**Cartesian Tree Sorting**

Prepared for

Dr. Arvind Dagur

Galgotias University

Prepared by : group-1

Radheshyam-21SCSE1011633

Neeraj -21SCSE1011675

Nitin -21SCSE1011671

December 02,2022

TABLE OF CONTENTS

INTRODUCTION TO THE STUDY

* 1. Introduction…………………………………………………………………………………………… 3
  2. Steps………………………………………………………………………………………………………. 4
  3. Approach……………………………………………………………………………………………….. 4
  4. Flowchart………………………………………………………………………………………………… 6
  5. Implementation……………………………………………………………………………………….. 7
  6. Output…………………………………………………………………………………………………… 9
  7. Complexity…………………………………………………………………………………………… 9
  8. References……………………………………………………………………………………………… 10
  9. End………………………………………………………………………………………………………… 10

**Cartesian Tree Sorting**

**Section I**

Introduction to the Study

Introduction

**Cartesian Sort** is an Adaptive Sorting as it sorts the data faster if data is partially sorted. In fact, there are very few sorting algorithms that make use of this fact. For example consider the array {5, 10, 40, 30, 28}. 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. Below are steps used for sorting.

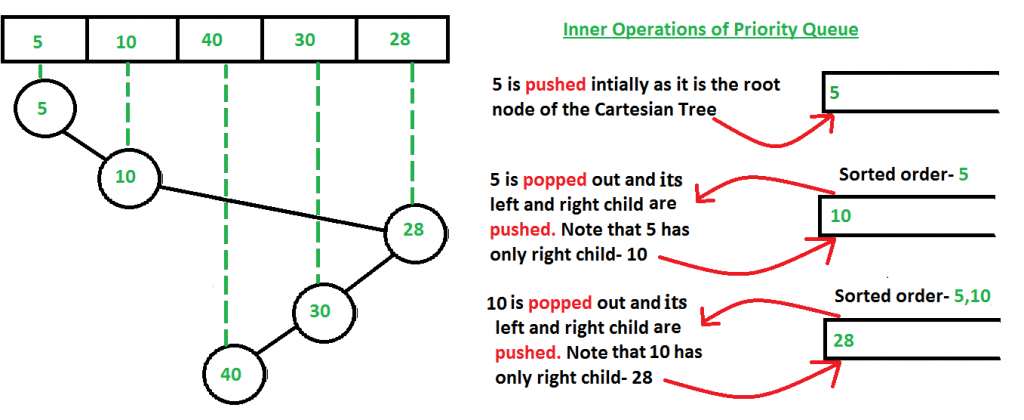
**Step 1 :**Build a (min-heap) Cartesian Tree from the given input sequence.

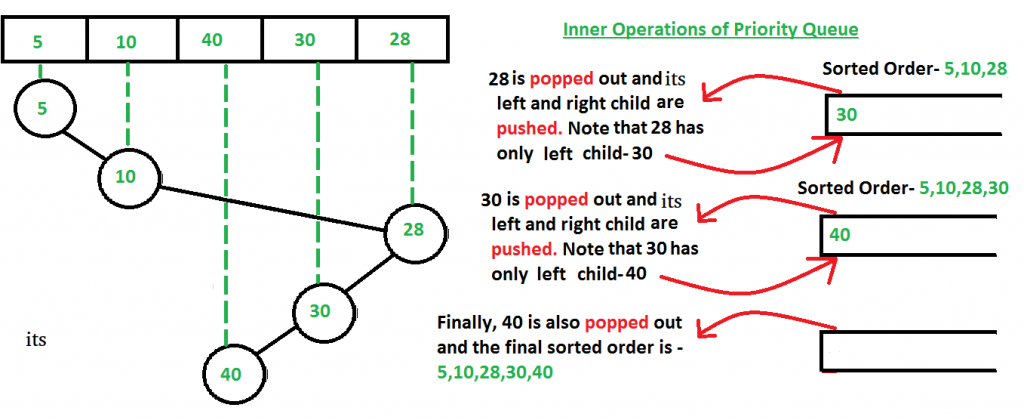
**Applications for Cartesian Tree Sorting:**

* It is possible to quantify how much faster algorithm will run than O(nlogn) using a measurement called **oscillation**. Practically, the complexity is close to O(nlog(Osc)) (where Osc is oscillation).
* Oscillation is small if the sequence is partially sorted, hence the algorithm performs faster with partially sorted sequences.

**Step 2 :**Starting from the root of the built 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.

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).





### ****How to build (min-heap) Cartesian Tree?****

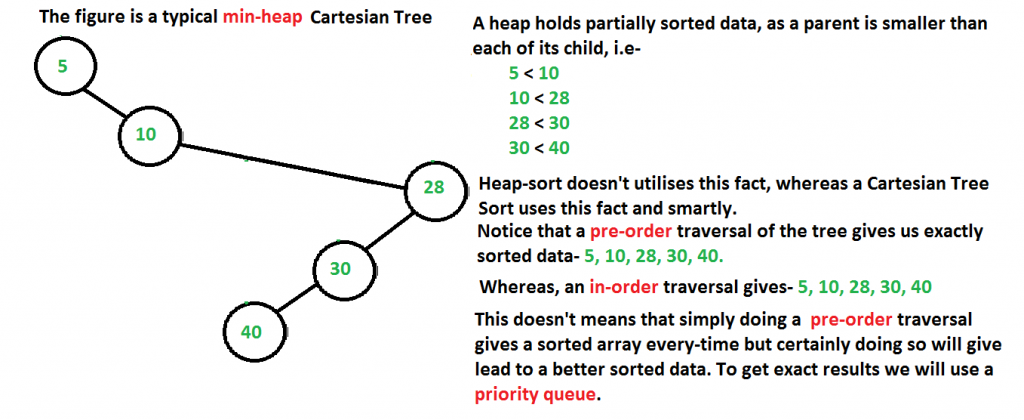
Building min-heap is similar to building a (max-heap) Cartesian, except the fact that now we scan upward from the node’s parent up to the root of the tree until a node is found whose value is smaller (and not larger as in the case of a max-heap Cartesian Tree) than the current one and then accordingly reconfigure links to build the min-heap Cartesian tree.

### ****Why not to use only priority queue?****

 One 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.

### ****Why preorder traversal?****

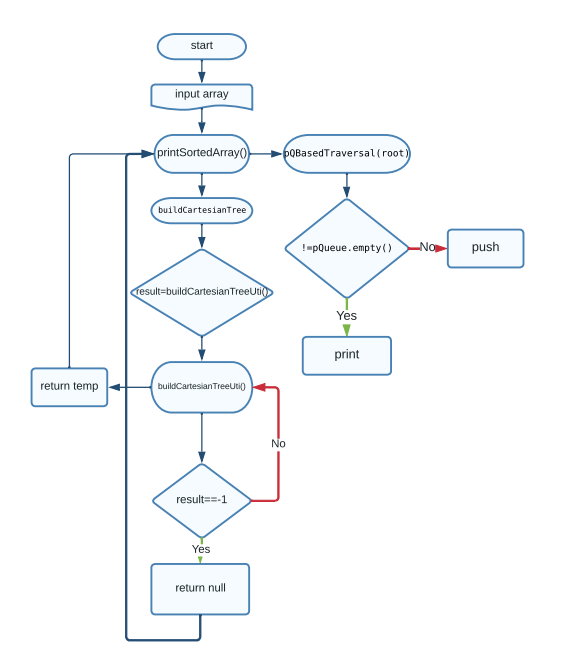
The answer to this is that since Cartesian Tree is basically a heap- data structure and hence follows all the properties of a heap. Thus the root node is always smaller than both of its children. Hence, we use a preorder fashion popping-and-pushing as in this, the root node is always pushed earlier than 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. Refer to the below figure for better



**Approach**:

* Construct a Cartesian tree for given sequence.
* Create a priority queue, at first having only the root of Cartesian tree.
* Pop the minimum value x from the priority queue
* Add x to the output sequence
* Push the left child of x from Cartesian tree into priority queue.
* Push the right child of x from Cartesian tree into priority queue.
* If priority queue is not empty, again goto step 3.

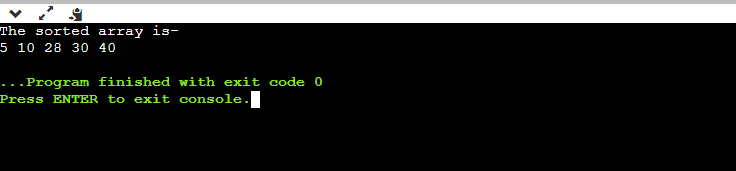
**Flowchart**



**Implementation**

|  |
| --- |
| #include<bits/stdc++.h>  using namespace std;  struct Node{  int data;  Node \*left, \*right;  };  typedef pair<int, Node\*> iNPair;  void pQBasedTraversal(Node\* root){    priority\_queue <iNPair, vector<iNPair>, greater<iNPair>> pQueue;  pQueue.push (make\_pair (root->data, root));    while (! pQueue.empty()){  iNPair popped\_pair = pQueue.top();  printf("%d ", popped\_pair.first);  pQueue.pop();  if (popped\_pair.second->left != NULL)  pQueue.push (make\_pair(popped\_pair.second->left->data,  popped\_pair.second->left));  if (popped\_pair.second->right != NULL)  pQueue.push (make\_pair(popped\_pair.second->right->data,  popped\_pair.second->right));  }  return;  }  Node \*buildCartesianTreeUtil(int root, int arr[],  int parent[], int leftchild[], int rightchild[]){  if (root == -1)  return NULL;  Node \*temp = new Node;  temp->data = arr[root];  temp->left = buildCartesianTreeUtil(leftchild[root],  arr, parent, leftchild, rightchild);  temp->right = buildCartesianTreeUtil(rightchild[root],  arr, parent, leftchild, rightchild);  return temp ;  }  Node \*buildCartesianTree(int arr[], int n){  int parent[n], leftchild[n], rightchild[n];  memset(parent, -1, sizeof(parent));  memset(leftchild, -1, sizeof(leftchild));  memset(rightchild, -1, sizeof(rightchild));  int root = 0, last;  for (int i = 1; i <= n - 1; i++){  last = i-1;  rightchild[i] = -1;  while (arr[last] >= arr[i] && last != root)  last = parent[last];  if (arr[last] >= arr[i]){  parent[root] = i;  leftchild[i] = root;  root = i;  }  else if (rightchild[last] == -1){  rightchild[last] = i;  parent[i] = last;  leftchild[i] = -1;  }  else{  parent[rightchild[last]] = i;  leftchild[i] = rightchild[last];  rightchild[last]= i;  parent[i] = last;  }  }  parent[root] = -1;    return (buildCartesianTreeUtil (root, arr, parent,  leftchild, rightchild));  }  int printSortedArr(int arr[], int n){  Node \*root = buildCartesianTree(arr, n);  printf("The sorted array is-\n");  pQBasedTraversal(root);  }  int main(){  int arr[] = {5, 10, 40, 30, 28};  int n = sizeof(arr)/sizeof(arr[0]);  printSortedArr(arr, n);  return(0);  } |

**Output**

****

**Complexity**

**Time Complexity:**

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

**Auxiliary Space :**

We use 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**

**[1] Cartesian Tree Sorting-** [**https://www.geeksforgeeks.org/cartesian-tree-sorting/**](https://www.geeksforgeeks.org/cartesian-tree-sorting/)

***Thank You***