*) becomes smaller than that of its children. Furthermore, there is a slight catch when choosing to do a left rotation or right rotation. Since we will put one of the children above the other, we must choose to rotate at the child with the smallest y-coordinate in order to maintain the heap property. We show the case illustrated above in the *else* block of the following code.

```
FIX DOWN(u, t):
 1  while u ≠ leaf && (time[u] > time[left[u]] OR time[u] > time[right[u]])
 2      ▷ We assume that the time of a non-existing node is ∞
 3      if time[right[u]] > time[left[u]] then
 4          w = left[u]
 5          left[u] = right[w]
 6          right[w] = u
 7          parent[w] = parent[u]
 8          parent[left[u]] = u
 9          parent[u] = w
10      else
11          w = right[u]
12          right[u] = left[w]
13          left[w] = u
14          parent[w] = parent[u]
15          parent[right[u]] = u
16          parent[u] = w
```

Other operations

We will briefly describe four operations that follow almost immediately from the two atomic operations. We leave the details of the pseudocode as an exercise.

1. INSERT(t,(x, y)): Inputs are a Cartesian tree t and a node (x, y) to be added to the tree. First, insert the node (x, ∞) just as you would in a binary search tree, implying that the node is a leaf since the heap property must be satisfied. Second, change the node to (x, y) and use FIX-UP to fix the Cartesian tree.

2. DELETE(t, u): Inputs are a Cartesian tree t and a pointer u to a node that must be removed. First, change $time[u]$ (the time stamp of u) to ∞ and use FIX-DOWN to move the node down the tree until it end up as a leaf. Then simply remove that leaf.

3. JOIN(t_1, t_2): Inputs are two Cartesian trees that need to be joined into a single tree. It is understood that all keys in t_1 are smaller than all keys in t_2. First, create a temporary node $(k, -\infty)$ such that $\text{MAX}_X(t_1) < k < \text{MIN}_X(t_2)$ and let its left child be the root of t_1 while its right child is the root of t_2. Second, delete the node and the tree will fix itself up.

4. SPLIT(t,k): Inputs are a Cartesian tree t and a value k that will split the tree into two trees that have x coordinates smaller than k and

bigger than k, respectively. First, insert a temporary node $(k, -\infty)$ (it will end up as a root) and after the procedure, the left subtree and right subtree of the node are the trees we are looking for.

Treaps and Abacus

Another way to look at a Cartesian tree is such that all the nodes are attached to rods of an abacus. Following from left to right we have the in-order numbering as shown below by the vertical dotted lines and top to bottom we follow a min heap structure, represented by the number inside each node. It is important to realize that we can learn about the structure of the tree even without connecting the nodes. Consider the node u, by noticing the time stamp of the node following and preceding it in the in-order numbering, we can predict that u would be a leaf. This is because both its neighbours v and w have a timestamp lower than that of u and thus are both ancestors of u.

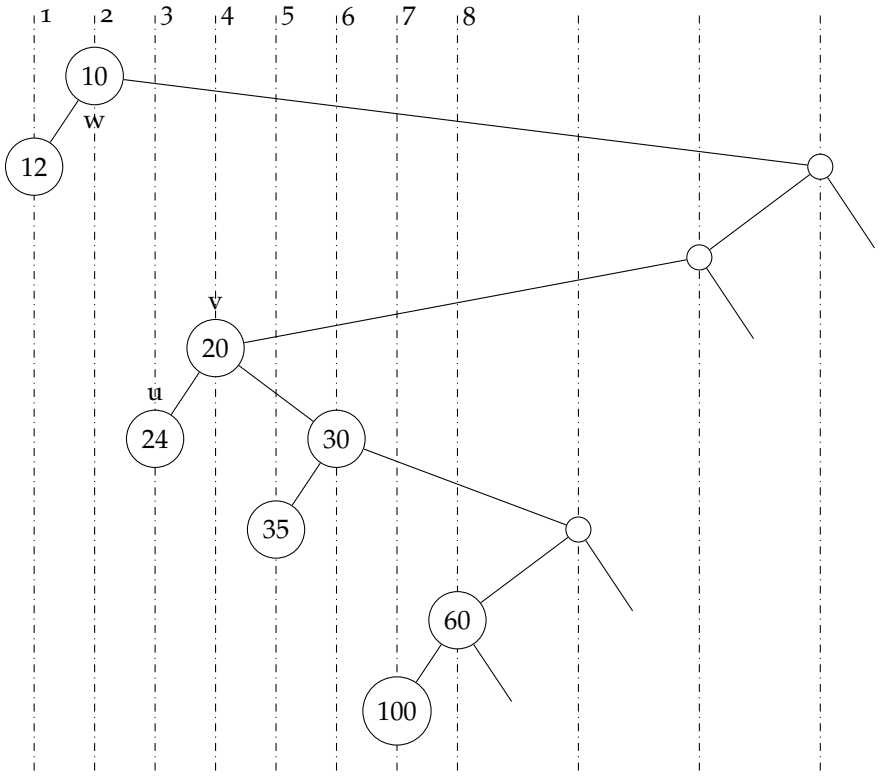

Figure 4: Treaps and Abacus

Probability review

To provide a probabilistic analysis of treaps, we have to provide some background on probability.[5] We define a **sample space** (usually denoted as Ω) as a set of **elementary events**, or possible outcomes of an experiment. We also denote $\mathcal{S}$ as the set of all **events** on Ω, namely all subsets of Ω.

[5] Grimmett and Stirzaker [2001]

Example 2. If a coin is flipped: $\Omega = \{H, T\}$, $\mathcal{S} = \{\{H\}, \{T\}, \{H, T\}, \emptyset\}$

Definition 3. A **probability** ($\mathbb{P}$) is a mapping from a sample space to $\mathbb{R}$ satisfying the following properties:

1 $\forall A \in \mathcal{S}, \mathbb{P}(A) \geq 0$

2 $\mathbb{P}(\Omega) = 1$

3 If $A_i \in \mathcal{S}; i \in \mathbb{N}$; and $\forall i \neq j, A_i \cap A_j = \emptyset$ then $\mathbb{P}\left(\cup_i A_i\right) = \sum_i \mathbb{P}(A_i)$

Example 4. Given the coin flip example from above, $\mathbb{P}(\{H\}) = 1/2$, $\mathbb{P}(\{H, T\}) = 1$, $\mathbb{P}(\emptyset) = 0$.

Definition 5. A **random variable** is a function that associates each outcome in the sample space with a real number.

Example 6. If we want to model an experiment with two outcomes we can express this as $\mathbb{P}(X = 1) = p$ and $\mathbb{P}(X = 0) = 1 - p$. In other words the probability of the first event happening is p, and the probability of the second event is $1 - p$. X is called a Bernoulli random variable of parameter p.

Lastly, we need to develop the idea of an expected value.

Definition 7. The **expectation** (we can think of this as the "mean" or "average") of a random variable X is defined to be

$$\mathbb{E}[X] = \sum_{n=1}^{\infty} x_n \mathbb{P}(X = x_n).$$

Example 8. If X is Bernoulli of parameter $p \in (0, 1)$), then

$$\mathbb{E}[X] = 0(1 - p) + 1 \cdot p = p.$$

Expected value analysis for treaps

In a treap, the data are $(x_1, T_1), \ldots, (x_n, T_n)$, where the T_i are random time stamps. Equivalently, and for simplicity, let the data be $(1, T_1), \ldots, (n, T_n)$ where $1, \ldots, n$ are the ranks of the nodes.

Define D_k to be the depth of the node of rank k, namely (k, T_k). Since the depth of a node is equivalent to the number of ancestors it has, let

$$D_k = \sum_{j \neq k} X_{jk} \text{ where } X_{jk} = \begin{cases} 1, & \text{if } j \text{ ancestor of } k \\ 0, & \text{otherwise} \end{cases}$$

So next we wish to calculate the expected value for D_k.

We will denote H_k to be the harmonic number, $\sum_{n=1}^{k} \frac{1}{n} = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$. It can be approximated by $\ln(k)$ but more importantly, it can be bounded as follows $H_k \in [\ln(k+1), \ln(k)+1]$.

$$\begin{aligned}
\mathbb{E}[D_k] &= \sum_{j \neq k} \mathbb{P}((j, T_j) \text{ is an ancestor of } (k, T_k)) \\
&= \sum_{j \neq k} \mathbb{P}(T_j \text{ is the smallest of } T_j, T_{j+1}, \cdots, T_k) \\
&= \sum_{j<k} \frac{1}{k-j+1} + \sum_{j>k} \frac{1}{j-k+1} \\
&= \left(\frac{1}{2} + \cdots + \frac{1}{k}\right) + \left(\frac{1}{2} + \cdots + \frac{1}{n-k+1}\right) \\
&= (H_k - 1) + (H_{n-k+1} - 1) \\
&\leq 2\ln(n) \\
&\simeq 1.39 \log_2(n).
\end{aligned}$$

If we force $T_k = \infty$ (as in an insert or delete), then

$$\mathbb{E}[D_k] = H_{k-1} + H_{n-k}.$$

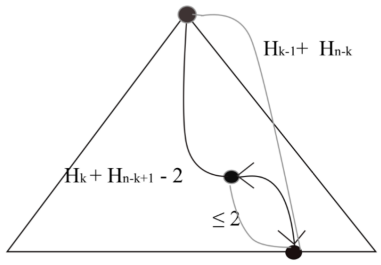

Figure 5: Representation of a treap with the expected height difference when inserting a new node.

So next we wish to see how many FIX steps are expected when inserting (k, T_k). Using the two results above, we find that it is equal to

$$(H_{k-1} + H_{n-k}) - (H_k + H_{n-k+1} - 2) \leq 2.$$

Using Figure 4 and the above argument, we can see that it is expected to take at most two steps to fix the Cartesian tree.

We have shown that search, insert, and delete take $O(\log(n))$ expected number of steps, making the treap a viable data structure for the abstract data type "dictionary".

Quicksort

Given a set of numbers, Quicksort[6] picks a pivot at random from the set, partitions the remaining numbers into elements smaller than the pivot and elements bigger than the pivot and recurses into the two new sets. There are many different algorithms to partition the elements of the set but it must be done in $O(n)$ time for a set of n elements. A simple pseudocode follows.

One can see that the Quicksort procedure is equivalent to building a random binary search tree. In fact, the number of comparisons needed to build a random binary search tree is the same needed to Quicksort.

[6] This algorithm was first introduced by Tony Hoare [1962], while a full analysis of the algorithm was done by Robert Sedgewick [1977] in his Ph.D. thesis.

Algorithm 1: QuickSort($list, low, high$)

QUICKSORT$(list, low, high)$:

```
1  ▷ sort list between indices low and high
2  if low < high then
3      j = random index in the range [low, high]
4      x = key[j]
5      Determine all indices i ≠ j in [low, high] with key[i] ≤ x.
6      Let their number be N.
7      Rearrange the items in list from [low, high] so that the N elements
8      are before x, which in turn is before the other elements.
9      QuickSort(list, low, low + N − 1)
10     QuickSort(list, low + N + 1, high)
```

Expected number of comparisons

Let C_n = # of comparisons $= \sum_{i=1}^{n} \sum_{j \neq i} X_{ji} = \sum_{i=1}^{n} D_i$. Exchanging summation signs, one can show that

$$\begin{aligned}\mathbb{E}[C_n] = \sum_{i=1}^{n} E[D_i] = \sum_{i=1}^{n} (H_i + H_{n-i+1} - 2) = 2\sum_{i=1}^{n} (H_i - 1)\\ = 2(n+1)H_n - 4n\\ \sim 2n\ln(n)\\ = 1.386294\ldots n\log_2(n).\end{aligned}$$

It is noteworthy that this is about 39% worse than if we had used mergesort, or indeed, the best possible comparison-based sorting method.

Improvements

One can improve Quicksort, by picking three random numbers and taking the median of the three as the pivot. The resulting expected number of comparisons is $\sim \frac{12}{7} n \ln(n)$. If we take the median of five rather than three random numbers, the expected number of comparisons becomes $\sim \frac{60}{37} n \ln(n)$.

Exercise 9. Show that the expected number of FIX-DOWN rotations needed for the operation JOIN on two treaps of size m and n, respectively, is $H_n + H_m$.

References

C. R. Aragon and R. G. Seidel. Randomized search trees. *30th Annual Symposium on Foundations of Computer Science*, pages 540–545, Oct

1989. DOI: 10.1109/SFCS.1989.63531. URL http://faculty.washington.edu/aragon/pubs/rst89.pdf.

Geoffrey R. Grimmett and David R. Stirzaker. *Probability and Random Processes*. Oxford University Press, 3rd edition, 2001. ISBN 9780198572237.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5:10–16, Jan 1962. DOI: 10.1093/comjnl/5.1.10. URL https://doi.org/10.1093/comjnl/5.1.10.

Robert Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7(4):327–355, 1977. ISSN 1432-0525. DOI: 10.1007/BF00289467.

Daniel D. Sleator, Robert E. Tarjan, and William P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *Journal of the American Mathematical Society*, 1(3):647–681, 1988. ISSN 08940347, 10886834. URL http://www.jstor.org/stable/1990951.

Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23:229–239, Apr 1980. DOI: 10.1145/358841.358852. URL https://pdfs.semanticscholar.org/1742/195ba5043ec45345d6a9c9ee65145d345ecd.pdf.