Question 1: A very large number of random numbers are added to a list. Design and implement an efficient data structure that will maintain a separate list of the k smallest numbers that are currently in the list. Space efficiency must be O(k + n). How would you handle deletions? Perform an amortized analysis of your data structure.

Solution: Binary Heap

Using 2 Binary heaps, one stores the k smallest number that has the largest root, and the other stores the rest number that has the smallest root.

Heap data structure:

|  |  |  |  |
| --- | --- | --- | --- |
| data | Vector<int><int> |  |  |
| Heap\_size | int | 0 |  |
| Parent(i) | int | Return (i-1)/2 | O(1) |
| Left(i) : Return left child | Int | Return 2\*I +1 | O(1) |
| Right(i) : Return right child | Int | Return 2\*I +2 | O(1) |
| Insert(k) | Bool | If(the heap is full)  Return false  I = heap\_size  Heap\_size ++  Data[i] = k  While(I is not 0 and the parent of I is less priority)  Swap I with the parent  Return true | O(log(n)) |
| Delete[i]: delete the index I of data | bool | If(Index is not available):  Return 0  Convert the index with the least priority  Loop through parent to bring this index value to the root  Extract root | O(log(n)) |
| extractRoot() | Int | Int root = data[0]  Swap the root with the last value  Converts the last value to the least priority  Hepify(0) | O(log(n) |
| Hepify(i) | void | Compare the current value to the next children and replace the current one with the higher priority. Replace the step until the current is in the correct position | O(logn) |

The data structure of the list kSmallest.

Let N = k+ n

|  |  |  |  |
| --- | --- | --- | --- |
| kHeap | Heap(larger priority) |  |  |
| nHeap | Heap(smaller priority) |  |  |
| kSmallest(vector<int>, int k) | kSmallest | Create kHeap with the k size  Create nHeap with the n size  Insert the first k element from list to k  Loop to n time, compare if the element is smaller than the biggest number in k, extract the big number to n and push to k. other wish just push to n | O(Nlog(N)) |
| ksmallest | Vector<int> | Return kHeap data | O(1) |
| deleteIndex(int i) | bool | If the index I is less than k, pop I in k, extract the root from n and push to k.  If the indec I is larger than k, extract I – k in n. | O(log(N)) |

Amortized Analysis

Let use the Aggregate method for the Heap:

In the insertion, let call each time we insert an element, in the end, is 1. Each swap we have for each element with the parent is 1. Therefore, each element is added in a height is :

* Insertion = 1.
* Maximum swap = h = log2(n)

We have :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Height (h) | Number of Element (n) | Total of maximum operations  n\*(1+ h) | Total of the element | Average |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 2 | 4 | 3 | 1.333 |
| 2 | 4 | 12 | 7 | 1.714 |
| 3 | 8 | 32 | 15 | 2.133 |
| 4 | 16 | 80 | 24 | 3.333 |

Therefore, the insertion heap is

So, at each insertion, the heap has an average runtime at 1 <= Ai <log(n)

For deletion, let call each time replace the value at index I is 1. The swap in the index I to the root is 1. and hepify is cost log2(n) time. So each time of deleting is cost : 1 + 1 + log(n)

Therefore, the deletion heap is

Let use the Aggregate method for the kSmallest:

For generation, we need to insert a list N element to the kHeap and nHeap.

The first k insertion need an average of k <=k\* Aki < k\*log(k) operation each.

From the next insertion, it need to compare with the root of k, cost 1

And then it may lead to 2 cases:

When element e is less than root of k: Ai1 = 2log(k) + log(n)

We need to extract the root k: log(k)

We need to insert e: log(k)

We need to insert root to nHeap: log(n)

When e is greater than root of k, we only need to insert to nHeap: Ai2 = log(n)

For deletion, an index in kSmallest:

If i is less than k, the deletion will be Ad1 = 2\*log(k) + log(n)

We need to delete I in k = log (k)

We need to extract root of n = log(n)

We need to insert the root in k = log(k)

If i is larger than k, we only need to insert to nHeap: Ad2 = log(n)

By using the Aggregate method and a list example below we have:

Assume all log() is base 2.

As we know that when an element is inserted, we need 2\*log(k) + log(n) or log(n) for both deletion and insertion. So, every time we insert an element we prepay for the deletion as well.

Let have a random list: 8 1 9 2 0 5 7 3 4 6, and k = 5, n = 10

At the first 5 number, we need to insert 8 1 9 2 0, that we have 1 + 2 + 2 + 3 + 3 = 11 < k\*log(5) for insertion.

Then for 5 7 3 4 6, we need

5 is less than 9 (9 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(1) = 4

7 is less than 8 (8 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(2) = 5

3 is less than 7 (7 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(3) = 6

4 is less than 5 (5 is the root k), it cost 2\*log(k) + log(n) = 2\*log(5) + log(4) = 6

6 is larger than 4 (4 is the root k), it cost log(n) = log(5) = 6

Total is 11 + 4+5+6+6+6 = 38 => A = 38/ 10 = 3.8 < log(k) + log(n) = log2(5) + log2(5) = 4.6

For deletion, the algorithm is similar to generation, the amount of insertion is as same as the amount of deletion (Ai1= Ad1 and Ai2 = Ad2)

So A = Ad = Ai =