Data Structures and Algorithms

Short Notes

Topics : Application of Heaps

Heaps

A *max_heap* is a container which can facilitate the following operations efficiently

1. Insert a new element into the container in $O(\log n)$
2. Get a view on the maximum element in the container in $O(1)$
3. Delete a single copy of the maximum element from the container in $O(\log n)$
4. Get the size of the container in $O(1)$
5. Check whether the container is empty or not in $O(1)$

Similarly, we can also define a *min_heap* which facilitates all the above operations with *maximum* replaced by *minimum*

Usage

Heaps are implemented in **STL** via the *priority_queue* class. Instantiating a new object of this class would by default result in a *max_heap*. To see how to create a *min_heap*, see the letcure notes on comparators.

Applications

We discussed 3 classical applications of heaps in the class. You can find their discussion and pseudocodes in the following section

Minimum Cost to Connect Ropes

Problem : Given n ropes of possibly different length, find the minimum cost to connect them. Connecting 2 ropes of length a and b incurs a cost of $a + b$.
Solution : Let us focus on the first 2 ropes that are connected.Let their length be a and b. At some stage, the resulting rope of length $a + b$ would be connected to another rope of length, say x, which would further be connected to y. Let us analyze the total cost.

$$\begin{aligned} a+b \\ a+b+x \\ a+b+x+y \\ \vdots \\ a+b+x+y+z+\dots \end{aligned}$$

The total cost would be the summation of all the numbers in all the above levels. Notice that a and b are appearing at every level and hence we should minimize them. The only choice is to pick the 2 smallest elements of the array and connect them in the first step. After this step, the problem is reduced to a sub problem of the same nature. Hence, we again pick the current 2 smallest elements and add them. We continue this until we have one single rope.

Algorithm Find the minimum cost to connect n ropes

```
1: function CONNECT_ROPES($Arr$)
2:     for $i = 1 : N$ do
3:         $min\_heap.push(a[i])$
4:     $cost \leftarrow 0$
5:     while $min\_heap.size > 1$ do
6:         $min\_element \leftarrow min\_heap.top, min\_heap.pop$
7:         $second\_min\_element \leftarrow min\_heap.top, min\_heap.pop$
8:         $cost += min\_element + second\_element$
9:         $min\_heap.push(min\_element + max\_element)$
```

Time Complexity

$O(n \log n)$

Median in a Stream of Integers

Intuition

Suppose we have an array of n numbers. We want to divide it into 2 segments. The first segment should contain all the elements that are smaller than or equal to the median and the second segment should contain elements bigger than or equal to the median.

1. If n is even, we can first sort the elements and then put the first $n/2$ elements in the first segment and the remaining elements in the other segment. The median in this case is the average of the maximum element of the first segment and the minimum element of the second segment.

2. If n is odd, we can sort the elements. Now, we can place the first $(n-1)/2$ elements in the first segment and the last $(n-1)/2$ elements in the second segment. For the middle element, we have a choice. Let's say that we would always prefer insertion in the first segment whenever there's a choice. Hence, we would insert the middle element in the first segment and the median would be the maximum element in the first segment.

Naturally, this gives us a hint to use *heaps*. We will store the first segment in a *max_heap* and the second segment in a *min_heap*.

Invariant 1

We'll maintain an invariant that the size of the *max_heap* cannot be strictly smaller than the size of *min_heap* (because of the second point above) . Of course, any element of *max_heap* should be less than or equal to any element of *min_heap* (and vice versa).

Handling Insertions

It's clear that if we maintain the above rules and invariant while inserting a new element, we can quickly find the new median.

Invariant 2

Let us say that we would always insert a new element into the *max_heap* and immediately transfer the maximum element of the *max_heap* to the *min_heap*.

Balancing

Now, we need to balance these 2 heaps.

1. If the size of the *max_heap* is greater than the size of *min_heap*, we don't do anything.

2. If the size of the *max_heap* is equal to the size of the *min_heap*, we don't do anything.

3. If the size of the *max_heap* is less than the size of the min heap, we transfer the minimum element of the *min_heap* to the *max_heap*

Can you now see why the size difference of these 2 heaps would never be more than 1?

Algorithm Find the **Median** in a stream of integers

```
1: function MEDIAN_IN_STREAM(Stream)
   ▷ max_heap contains elements smaller than or equal to median
   ▷ min_heap contains elements bigger than or equal to median

2:     for each element in stream do
3:         max_heap.push(element)                                  ▷ Insertion
4:         max_element ← max_heap.top
5:         max_heap.pop
6:         min_heap.push(max_element)

7:         if max_heap.size < min_heap.size then                   ▷ Balancing
8:             min_element ← min_heap.top
9:             min_heap.pop
10:            max_heap.push(min_element)

11:        if max_heap.size == min_heap.size then                  ▷ Find Median
12:            Print (max(max_heap) + min(min_heap))/2
13:        else
14:            Print max(max_heap)
```

Extended Discussion

As you can see, the code is quite concise. It handles a lot of tricky cases underneath. Let us explore some of them

1. Initially, both the heaps are empty. Let us see what happens at the first iteration. In the insertion phase, the first element goes to the *max_heap* and then immediately to the *min_heap*. In the Balancing phase, it comes back to the *max_heap* as its size has become smaller than *min_heap*. In the last phase, we get the correct median.

2. Suppose at some stage, both the heaps are balanced. Now, let us assume that the incoming element needs to go to the *max_heap*. Then, in the insertion phase, it would be inserted into the *max_heap* and the new maximum element would be transferred to the *min_heap*. However, this element would again come back in the insertion phase. Finally, we would get the correct median in the last phase.

3. Suppose at the some stage, both the heaps are balanced and the new element needs to go into the *min_heap*. Then, we would still insert it into the *max_heap* first but it would

immediately be transferred to the *min_heap*. In the balancing phase, a new element would be transferred to the *max_heap* which would be the median.

4. If the size is not equal, then it means that the size of *max_heap* is bigger. Now, suppose the incoming element needs to go to the *max_heap*. Then, in the insertion phase, we would insert it into the *max_heap* and transfer the biggest element to the *min_heap*. There's no need to balance now as the size have become equal. Finally, we would get the correct median.

5. If *max_heap* is bigger and the new element needs to go to the *min_heap*, then we would insert it into the *max_heap* first and then immediately take it out and put it into the *min_heap*. Of course, no balancing would happen and we would get the correct median.

In a nutshell, if the size of both the heaps are equal, then in the next phase *max_heap* would become greater in size. And if at some stage the *max_heap* is bigger in size, then in the next phase, the size of both the heaps would become equal. Of course, we maintain the invariant that any element of *max_heap* is less than or equal to any element of *min_heap* and hence our median formula works.

Time Complexity

$O(n \log n)$

Merge K Sorted Vectors

One technique is to merge the first 2 vectors using the merging technique developed in class. Then, merge the resultant with the third one, and so on. However, the complexity would be $n+2n+3n....$ $= O(n^2)$.
We can use heaps to do it in $O(Total\_Elements * \log k)$. We create a *min_heap* and insert the minimum element of each of the k vectors into the heap. The overall minimum would be the minimum element in the heap. We pop it and print it. Now, the second minimum can either be in the heap or it can be the next element of the vector that the popped element came from. Hence, we insert its right neighbour into the min heap, thus maintaining the size of the heap as k at each step.

Algorithm Merge k Sorted Vectors

Require: This pseudocode is specific to *vector* container in C++

```
1:  function MERGE_SORTED_VECTORS($mat$)
    ▷ This solution destroys the original vectors
    ▷ $min\_heap$ contains the element, and the index of the vector it came from

2:      for each $vector$ in $mat$ do
3:          $reverse(vector)$
4:      for $index = 1 : mat.size$ do                 ▷ Push the minimum element into $min\_heap$
5:          $min\_heap.push(mat[index].back, index)$
6:      while $min\_heap$ is not empty do
7:          $(current\_element, index) \leftarrow min\_heap.top$
8:          $min\_heap.pop$
9:          $mat[index].pop\_back$
10:         Print $current\_element$
11:         if $mat[index]$ is not empty then
12:             $min\_heap.push(mat[index].back, index)$
```

Pratice Problems

1. $k-th$ **Smallest Largest Element** : Given an unsorted array, find the $k-th$ order statistics effectively (i,e without sorting the entire array).

2. **Sort a Nearly Sorted Array** : Given an unsorted array where every element is atmost k distance away from its correct position (on either side), given an efficent algorithm to sort the array in $O(n \log k)$