A Lecture on Cartesian Trees
Kaylee Kutschera, Pavel Kondratyev, Ralph Sarkis
February 28, 2017

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 Cartesian tree is unique — its form is completely determined by the data.

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 \((\sigma_1, \ldots, \sigma_n)\). 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 \(n\) random numbers \(T_1, \ldots, T_n\) called random time stamps and pair each \(x_i\) with a \(T_i\), thus giving data pairs \((x_i, T_i)\).

---

\(^1\) 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.
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, but are easier 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.\(^2\)

**Atomic operations**

Firstly, we informally define what we mean by a fix. The input is $v$ 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 have to refer to Figure 2.

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

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

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

\(^2\) 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.

\(^3\) More information on rotations in Sleator et al. [1988]
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. There are two possibilities for the loop to stop. If \( v.y > v.parent.y \), then \( v \) must satisfy the heap property because the \( y \)-coordinate of \( v \) is smaller than all the \( y \)-coordinates of its descendants (which form a well ordered Cartesian tree) and it is greater than all the \( y \)-coordinates of its ancestors\(^4\) (well ordered as well). If the loop stops because \( v \) is at the root, then this means its \( y \)-coordinate is smaller than every other node in the tree and hence \( v \) also satisfies the heap property.

**FIX-UP**(node)

```cpp
while node \neq root && node.y < node.parent.y
    // We assume the parent pointers are updated as well
    if node == node.parent.right then
        node.parent.right = node.left
        node.left = node.parent
    else // Completely symmetric process to above
        node.parent.left = node.right
        node.right = node.parent
```

\(^4\)Recall the insertion sort comparison

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.
The algorithm for this operation is quite similar but it loops until $v$ becomes a leaf or until its $y$-coordinate 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.

**FIX-DOWN**\((\text{node})\)

1. \textbf{while} \(\text{node} \) is not leaf \&\& \((\text{node}.y > \text{node}.left.y \mid \mid \text{node}.y > \text{node}.right.y)\)
2. \/// We assume that the $y$ coordinate of a non-existing node is $\infty$
3. \textbf{if} \(\text{node}.right.y > \text{node}.left.y\) \textbf{then}
4. \hspace{1em} \(\text{node}.left = \text{node}.left.right\)
5. \hspace{1em} \(\text{node}.left.right = \text{node}\)
6. \textbf{else} \/// Completely symmetric process to above
7. \hspace{1em} \(\text{node}.right = \text{node}.right.left\)
8. \hspace{1em} \(\text{node}.right.left = \text{node}\)

**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. \textbf{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 $(x, \infty)$ just as you would do it in a binary search tree, implying the node will be a leaf since the heap property must be satisfied. Second, change the node to $(x, y)$ and use \textbf{FIX-UP} to fix the Cartesian tree.

2. \textbf{DELETE}(t, node): Inputs are a Cartesian tree $t$ and a pointer $node$ to a node that will be removed. First, change $node.y$ to $\infty$ and use \textbf{FIX-DOWN} to move the node down the tree. It will end up as a leaf so the second step is just to remove the node.

3. \textbf{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.

**Probability review**

To provide a probabilistic analysis of treaps, we have to provide some background on probability. We define a **sample space** \(\Omega\) as a set of elementary events, or possible outcomes of an experiment. We also denote \(S\) as the set of all events on \(\Omega\), namely all subsets of \(\Omega\).

**Example 2.** If a coin is flipped: \(\Omega = \{H, T\}\), \(S = \{\{H\}, \{T\}, \{H, T\}, \emptyset\}\)

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

1. \(\forall A \in S, P(A) \geq 0\)
2. \(P(\Omega) = 1\)
3. If \(A_i \in S; i \in \mathbb{N}; \text{ and } \forall i \neq j, A_i \cap A_j = \emptyset\) then \(P(\bigcup_i A_i) = \sum_i P(A_i)\)

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

**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 \(P(X = 1) = p\) and \(P(X = 0) = 1 - p\). In other words the probability of the first event happening is \(p\), and the probability of the second event is \(1-p\). This particular example is often referred to as the Bernoulli random variable.

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

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

**Example 8.** If \(X\) is the Bernoulli random variable(as above) \((p \in (0,1)), X \in \{0,1\}\) then

\[
E[X] = 0(1 - p) + 1 \cdot p = p.
\]
**Expected value analysis for treaps**

In a treap, data is represented as \((x_1, T_1), \ldots, (x_n, T_n)\), where the \(T_i\) are random time stamps. Equivalently, and for simplicity, let the data be \((1, T_1), \ldots, (n, T_n)\) where \(1, \ldots, n\) will be refereed to as the rank of the node/pair.

Define \(D_k\) to be the depth of the node with rank \(k\), namely \((k, T_k)\).

Since the depth of a node is equivalent to the number of ancestors it has, let

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

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

\[
\mathbb{E}[D_k] = \sum_{j \neq k} \mathbb{P}((j, T_j) \text{ is an ancestor of } (i, T_i))
\]

\[
= \sum_{j \neq k} \mathbb{P}(T_j \text{ is the smallest of } T_j, T_{j+1}, \ldots, T_k)
\]

\[
= \sum_{j<k} \frac{1}{k-j+1} + \sum_{j>k} \frac{1}{j-k+1}
\]

\[
= \left( \frac{1}{2} + \cdots + \frac{1}{k} \right) + \left( \frac{1}{2} + \cdots + \frac{1}{n-k} \right)
\]

\[
= (H_k - 1) + (H_{n-k+1} - 1)
\]

\[
\leq 2 \ln(n) \approx 1.39 \log_2(n).
\]

If we set \(T_k = \infty\), then

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

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

\[
(H_k + H_{n-k+1} - 2) - (H_{k+1} + H_{n-k}) \leq 2
\]

Using Figure 4 and the above argument, we can see that it is expected to take 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

\(^6\) This algorithm was first introduced by Tony Hoare [1962] and a full analysis of the algorithm was done by Robert Sedgewick [1977] for his Ph.D. thesis.
the pivot and elements bigger than the pivot and recurses into the
two new sets. There are many different algorithm 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.

**QUICKSORT**(*list*, *low*, *high*)

1. $i =$ random index between *low* and *high*
2. $a =$ *list*[i]
3. **if** *high* > *low** then
   4. **Partition** *list*[low...high] so elements before index *j* are
      smaller than *a* and elements after index *j* are greater than *a*
   5. QUICKSORT(*list*, *low*, *j* − 1)
   6. QUICKSORT(*list*, *j* + 1, *high*)

One can see that the Quicksort procedure is equal 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.

**Comparison Complexity**

Let $C_n =$ # of comparisons = $\sum_{i=1}^{n} D_i$.

\[
E[C_n] = \sum_{i=1}^{n} E[D_i]
\]

\[
= \sum_{i=1}^{n} (H_i + H_{n-i+1} - 2)
\]

\[
= 2 \sum_{i=1}^{n} (H_i - 1)
\]

\[
= 2 \sum_{i=1}^{n} H_i - 2n
\]

\[
= 2 \sum_{i=1}^{n} \sum_{j=1}^{i} \frac{1}{j} - 2n
\]

\[
= 2 \sum_{j=1}^{n} \frac{1}{j} \sum_{i=j}^{n} 1 - 2n
\]

\[
= 2 \sum_{j=1}^{n} \frac{1}{j} (n - j + 1) - 2n
\]

\[
= 2(n + 1) H_n - 4n
\]

\[
\sim 2n \ln(n)
\]

\[
= 1.386294 \ldots n \log_2(n)
\]

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 number of comparisons would be approximately $\frac{12}{7} n \ln(n)$. If we take five rather than three random numbers, the number of comparisons would be approximately $\frac{60}{37} n \ln(n)$.

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


