Sorbonne Université
Sciences et ingénierie
Project — Msc Computer Science: DAC

Implementing Cartesian Tree data structures in Java.

Armand Bonn

21400077

Course:	Complexité, algorithmes randomisés et approchés
Date final version:	November 29, 2024

1 Introduction

A Cartesian tree is a data structure that combines the properties of a binary search tree and a stack. All the nodes in a Cartesian tree necessarily have a key and a value. The keys of the Cartesian tree are sorted using a binary search tree method: for each node, all the nodes on the left sub-tree have keys that are smaller than the current node, and all the keys that are bigger are on the right sub-tree of the node. The values of each node a sorted in a similar method of a stack such that every parent node should have a higher priority value than its children node. An example of a cartesian tree with the following values is given in Figure 1:

$$(A:5),(B:3),(C:8),(D:2),(E:6),(F:7),(G:9),(H:1),(I:10),(J:12).$$

The Figure shows how the Cartesian tree is implemented with this specific example. The solution is unique with the given data as the keys and priorities of every node are unique. At the moment the uniqueness of the keys and values is not ensured, different possible solutions to the Cartesian tree are possible. For example, if two nodes would have the same priority it would not matter which one of these two nodes would be the parent to preserve the stack property. Hence, there would be multiple solutions to the problem. The figure given below is the final result of a unique Cartesian tree.

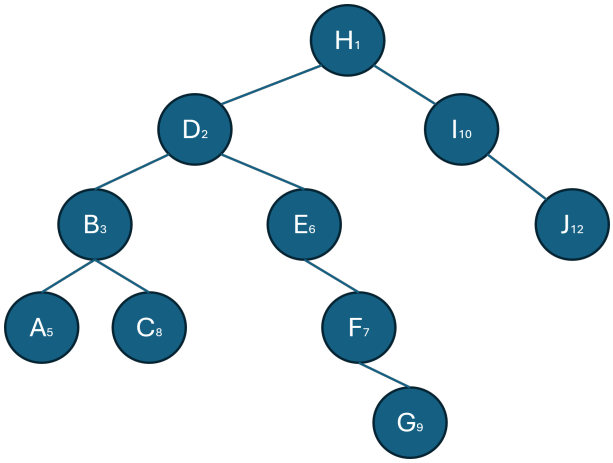


Figure 1: Example of Cartesian tree structure.

When implementing the given data structure the nodes need to be inserted following the binary search tree properties. Once the new node is inserted, it will become one of the leaves of the tree. Afterwards, if the stack properties are violated rotations are applied to ensure all properties of the Cartesian trees. Generally, the issue is to understand the complexity of inserting all nodes into our Cartesian tree. More specifically, what could be the most efficient to insert the nodes in the Cartesian Tree?

Rotations are applied when the priority constraint is not met anymore. Hence, an efficient method to insert the nodes is by the order of their priority (starting with the lowest value). In our specific example, the sorted nodes with respect to priorities are the following: $H, D, B, A, E, F, C, G, I$ and J. Inserting the nodes in this order will give us the same tree structure as given in Figure 1 without having to apply any rotations. Generally, if the nodes are inserted with respect to the priority order, the stack property will not be violated. Moreover, the nodes are automatically inserted with respect to the binary search tree properties.

Now that the general idea of a Cartesian tree has been introduced and an example has been given, the issue is to implement such a Cartesian tree in Java programming. Giving the practical base of the implementation of this data structure is relevant as it has various applications such as in the structure of database models. In this report, the implementation of the Cartesian Tree will be given in the Java programming knowledge.

2 Implementation of a Cartesian Tree in Java

2.1 Basic framework

In the following sub-section, the basic components of the Cartesian Tree class will be given to understand the fundamental programming steps of creating the class. After these have been explained, the following subsections will give the code structure of the insertion and suppression algorithms for the class. The main assumption in the rest of the report is the uniqueness of the keys and priority values. This ensures that the Cartesian tree has only one solution. First, a simple *Node* class is created to be used in the Cartesian Tree, where this class has the attributes key, priority, left-node, right-node, and parent-node. It is assumed that the key- and priorities values are unique. The class has been structured such that any general type can be used for both keys and priorities. The only constraint is that the general classes that are used for the key and priority extend the comparable class. Through this implementation choice, all different types of keys and priorities can be used for a Node in a Cartesian tree. Lastly, when a new node is initiated, it is assumed that it does not have any reference nodes. As a node is not connected to any other node when initiating it, functions are created to add a left-, right-, and parent node for an instance of the *Node* class.

After the node class has been created, the foundations of the *CartesianTree* class can be implemented. The class has as its only attribute the root node which is of type *Node* and initializing it sets the root node to zero. The Cartesian tree class also takes in two general types for its key and values, and these types are then used to create nodes inside its functions. In this first subsection, it will be assumed that the nodes of the tree will be inserted according to their priority order (see Section 1). First, a function was implemented to insert a new node in the tree by only looking at the keys and not the priorities in the tree. This method was named *insertNodeKey* and takes as input a key value and finds the leaf in the tree under which the given node needs to be inserted. The leaf node under which the node needs to be inserted is found through a recursive algorithm, that iterates over the branches until the correct node is found. Through this method, the example in Figure 1 could be implemented by inserting the nodes by order of their priority.

Furthermore, as a starting block of the Cartesian tree the function *printTreePreOrderTraversal* was created to be able to print a constructed tree. As the name tells, the tree is printed in a pre-order traversal with respect to the trees. This ensures that the root node is printed first after which the left subtree and lastly the right subtree of a node.

As of now, in the fundamental framework of the class, the priorities are not yet taken into consideration. However, it is relevant to already write a function *findNode* in order to look for a node with a specific key. The function takes a key as input and returns the node with the given key and null otherwise. The function traverses the tree with accord to the binary search tree properties, starting at the root node of the tree. If the key is smaller than the current node's key it will go to the left sub-tree, and if the key is bigger it will go to the right sub-tree. It will do so recursively till it finds the key or arrives at a leaf. If the function arrives at a leaf and no node exists with the given key, the function will return null.

If the key exists in the tree and the node is at the depth k_1 of the given Cartesian tree, the algorithm will be able to find the node in $O(k_1)$. If the node does not exist, the algorithm will in the worst case stop when it arrives at the height of tree h at the leaf node where the key would be expected. Hence, in this case, the worst-case complexity would be $O(h)$, where h is the height of the tree.

2.2 Insertion method

In the previous section, it was assumed that the nodes of the Cartesian tree would be inserted in the order of priority. In this way, the stack property would not be violated when inserting

the nodes. However, from this section onward that assumption is not made anymore and the insertion algorithm needs to be implemented such that also the stack properties are not violated when a new node is inserted. An example is given in Figure 2, wherein the given data example of Section 1 an extra node $(K : 4)$ is inserted with the methods used previously. In the Figure, it can be seen that the tree does follow all the properties of a binary search tree (given the keys) but the priority property does not hold anymore due to the priority value of 4 of the new node.

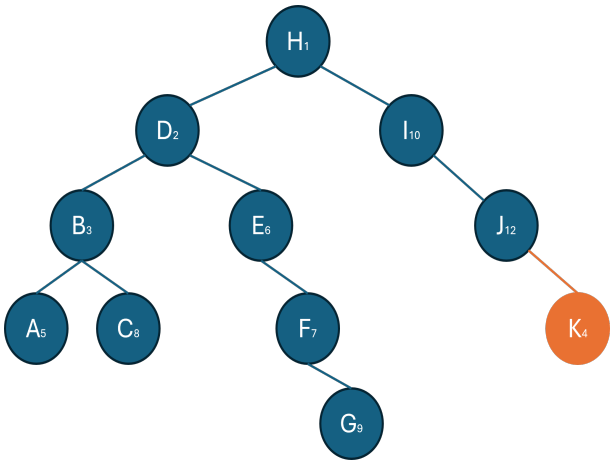


Figure 2: Example of Cartesian tree structure where stack property is violated.

To correct for these violations, the methodology for insertion will be presented in this report. First, a new node is inserted with respect to the binary search tree properties, this method was already formulated in Section 2.1 and is defined as *insertNodeKey* in this report. After the node is inserted using its key value, it will be verified if the priority condition does not hold anymore. If it does not, rotations will be applied such that the inserted node z will moved 1 position higher in the tree and its parent with lower priority will be moved one position lower. This will be done iteratively until the priority condition holds in the Cartesian Tree. The rotation of nodes should be undertaken in constant time $(O(1))$ by managing the pointer reference of the variables. In the worst case, the priority of the inserted node will be the highest in the tree and the node would need to be rotated to the root of the tree. If the height of the predecessor of the inserted node is h and every rotation would take constant time, this means that the worst-case complexity of the algorithm is $O(h)$. An example of the rotations is given in Figure 3, where the distinction is made between the right rotation and left rotation. If node x is lower in the tree while having higher priority than parent y a right rotation is applied. If node y has higher priority than node x and is lower in the tree, a left rotation as shown in the figure 3.

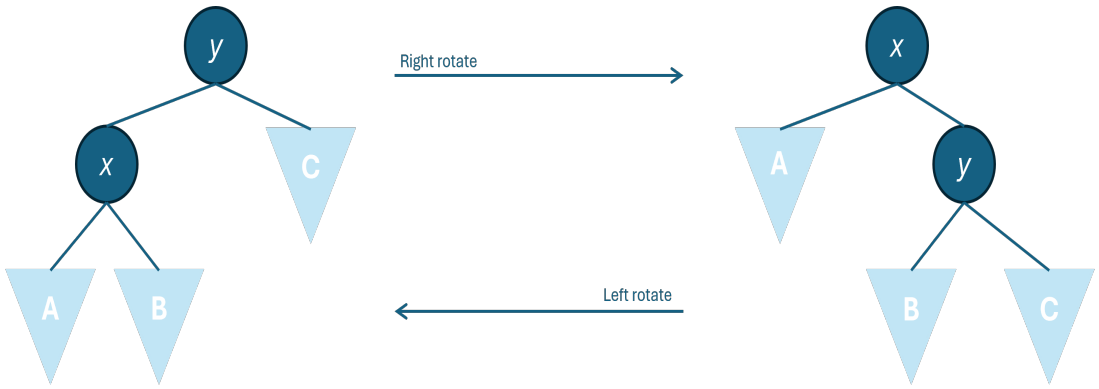


Figure 3: Rotation of nodes in a Cartesian tree to ensure that priority properties are met.

In Algorithm 1, the pseudo-code is given for the insertion function of a new node in a Cartesian Tree. As the base case of the function, if there is no node yet in the tree (the root node is null) the inserted node becomes the root. If there is an existent root, then the function will find the leaf where the new node needs to be inserted. The corresponding leaf node is found through the properties of the binary search tree with its keys. So here the new node will be inserted as a child of the lead node in the correct position corresponding to the key values. This is the same procedure as $InsertNodeKey$ in Section 2.1. As the tree will hold the properties of a binary search tree this worst-case running time to to find the leaf node will be $O(h)$ for a tree with height h. In section 4, it will be shown that the Cartesian tree has on average a height of $h = log(n)$.

After the new node has been inserted in the correct position according to the binary search tree conditions a while loop is initiated. This while loop will continue to perform rotations with parent nodes until the priority condition is held, meaning that the parent does not have a lower priority than the newly inserted nodes. The loop can also be terminated if the new node is the highest in the tree (root) and can not be rotated anymore. The new node is iteratively left-rotated or right-rotated according to its key value relative to the parent node (see Figure 3). The worst-case complexity of this loop would be $O(h)$ where h is the tree's height. This is the case when the inserted node needs to rotate up until it is the root node.

Algorithm 1 Node Insertion function

```
1:  procedure InsertNode(Key, Priority)
2:      if rootNode = null then
3:          Set rootNode with Key and Priority
4:          return
5:      end if
6:      Find insertion leafNode of newNode with Key
7:      Create newNode with Key and Priority
8:      Attach newNode as child of leafNode
9:      while newNode.Priority < parentNode.Priority do
10:         if Key of newNode > Key of parentNode then
11:             Left rotation on newNode
12:         else
13:             Right rotation on newNode
14:         end if
15:         if newNode does not have parentNode then
16:             break
17:         end if
18:     end while
19: end procedure
```

To get a general idea of the structure of the rotation function, the procedure for right rotation is given in Algorithm 2. What is important to note is that to be able to implement the given data structure in Java, another attribute named *parentNode* was added to the class Node so that the reference between layers of the tree could be managed more easily. The procedure performs the rotations that are also shown in Figure 3. Firstly, the right child of the current node (in this case the newly inserted node) is attached as the left child of the parent node (*l.1*). As these references have been changed, the parent of the new left child is updated (*l.2*). Since the (old) reference of the right child is transferred to the parent node, now the right child of the new node can be updated to be the parent node (*l.3*). This ensures that the new node goes down in the height of the tree. Afterwards, the parent of the parent node is accessed to become the new parent of our new node and the current node becomes the new parent of the

parent node (*l.4-5*). Lastly, the updated parent of the new node needs a new reference as a child. The updated parent of the new node still holds the old parent of the new node as a child node. Hence, the child of the updated parent needs to be updated to be the new node. This will ensure that the new parent of the rotated node has as a child the sub-tree with rotated positions (*l.7-13*). If the rotated node does not have parents, it has to be updated as the new root node of the tree (*l.14*). The left rotation is implemented in a similar manner as in Algorithm 2.

Algorithm 2 Rotate Right

```
1:  procedure RotateRight(currentNode, parentNode)
2:      Attach the right child of currentNode to the left child of parentNode
3:      Update the parent of the new left child of parentNode
4:      Attach parentNode as the right child of currentNode
5:      Update the parent of currentNode to the parent of parentNode
6:      Update the parent of parentNode to currentNode
7:      if currentNode's parent is not null then
8:          if currentNode's parent's right child is parentNode then
9:              Set currentNode as the right child of currentNode's parent
10:         else
11:             Set currentNode as the left child of currentNode's parent
12:         end if
13:     else
14:         Update rootNode to currentNode
15:     end if
16: end procedure
```

2.3 Suppression method

Deleting a node in a tree needs be to performed carefully, as it is not possible to just randomly delete a node since this could violate the fundamental properties of the Cartesian tree. A method to delete nodes is to rotate a given node z which needs to be deleted with its child which has the highest priority. In this way, the child is one level lower in the tree (closer to the root), and the node z moves up one level. This is in the opposite manner as the insertion, as the goal of the rotations was to move up a node. Here, the rotations are used inversely such that the node z can be rotated with its child with the highest priority. This is done iteratively until z is a leaf node after which the node is deleted. As the node z is rotated with its child that has the highest priority, the stack property of the Cartesian tree will still hold. Since the node z rotates with the child with the highest priority, this node will have a higher priority than the other sub-tree of z, and implicitly also its own sub-tree.

Algorithm 3 gives the general structure of the suppress function implemented in pseudo-code. First, the node that needs to be deleted is found in the tree through the *findNode* function that was described in Section 2.1. If the node is not found, the function will throw an exception (*l.1-5*). If the node in the tree is found, the function accesses its left and right child to start the rotations (*l.6*). The while loop is started and will continue until the node that needs to be deleted is a leaf, i.e. has no children. If the deleted node has both left- and right children, the priority values of both will be verified to understand which children node needs to be rotated with the node to be deleted. If the left child has higher priority, a right rotation will be performed between the left child and the node to be deleted as described in Algorithm 2 and vice versa if the right child has higher priority (*l.8-13*). If a node has only one child, e.g. right child, the node to be deleted will be left-rotated with its right child (*l.14-16*). After the rotations have been applied, the tree is updated and therefore the left- and right children of the node to be deleted are also updated. These latter two are variables used inside the function and updated

in each iteration (*l.19*). This process continues until the node to be deleted is a leaf node (no children). Lastly, at the moment all the rotations have been applied, the predecessor of the node is updated such that it does not contain the node to be deleted as a child anymore (*l.21-25*).

As analyzed before in Section 2.1, the complexity of finding a node is $O(k_1)$ if k_1 is the depth of the node. This node, however, needs to be rotated till the top of the tree. The number of levels until the top of the tree is reached is $l = h - k_1$, and all rotations have a constant running time of $O(1)$. So in other words it is first necessary to find the node to be deleted from the root which takes k_1 operations and after it takes l rotations to bring this node to the top of the tree. Hence, the total running time of the function will be $O(h)$ where h is the height of the tree.

Algorithm 3 Suppress Node

```
1:  procedure DeleteNode(Key, Priority)
2:      Find NodeDelete in the tree by Key
3:      if node is not found then
4:          Throw an exception: "Element not found"
5:      end if
6:      Access LeftChild and RightChild of NodeDelete
7:      while NodeDelete has at least one child do
8:          if both left and right children exist then
9:              if Priority of LeftChild < Priority of RightChild then
10:                 Right rotation LeftChild and NodeDelete
11:             else
12:                 Left rotation RightChild and NodeDelete
13:             end if
14:         else if LeftChild is null then
15:             Left rotation RightChild and NodeDelete
16:         else
17:             Right rotation LeftChild and NodeDelete
18:         end if
19:         Update LeftChild and RightChild after rotation
20:     end while
21:     if NodeDelete is the right child of its parent then
22:         Remove RightChild reference in parent node
23:     else
24:         Remove LeftChild reference in parent node
25:     end if
26: end procedure
```

3 Performance analysis of implementation with random priority generation

3.1 Set-up for performance testing

One of the main advantages of a Cartesian tree is that the specific structure can be used to attempt to keep the binary search tree balanced. By generating random priorities for every key, the goal is to create a balanced binary search tree through these generated priorities. Since the Cartesian tree needs to preserve the stack properties, rotations will be applied so as not to violate these properties. Problems with data structures such as Balanced Binary Search Trees include the fact that rotations need to be applied in the majority of insertion operations, making them computationally inefficient to create. Through having a Cartesian tree data structure

with randomly generated priorities, the aim is to create an approximately balanced tree in a probabilistic manner. In this section, a structured analysis will be given to understand if this goal is indeed reached with the implemented data structure in Java.

Consequently, To give the performance analysis of the Cartesian tree random priorities will be generated for the inserted nodes of the tree. The random priorities for each node are generated from a uniform distribution, which implies that the priorities are allocated in a balanced manner. As discussed before in the report, the best method to implement a Cartesian tree is to keep the priority values unique. If the keys and priorities are unique, it is ensured that there is a unique solution to the tree. In this report's used implementation, a random priority $p \in N$ is generated through the following formula:

$$p_{col} = \lfloor 100 \cdot n^2 \cdot U(0,1) \rfloor \tag{1}$$

Where n is the number of nodes going to be inserted in the Cartesian tree and $U(0,1)$ is a random number drawn from the uniform distribution. Subsequently, the statistical foundation behind this formula will be given. The range of generated priority is in $[0, 100n^2)$ and thus the probability of having a collision between one pair is given in Formula 2.

$$P(\text{collision}) = \frac{1}{100n^2} \tag{2}$$

This event of getting a collision can also be seen as a random variable X with Bernoulli distribution where $p_{collission}$ is the probability of getting a collision. Subsequently, as we acquired the Bernoulli distribution for the collision of one pair, this can be extended to a Binomial distribution with all the node pairs of the tree. For a tree with n nodes, there are $\binom{n}{2}$ combinations of pairs such that the event of having a collision in priority is the random variable $Y \sim \text{Binom}(\binom{n}{2}, p_{collission})$. Hence the probability of having at least one collision is equal to:

$$P(\text{at least one collision}) = 1 - P(Y = 0) = 1 - (1 - \frac{1}{100n^2})^{\binom{n}{2}} \approx 1 - \exp\left(-\frac{n(n-1)}{2} \cdot \frac{1}{100n^2}\right). \tag{3}$$

The approximation used for the binomial distribution comes from the same analysis used in the clash of birthdays in the famous birthday paradox. By using the squared term in the priority generation of Formula 1 the probability of getting a collision stays small when n reaches infinity. As n grows larger the following approximation can be used for Formula 3. At the moment the term in the exponent is very small, it can be rewritten to the final formula given in Formula 4. Hence when n becomes large the probability of getting a collision with the priorities generated through Formula 1 tends to 0.005. This will ensure that the tree will be a unique solution with a very high probability.

$$P(\text{at least one collision}) \approx 1 - \exp\left(-\frac{1}{200}\right) \approx 1 - \left(1 - \frac{1}{200}\right) = \frac{1}{200} \tag{4}$$

The importance to not have the collision of priority values and a unique solution to the Cartesian tree is to ensure its overall efficiency. If priority values would collide, then every time a new node would be inserted there would be significantly more rotations as the algorithm tries to rotate with respect to priority values. When a new node is inserted, the algorithm would think that the two nodes with the same priorities are not ordered correctly. This would mean that more rotations will be applied only to ensure (wrongfully) that the priority property is not values. This also applies to other operations such as suppression, when the priority values are not unique. Hence, to keep the operations of the Cartesian tree efficient it is necessary to ensure the uniqueness of the priority values through Formula 1. Lastly, it is important to note that the data type used for the priority is of type *Long*. An *Integer* type in Java cannot handle priority values of very large sizes at the moment when p_col becomes too large. If the *Long* type

is not used as the the data type of the priority values, the incorrect values will be assigned to the variable as it cannot handle the large number.

As it is not very logical for larger-sized trees to use character values (such as demonstrated in the previous sections) as the keys, the $CartesianTree$, and $Node$ classes were extended to be able to take any data type as a key value, and priority value. The keys that are generated are integers and the values of the keys are all from 0 to n. It is thus ensured that the keys have unique values. After the list has been generated the $Collections.shuffle$ function is used with a seed of 54 (randomly selected) to shuffle this list. The list of priorities and list of keys are generated for every iteration after which the corresponding nodes are inserted in the tree. This has been implemented for different sizes and for each size 100 iterations have been tested and saved to get an average performance of each size. The statistics and graphs will be given in the subsequent parts.

3.2 Results performance

To understand the performance of the implemented Cartesian tree structure two main metrics are important to analyze. Firstly, the height of the Cartesian tree needs to be analyzed as the goal of the data structure is to get a balanced tree in a probabilistic manner. The average height is thus expected to be around the order of $log(n)$, with the logarithm being to the base of 2. This is indeed slightly different than a balanced Binary Search Tree (BST), where the balance of the tree is ensured. Furthermore, another metric that is important to examine is the processing time of inserting all the nodes for the Cartesian tree. As discussed in previous sections, one of the main goals of the Cartesian tree is to have low processing times. As the Cartesian tree is expected to be balanced on average, the processing time of inserting a key would be expected to be of order $O(log(n))$ on average. This is due to the properties of inserting a key in a BST. The rotations due to priority violations are considered to be executed in constant time implicating they can be neglected in the complexity of inserting nodes in the tree. As n nodes are inserted for a Cartesian Tree of size n, the average processing time for inserting n nodes is expected to be $O(nlog(n))$. Hence, these are the important metrics to understand if the implementation of the Cartesian Tree is efficient.

Table 1

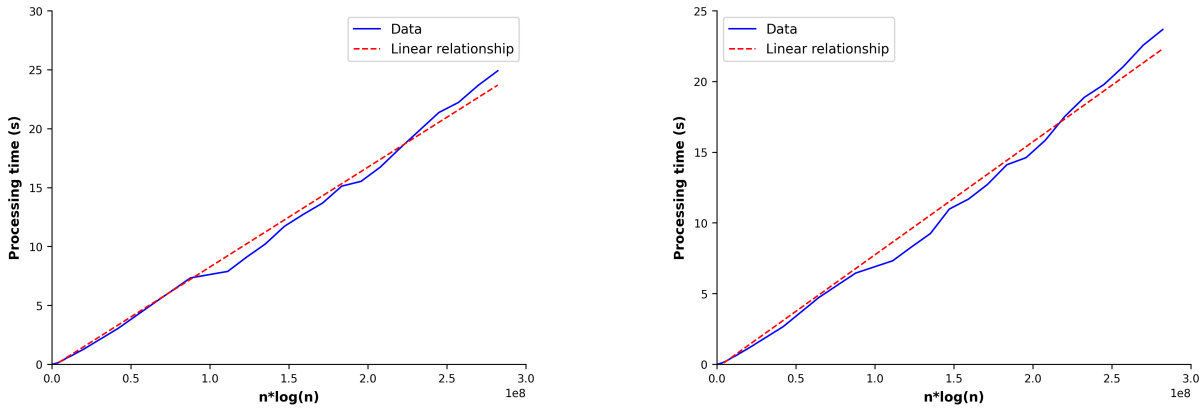
Descriptive statistics performance metrics

Size n	Avg. height	Avg. $t_{insertion}$	Var. $t_{insertion}$	Avg. $t_{supression}$	Var. $t_{supression}$	Iterations
1,000	21.02	2.7e-4	2.3e-6	1.7e-4	1.0e-6	100
10,000	30.44	2.9e-3	3.8e-5	1.5e-3	2.1e-5	100
100,000	39.61	0.05	7.3e-4	0.04	3.7e-4	100
500,000	45.77	0.55	1.9e-2	0.47	1.2e-2	100
2,000,000	52.27	3.04	0.11	2.65	0.14	100
5,000,000	55.75	7.88	0.14	7.32	0.35	100
10,000,000	58,62	22.21	0.41	21.06	0.60	100

First in Table 1, some descriptive statistics of the metrics are given for different tree sizes n. It seems that the average height does not increase very fast with the increasing number of nodes. The same can be seen with the mean processing time of inserting the nodes for a tree of size n. This processing time does also not appear to have very high increases with the bigger tree sizes. Furthermore, the variance of the processing time seems to increase with the size of the tree. The average insertion times seem to be slightly higher than the suppression times, this also seems to hold for the variance. As the tree becomes bigger the statistics show that the processing times will vary more. A possible method to keep the variance of the processing time lower for the bigger trees is to increase the number of iterations. These statistics seem to show

the first indications that the implementation of the Cartesian tree does what it expects. A more extensive analysis will now be given with figures.

In Figure 4, the average processing time (in seconds) of inserting and deleting all the nodes is plotted against the expected complexity of inserting the nodes for a tree of size n. The average processing time for each tree size n is taken from the 100 measured iterations. This figure shows a clear linear relationship between the processing time of the functions of the Cartesian tree and the expected complexity. In red, the linear fit of the data is given to understand if there is indeed a linear relationship. Remarkably, it does seem that the tree implementation is faster than expected with tree sizes smaller than $9e6$. However, past this point, the data trend seems to have a linear relationship with the steeper slope. The relationship seems linear overall, it should be inspected further why the increase in time becomes larger with tree sizes. The larger tree size could have slightly more variance in their structure implying more variance in the processing time. Figure 4 does show that on average, the expected complexity of processing time for inserting n nodes is of order $O(nlog(n))$ in the implemented Cartesian Tree data structure for this report.



(a) Insertion **(b)** Suppression

Figure 4: Processing time of functions against expected processing time complexity.

Moreover, Figure 5 shows us the relationship between the average height of the tree in the recorded results and the expected height of the tree $log(n)$. Here, it is expected for the tree to be balanced on average. The growth of the average height needs to be the same as the growth of $log(n)$ (linear relationship). The figure indeed shows us that there is an almost perfect linear relationship between the measured average height of the tree and the expected height. Due to the randomness of the Cartesian tree, it is normal for the relationship not to be exactly linear ($h = log(n)$) but there could be some scaling factor in this relationship due to randomness. The crucial part is for the relationship to be linear and have the same growth rate on average. This can be seen through the almost perfect straight line in Figure 5.

The structure of a Cartesian tree and the fact that the height of the tree is balanced on average make it an efficient data structure. For a Binary Search Tree (BST), the balance is not ensured deterministically or probabilistically. Hence, if the nodes in a BST are inserted in an inefficient way a tree could become just a type of linked list. As an example we use again the keys to be character values of the alphabet, inserting the nodes in alphabetical order in ascending or descending way would create a structure that is essentially the same as a linked list. In Figure 6 on the left side, it can be seen how inserting the nodes in alphabetical order with their key values can lead to an unbalanced tree for a BST (on the left side the priority values are not relevant). The Cartesian tree balances this tree through its random priority assignation. For a Cartesian tree, the priorities and the stack properties need to hold. Consequently, even if the keys are inserted in alphabetical order, the data structure will need to rotate the nodes to keep

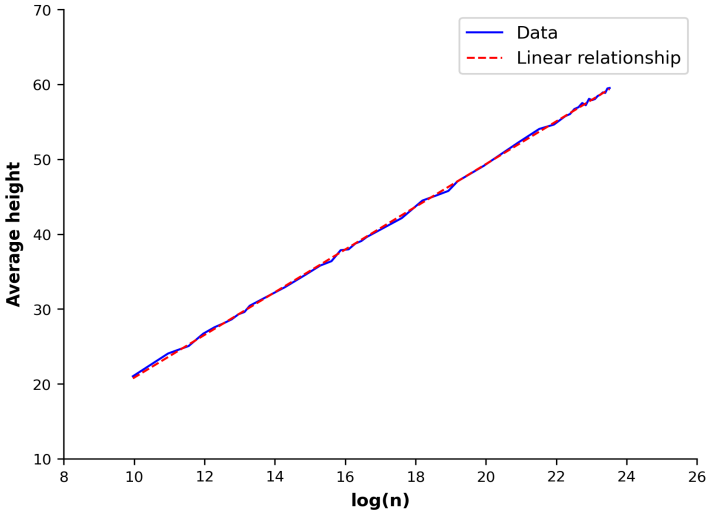


Figure 5: Relationship between average height of tree and logarithm of size n.

the stack properties. This leads to a balanced tree on average and its result for the example can be seen in Figure 6.

Overall, it can be concluded that the applied data structure works approximately as expected in the Java implementation in this report. The processing time of inserting and deleting the nodes for a tree of size n seems to be linear with $O(nlog(n))$. This is the expected result since the complexity of inserting or deleting one node in a Cartesian tree is of complexity $O(log(n))$. Furthermore, the height h seems to be on average of the same order as $log(n)$. This also would be expected due to the balancing of the tree such that the priority properties still hold. For the height, it is expected to grow as fast and thus linearly with $O(log(n))$, and this can also be seen in Figure 5.

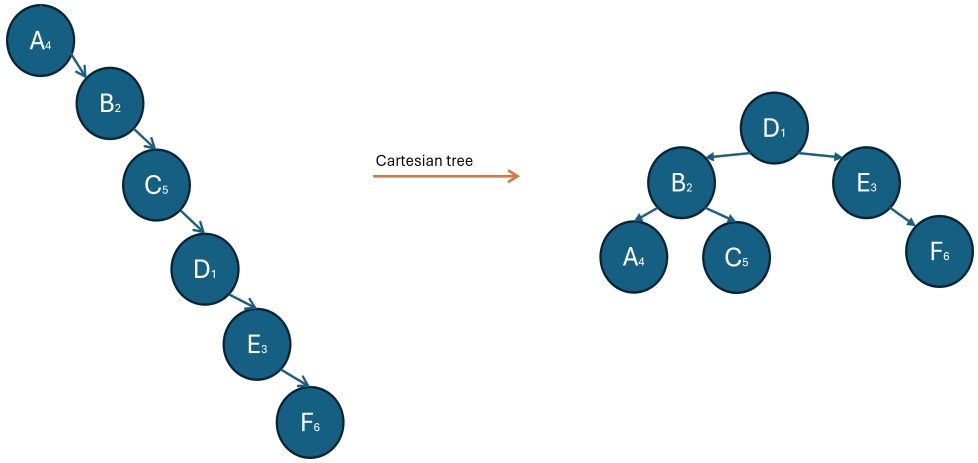


Figure 6: Inefficiency of BST when inserting nodes in alphabetical order.

4 Theoretical analysis average height of tree

To finalize this report, the probabilistic formula of the average height of a Cartesian tree will be analyzed to see if it is indeed $O(log(n))$. First, the following variables are defined:

- x_k is the node in the tree where the key is the k-th smallest.

- X_{ij} is the random variable taking 1 when x_i is a proper ancestor of x_k.
- $X(i,k)$ is the set of nodes $\{x_i, x_{i+1}, ..., x_k\}$ or $\{x_k, x_{k+1}, ..., x_i\}$ for $i < k$ and $k < i$ respectively.

Hence, with the following definitions it can be understood that the height of a tree at node k is also equal to Formula 5. This is because the height of a node k is also equal to the amount of ancestor the nodes has, which is exactly the sum of the defined random variable for node k.

$$\text{height of the tree at } k = \sum_{i=1}^{n} X_{ik} \tag{5}$$

Through formula 5, the average height of a tree at any node k is the expected value of this sum. Furthermore, since we have the linearity property of expectations the sum can be taken out of the expected value. This gives us the following derivation:

$$\text{average height of the tree at } k = E(\sum_{i=1}^{n} X_{ik}) = \sum_{i=1}^{n} E(X_{ik}). \tag{6}$$

Now that the formula is given to calculate the average height, another proof is undertaken which is necessary to acquire the complexity of the average height. The statement is given in the following formula:

$$X_{ik} = 1 \iff x_i \text{ has the lowest priority in } X(i,k) \tag{7}$$

If the hypothesis is that $X_{ik} = 1$ ($\implies$), this means that node i is a proper ancestor of k. For i to be an ancestor of k, the latter node needs to be in the sub-tree of the former one. By contradiction, if i is not the smallest priority in $X(i,k)$ there will be the node j with the lowest priority as the root of the sub-tree $X(i,k)$. If $i < j < k$ or $k < j < i$, the nodes i and k will be in different sub-trees of node j with the lowest priority. This contradicts the hypothesis that $X_{ik} = 1$. Hence if $X_{ik} = 1$, this implies that x_i has the lowest priority and needs to be the root of the sub-trees in $X(i,k)$.

If the hypothesis is that x_i has the lowest priority in $X(i,k)$ ($\impliedby$), by the properties of a Cartesian tree this means that x_i needs to be the ancestor of all nodes in $X(i,k)$. By definition of $X(i,k)$, if $i < k$ all other nodes would be on the right sub-tree and on the left sub-tree if $k < i$. If this is not the case the stack properties of the Cartesian tree would be violated. Hence, by definition, if x_i has the lowest priority in $X(i,k)$, x_i is an ancestor of all nodes in $X(i,k)$ and thus also x_k ($X_{ik} = 1$). Otherwise, the properties of the cartesian tree would be violated.

Using the Formula 6 and the equivalency in 7 the formulas can be rewritten as:

$$\sum_{i=1}^{n} E(X_{ik}) = \sum_{i=1}^{n} P(X_{ik} = 1) = \sum_{i=1}^{n} P(x_i \text{ has smallest priority in } X(i,k)). \tag{8}$$

As the priorities are generated through a uniform random distribution there is an equal chance for every node to have the priority hence the formula is rewritten again. Applying a variable change $j = k - i$ when $i < k$ and $j = i - k$ when $k < i$ to the sum in Formula 9 gives us the final sum which is smaller than the harmonic sum H_n. Lastly, by combining Formula 6, Formula 8, and formula 9 the final conclusion of the complexity of the average height of the Cartesian tree can be derived to be $O(log(n))$.

$$\sum_{i=1}^{n} P(x_i \text{ has smallest priority in } X(i,k)) = \sum_{i=1}^{n} \frac{1}{|k-i|+1}$$
$$= 1 + \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1}$$
$$= 1 + \sum_{j=1}^{k-1} \frac{1}{j+1} + \sum_{j=1}^{n-k} \frac{1}{j+1}$$
$$\leq 1 + 2 \cdot \sum_{j=1}^{n} \frac{1}{j}$$
$$= 1 + 2 \cdot H_n = 1 + 2 \cdot \ln(n) = O(\log(n)). \quad (9)$$

5 Conclusion and Discussion

Overall, the framework of the Cartesian tree implementation in Java has been discussed in this report. First, the general definition of a Cartesian tree was given and some examples were demonstrated. In Section 2, all the details and architecture choices of the classes for the Cartesian tree were given. The time complexity and challenges of the insertion and suppression function were also discussed in detail. Furthermore, the pseudo-code and general explanation of the methods for insertion and suppression were given in this Section. After the implementation choices were justified, a performance analysis was made on the data structure through random Cartesian trees. The benefit of these random Cartesian trees is the fact that through generating the random priorities it can be possible to keep the tree approximately balanced. As it is generally known, a balanced binary search tree has a lot of benefits in search complexity. In the results tree sizes up until 12,000,000 nodes were tested, where 100 iterations were saved for each tree size. The result showed that on average the height of the Cartesian trees is linear with complexity $O(log(n))$. This is indeed the expected and hoped-for result for the data structure. Furthermore, the complexity of the insertion and suppression algorithms was also examined and gave the expected results. The methods were on average linear with the complexity $O(nlog(n))$, when using the methods for tree sizes of size n. A less expected result was the more-than-expected increase in running time for tree sizes over 9,000,000 nodes. This made the linear relation slightly less accurate than hoped. One possible explanation could be the variance of larger trees are higher due to their bigger structure. Subsequently, it gives an investigating point for further research. Lastly, in the last section, the theoretical analysis was given for the probabilistic Cartesian tree and its average height to be of complexity $O(log(n))$.