Cartesian Tree - Algorithms Laboratory

Giuliano Lazzara

April 2021

Index

1 Introduction

Among the overview of data structure to manage and organise information, there is a particular model, based on the properties of an ordered heap and in the meantime in a symmetric (in-order) traversal of the tree, that returns the original sequence. This is the Cartesian tree, a binary tree derived from a sequence of numbers that in 1980, Jean Vuillemin introduced it in the context of geometric range searching data structures. This data structure could have several applications,in fact it could be used:
-in the definition of the treap,
-in the definition of the randomized binary search tree data structures for binary search problems,
-in the solving for sorting problems [1].

In this report will be discussed some functions usable in this data structure, their implementation and the complexity to implement this algorithm.

2 Cartesian Tree

Given a sequence of numbers, it is possible to define a Cartesian tree declaring its two main properties:

-An in-order traversal of the tree has to return exactly the original sequence of numbers.

-The Cartesian tree has to respect the **heap property**; the children have always to be higher than their parent, so the smallest number has to be the root of the tree.[2]
For a given array A of n elements, the relation between a node and its parent is:

$$A[i] \geq A[PARENT(i)].$$

(for this report has been chosen the property of the **min-heap**; the codes and all the documentations has been based on it).

It could be very unbalanced, for example if it is considered a sequence of numbers sorted in order of increasing, the built Cartesian tree on this sequence will be a tree where every element will be the parent of the next element of the sequence. Otherwise in the case of a sequence of numbers sorted in order of decreasing, the situation will be the same, with the last element of the sequence as root. It is important to notice how the property of the in-order traversal equal to the starting sequence is always respected, indeed, in these two cases showed the first element visited for the traversal is always the first element of the sequence; the former is the root and later is the deepest leaf of the tree.

Example of Cartesian Tree, given a sequence of 12 numbers

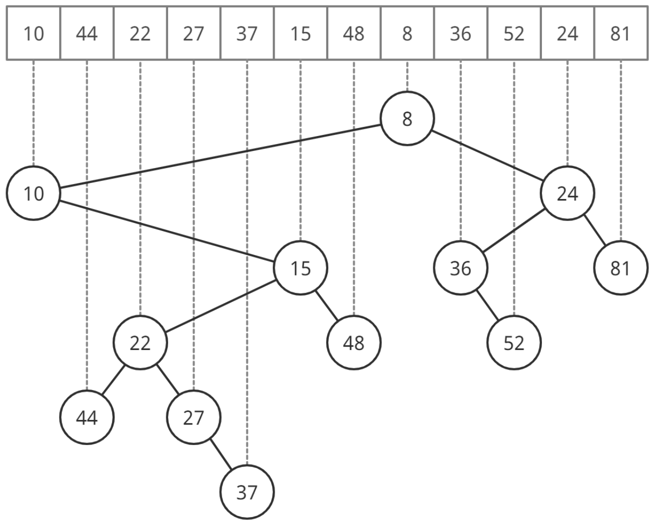


Figure 1

In the example showed in Figure 1, the smallest value (that in this case is the 8) is the root of the tree. Another interesting point visible from this example is that the direct children of the root are the smallest numbers that are possible to find in the two sub-sequence born after that the smallest number has became the root of the tree. In fact it is possible to see the creation of a Cartesian Tree as follows:

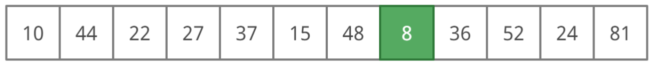


Figure 2

In the initial sequence, it can be noticed to see that the smallest value is the 8, so this is chosen and the sequence is divided in two sub-sequence.

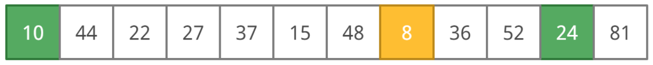


Figure 3

Each smallest element of both sub-sequences is chosen to become a child of the root, so at this point the root of the tree is the number 8, its left child is the number 10 and its right child is the number 24.

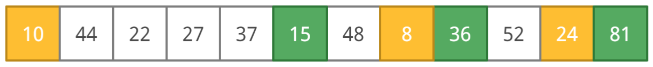


Figure 4

At this point the number 10, son of the number 8, has just a right child, that is the number 15. Despite the other 8's child, the number 24 that has both children, the numbers 36 and 81. This process goes recursively until all the elements of the sequence are chosen.

3 Implemented functions

In the next section will be showed the implementation of the main functions to build a Cartesian Tree, but in this section, it is going to be an explanation of other functions applied to this particular project, starting from a little description of the UML.

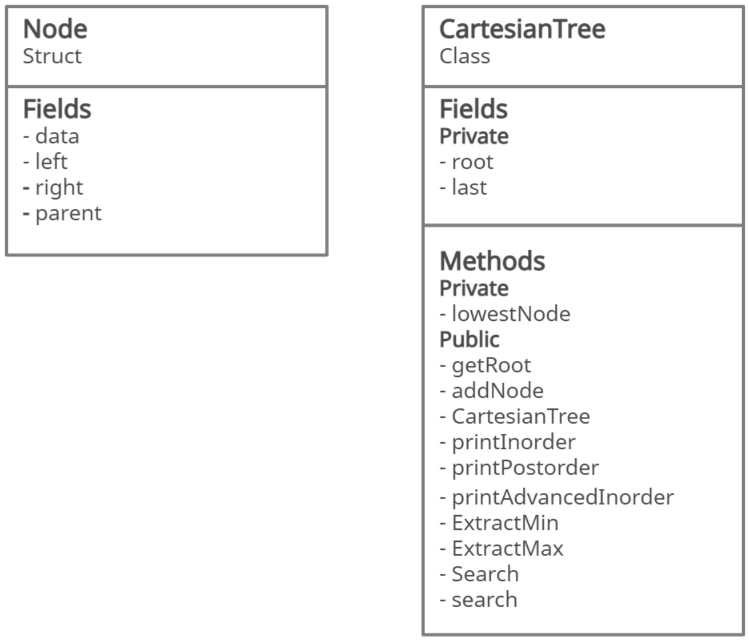


Figure 5

It has been decided to implement a Struct **`Node`** that in this case works just with integers, but obviously it could be extended also to other kind of data.
This is a basic implementation to show the potentially of this algorithm and its main applications. In the next section, the relation between the Struct **`Node`** and the Class **`CartesianTree`** will be showed more detailed, because this section is going to focus on the functions of the Class **`CartesianTree`** and for this reason, the functions LOWEST-NODE and GET-ROOT will be discussed in the next section.
The first function that it is important to explain is the function ADD-NODE: This function is very important for the building of a Cartesian Tree, but it has been implemented to add a node also to an existing tree.
For the best case, i.e., when the tree is empty, it is possible to approximate the complexity of ADD-NODE to $O(1)$, because it just checks if the **`root`** is NIL and in this case it sets the values **`last`** and **`root`** with the value that the function is going to add in the tree.

ADD-NODE

```
void addNode(int x) {
    Node *new_node = new Node;
    new_node->data = x;
    if(root == NULL)    {
        last = new_node;
        root = new_node;
        return;
    }
    Node *aux = new Node;
    aux = lowestNode(last, x);
    if(aux == NULL)   {
        new_node->left = root;
        root->parent = new_node;
        root = new_node;
    }
    else    {
        new_node->parent = aux;
        new_node->left = aux->right;
        if(aux->right!=NULL)
          aux->right->parent = new_node;
        aux->right = new_node;
    }
    last = new_node;
}
```

[3]*(inspiration source of the code)*

In the described above other cases it is used the function LOWEST-NODE, that basically checks if the last node exists and, in this case, if it is bigger or smaller than the value x that is going to join the tree. In fact the other two cases of this function rely on the case where the last is NIL and in the case where it isn't. At the end of this procedure, the added node is set as **`last`**.

The functions PRINT-IN-ORDER and PRINT-POST-ORDER are just two print functions for the traversal of the tree. The function PRINT-POST-ORDER is a kind of verify to see the correctness of the algorithm; because one of the properties of the Cartesian tree is that the in-order traversal of the tree is equal to the sequence of numbers used to build the tree, a function that does the post-order traversal is a good way to see, from a different point of view, if the built tree is correct or not.

Here the implementation of these two functions.

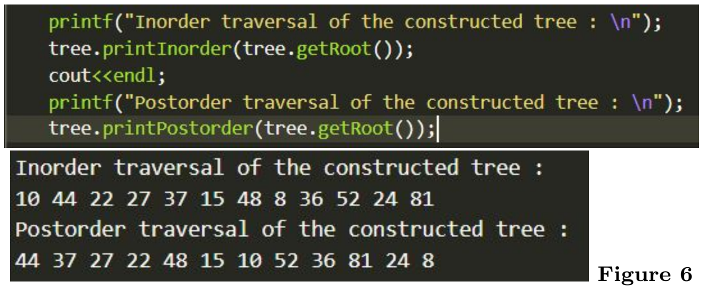
PRINT-IN-ORDER

```
void printInorder (Node* node) {
     if (node == NULL)
         return;
     printInorder (node->left);
     cout << node->data<<" ";
     printInorder (node->right);
}
```

PRINT-POST-ORDER

```
void printPostorder(Node* node)    {
     if (node == NULL) return;
     printPostorder(node->left);
     printPostorder(node->right);
     cout << node->data << " ";
}
```

Example using the tree built in the **Figure 1**:

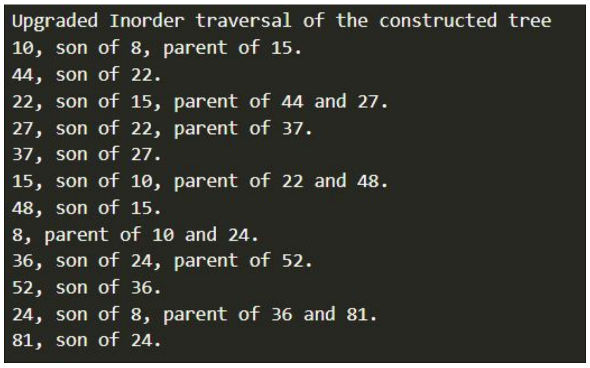


```
printf("Inorder traversal of the constructed tree : \n");
tree.printInorder(tree.getRoot());
cout<<endl;
printf("Postorder traversal of the constructed tree : \n");
tree.printPostorder(tree.getRoot());
```

```
Inorder traversal of the constructed tree :
10 44 22 27 37 15 48 8 36 52 24 81
Postorder traversal of the constructed tree :
44 37 27 22 48 15 10 52 36 81 24 8
```

Figure 6

The function PRINT-IN-ORDER has been updated with the function PRINT-ADVANCED-IN-ORDER. This function prints the in order traversal node by node printing also the relations that these node can have or not. This means that if a node x has a child y, the function will print: "x, *parent of* y"



```
Upgraded Inorder traversal of the constructed tree
10, son of 8, parent of 15.
44, son of 22.
22, son of 15, parent of 44 and 27.
27, son of 22, parent of 37.
37, son of 27.
15, son of 10, parent of 22 and 48.
48, son of 15.
8, parent of 10 and 24.
36, son of 24, parent of 52.
52, son of 36.
24, son of 8, parent of 36 and 81.
81, son of 24.
```

Figure 7

The functions EXTRACT-MAX and EXTRACT-MIN are very similar and both are based on the logic of the algorithm **Heap-Sort**.
In the first case, the maximum (in the second the minimum) is searched inside the tree, extracted from it, and then put in an array that will be printed at each step. In this way a decreasing sequence of numbers, which goes from the highest to the lowest value of the tree, is obtained from the function. Instead in the second case is obtained an increasing sequence of numbers that goes from the lowest to the highest value of the tree.
In this case these two implementations are very similar.

EXTRACT-MAX

```
void ExtractMax(int arr[], int n)    {
    int aux[n];
    for(int i=0; i<n; i++)
        aux[i] = arr[i];
    int MaxArr[n];
    int count = n;
    int m = 0;
    while(n>0)  {
        int max = -1000; #set value
        int k = 0;
        for(int j=0; j<n; j++)  {
            if(aux[j]>max)  {
                max = aux[j];
                k = j;
            }
        }

        MaxArr[m] = max;
        m++;
        for(int l=k; l<n-1; l++)
            aux[l] = aux[l+1];

        CartesianTree tree = CartesianTree(arr, n-1);
        cout<<max<<" removed, remining nodes: ";
        tree.printInorder(tree.getRoot());
        cout<<endl;
        n--;
    }
    cout<<"Sorted array: ";
    for(int i=0; i<count; i++)
            cout<<MaxArr[i]<<" ";
    cout<<endl;
}
```

In both functions there are implemented also two lines of code that give back how many nodes left there are in the tree.

EXTRACT-MIN

```
void ExtractMin(int arr[], int n)    {
    int aux[n];
    for(int i=0; i<n; i++)
        aux[i] = arr[i];
    int minArr[n];
    int count = n;
    int m = 0;
    while(n>0)  {
        int min = 10000;
        int k = 0;
        for(int j=0; j<n; j++)  {
            if(aux[j]<min)  {
                min = aux[j];
                k = j;
            }
        }

        minArr[m] = min;
        m++;
        for(int l=k; l<n-1; l++)
            aux[l] = aux[l+1];

        CartesianTree tree = CartesianTree(arr, n-1);
        cout<<min<<" removed, remining nodes: ";
        tree.printInorder(tree.getRoot());
        cout<<endl;
        n--;
    }
    cout<<"Sorted array: ";
    for(int i=0; i<count; i++)
            cout<<minArr[i]<<" ";
    cout<<endl;
}
```

The function SEARCH (divided in **`Search`** that is the "main" function, and **`search`** that does the recursive calls) has been implemented again in a way to verify the correctness of the algorithm, in fact it doesn't just print the searched node, but also its potential parent and its potential children.

SEARCH(*int x*)

```
Node* Search(int x)  {
    Node* current = getRoot();

    if(search(current, x))
        return current;
    else    {
        cout<<"Node not found!";
        return NULL;
    }
}
```

SEARCH($Node^*$, *int x*)

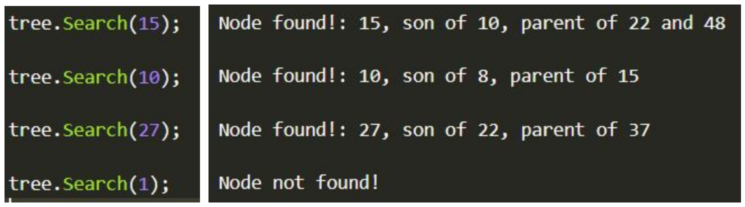
```
bool search(Node* current, int x)   {
    if(current==NULL)
        return false;

    if(current->data==x)    {
        cout<<"Node found!: "<<current->data;
        if(current->parent!=NULL)
            cout<<", son of "<<current->parent->data;
        if(current->left!=NULL || current->right!=NULL) {
            cout<<", parent of ";
            if(current->left!=NULL)
                cout<<current->left->data;
            if(current->left!=NULL && current->right!=NULL)
                cout<<" and ";
            if(current->right!=NULL)
                cout<<current->right->data;
        }
        return true;
    }
    bool sinistra = search(current->left, x);
    if(sinistra)
        return true;

    bool destra = search(current->right, x);
        return destra;
}
```

This function has complexity $O(n)$ in the worst case, that is when the wanted node is the most right leaf. In the average case it is $O(log\ n)$.

Example using the tree built in the **Figure 1**:



```
tree.Search(15);
tree.Search(10);
tree.Search(27);
tree.Search(1);
```

```
Node found!: 15, son of 10, parent of 22 and 48
Node found!: 10, son of 8, parent of 15
Node found!: 27, son of 22, parent of 37
Node not found!
```

Figure 7

4 Implementation

About the creation of Cartesian Trees there are four crucial functions: LOWEST-NODE, GET-ROOT, ADD-NODE (That has been discussed in the previous section) and CARTESIAN-TREE.
LOWEST-NODE is a recursive function to look for the lowest node inside the tree. It's used by the function ADD-NODE that gives, calling it, in input the values **`last`** and x (that is a node that in this time, is inserted in the tree).

LOWEST-NODE($Node$ $*node$, $int\ x$)

```
Node * lowestNode(Node *node, int x)    {
        if(node == NULL)
            return NULL;
        if(node->data < x)
            return node;
        else if(node->parent != NULL)
            return lowestNode(node->parent, x);
        else
            return NULL;
```

Therefore, it's possible to see from the code above, in which way this function moves going back to the parent of the CARTESIAN-TREE(). With this strategy, recursively, the function goes from the value **`last`** to the **`root`** of the tree.

GET-ROOT is just a function that gives back the value **`root`**.
GET-ROOT()

```
Node * getRoot()    {
        return root;
    }
```

GET-ROOT is used just to set a value, in fact it's used by the functions of search and print.

The last function that is going to be analyzed is the CARTESIAN-TREE. This function isn't particularly complex because is based on the structures of the other functions. Basically it sets the values **`last`** and **`root`** and then does the function ADD-NODE n times, where n is the size of the array taken in input to build the Cartesian Tree.
CARTESIAN-TREE(*int arr*[], *int n*)

```
CartesianTree(int arr[], int n)    {
        root = NULL;
        last = NULL;
        for(int i=0; i<n; i++)
            addNode(arr[i]);
    }
```

5 Complexity

About the complexity of this algorithm, it is necessary to study how the making process of this algorithm works. The function CARTESIAN-TREE builds the tree adding node by node, using in fact the function ADD-NODE. This function has to use the function LOWEST-NODE, to see which is the right position of the node that is going to join in the tree. This function has in the average case, complexity $O(log\ n)$ because the controls that it has to do undoubtedly, are less than n. So, considering that the function CARTESIAN-TREE builds a tree starting from a sequence of n numbers, the function ADD-NODE will be called n times. From all the facts considered, it is possible to conclude that the complexity of the function CARTESIAN-TREE is $O(n\ log\ n)$.

6 References

[1] - Wikipedia, "Cartesian tree",
URL: `https://en.wikipedia.org/wiki/Cartesian_tree`.
[2] - Authors: T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein
Year: 2009
Introduction to Algorithms (Third Edition), "Heapsort" (C. 6)
MIT Press
[3] - Prog.world, "Cartesian tree sorting",
URL: `https://prog.world/cartesian-tree-sorting/`.