Project Category- Sorting

Project Name/Purpose- **Cartesian Tree Sort** (To sort using a **Cartesian Tree data structure.**)

Prerequisites- http://www.geeksforgeeks.org/cartesian-tree/

Programming Paradigm Used- Sorting by utilising the fact that the Cartesian Tree is *already partially sorted.*

Data Structure Used- A Cartesian tree data structure , a priority queue.

Explanation of the algorithm-

This sorting algorithm sorts **cleverly,** that is it **utilises the fact that the array is already partially sorted when the input is stored in a Cartesian tree data structure**.

Before going further, let me ask you a question.

Suppose we have an input array which is partially sorted. Does traditional sorting algorithms like- Quick Sort and Heap Sort takes advantage of this fact and performs better than usual

Answer:- No.

In fact, there are very few sorting algorithms that make use of this fact. Such algorithms are called as **Adaptive Sorting Algorithms.**

Unlike typical heap-sort, ***Cartesian Tree Sort*** *use this fact very cleverly and sorts way faster than other non-adaptive algorithms if the data is partially sorted.*

Explanation of the algorithm-

Let us consider an input sequence- {5, 10, 40, 30, 28}.

The input data is initially unsorted and we want to sort it.

The input data is partially sorted too as only one swap between "40" and "28" results in a completely sorted order.

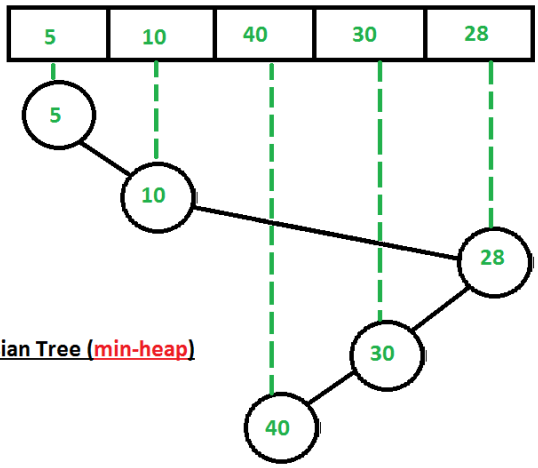
See how Cartesian Tree Sort will take advantage of this fact below.

The below steps will show how to sort the data according to **Cartesian Tree Sort Algorithm.**

Step 1:-

Build a **(min-heap)** Cartesian Tree [http://www.geeksforgeeks.org/cartesian-tree/] from the given input sequence. It is similar to building a **(max-heap)** Cartesian Tree, except the fact that now we will scan upward from the node's parent up to the root of the tree until a node is found whose value is **smaller** (and not **larger** as in the case of a **max-heap Cartesian Tree**) than the current one and then accordingly reconfigure links to build the min-heap Cartesian tree.

The min-heap Cartesian tree for the input sequence- {5, 10, 40, 30, 28} is shown below-



A sequence and its Cartesian Tree (min-heap)

Step 2:-

Starting from the **root** of the build min-heap Cartesian Tree, we push the nodes in a **priority queue.** Then we **pop the node at the top of the priority queue and push the children of the popped node in the priority queue in a pre-order manner** *(we are using a pre-order traversal technique because that gives a better performance as shown in the first figure of this article)* **as shown below:-**

1) Pop the node at the top of the priority queue and add it to a list.
2) Push left child of the popped node first (if present).
3) Push right child of the popped node next (if present).

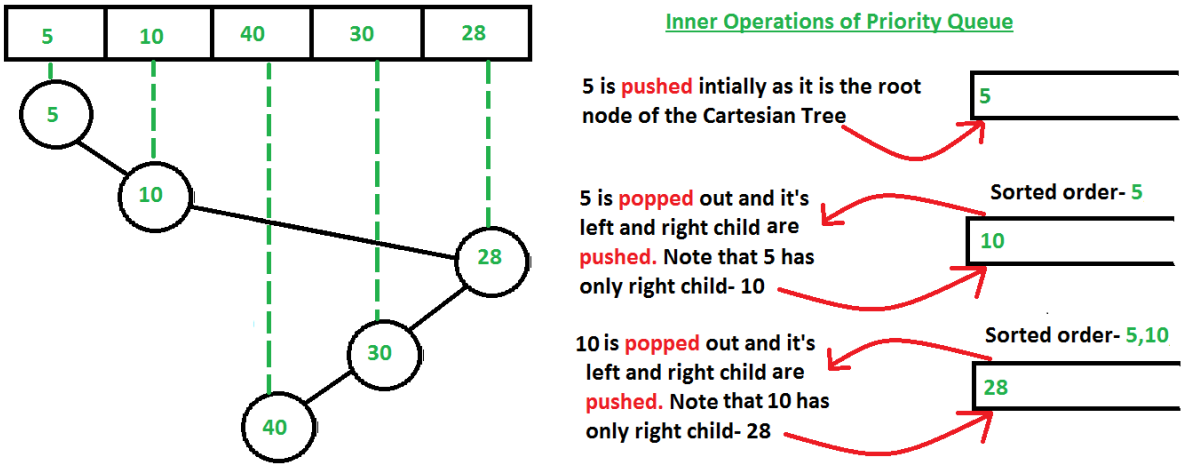
Step 3:-

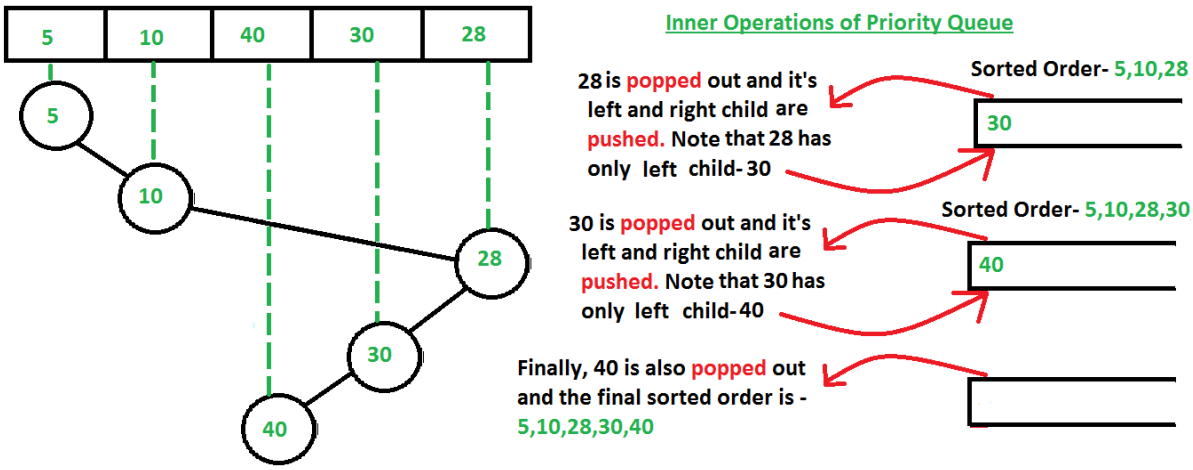
Repeat the ***Step 2*** till the priority queue is empty.

Step 4:-

The list contains the input data in a sorted manner.

The steps are clearly shown below for the input sequence- {5, 10, 40, 30, 28}





You might wonder that using priority queue would anyway result in a sorted data if we simply insert the numbers of the input array one by one in the priority queue (i.e- without constructing the Cartesian tree).

But the time taken differs a lot.

Suppose we take the input array – {5, 10, 40, 30, 28}

If we simply insert the input array numbers one by one (without using a Cartesian tree), then we

may have to waste **a lot of operations in adjusting the queue order everytime we insert the numbers** (just like a typical heap performs those operations when a new number is inserted, as priority queue is nothing but a heap).

Whereas, here we can see that using a Cartesian tree took only 5 operations (see the above two figures in which we are continuously pushing and popping the nodes of Cartesian tree), which is linear as there are 5 numbers in the input array also. So we see that the best case of Cartesian Tree sort is O(N), a thing where heap-sort will take much more number of operations, because it doesn't make advantage of the fact that the input data is partially sorted.

Hence, in a nutshell we can describe Cartesian Tree Sort as-

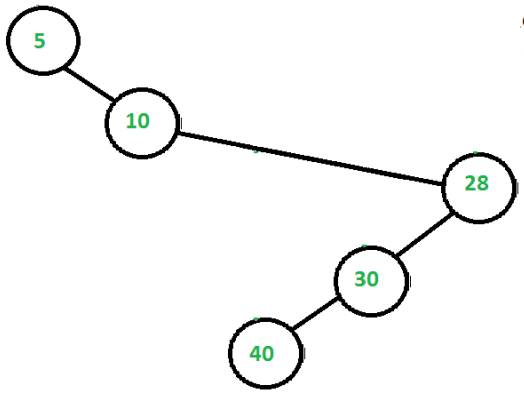
" Cartesian tree sort is at heart a form of selection sort, an algorithm that finds the elements of the output sequence in their sorted order, using data structures to speed up how it finds each element. It builds a Cartesian tree on the input, a tree in which the parent of each value is the larger of the two nearest smaller values on either side of it; this can be done in linear time. It maintains a binary heap of active values, initially just the root of the Cartesian tree. And then it repeatedly moves the smallest element of the heap to the output list, replacing it in the heap by its children in the Cartesian tree."

One may ask that why we should prefer *pre-order over in-order* while popping and pushing the data from the Cartesian Tree into the priority queue ?

The answer to this is that since Cartesian Tree is basically a heap- data structure and hence follows all the properties of a heap. Thus the root node is always smaller than both of its children. Hence an in-order fashion pushing-and-popping will not let Cartesian Tree Sort take advantage of the fact that the input data is partially sorted. Hence, we use a pre-order fashion popping-and-pushing as in this, **the root node is always pushed earlier than its children inside the priority queue and since the root node is always less than both its child, so we don't have to do extra operations inside the priority queue(the highlighted term "extra operations" refers to the restoring down of the heap when we insert a new element inside the priority queue which is basically a heap).**

Refer to the below figure for better understanding-

The figure is a typical min-heap Cartesian Tree



A heap holds partially sorted data, as a parent is smaller than each of its child, i.e-

5 < 10
10 < 28
28 < 30
30 < 40

Heap-sort doesn't utilises this fact, whereas a Cartesian Tree Sort uses this fact and smartly.
Notice that a pre-order traversal of the tree gives us exactly sorted data- 5, 10, 28, 30, 40.

Whereas, an in-order traversal gives- 5, 10, 28, 30, 40

This doesn't means that simply doing a pre-order traversal gives a sorted array every-time but certainly doing so will give lead to a better sorted data. To get exact results we will use a priority queue.

Time Complexity-

O(N) best-case behaviour(when the input data is partially sorted), **O(N log N) worst-case** behaviour (when the input data is not partially sorted)

Space Complexity-

We are using a priority queue and a Cartesian tree data structure. Now, at any moment of time the size of the priority queue doesn't exceeds the size of the input array, as we are constantly pushing and popping the nodes. Hence we are using **O(N)** auxiliary space.

References-

https://en.wikipedia.org/wiki/Adaptive_sort
http://11011110.livejournal.com/283412.html
http://gradbot.blogspot.in/2010/06/cartesian-tree-sort.html
http://www.keithschwarz.com/interesting/code/?dir=cartesian-tree-sort
https://en.wikipedia.org/wiki/Cartesian_tree#Application_in_sorting