2102. Sequentially Ordinal Rank Tracker

Problem Description

In this problem, we need to develop a custom `SORTracker` that tracks a given location's score and name. The goal is to store the location information of the top `n` scores. We need to create two functions Add (to add a location and its score to our data structure) and Get (to retrieve the location with the highest score):

1. `add(name: str, score: int)`: adds a location with its name and score to the data structure. If the data structure has more than `n` elements, remove the smallest element with the lowest score. If two locations have the same score, keep the location with the alphabetically smaller name.
2. `get() -> str`: retrieves the name of the location with the highest score and removes it from the data structure. If there are multiple locations with the highest score, return the location with the alphabetically smaller name.

Approach

The solution uses two priority queues (min and max heap). This allows getting and adding elements with the desired properties in $O(\log k)$ time, where k is the maximum number of elements we want to keep in our priority queues.

- Add comparator functions for both heaps as `CompareLeftMinHeap` and `CompareRightMaxHeap` that compare the scores and names of two given locations.
- The min heap (`l`) stores the largest k locations, sorted by their scores.
- The max heap (`r`) stores the removed locations (i.e., locations with scores smaller than the top k).

Example:

Consider an example where we want to store the top two locations. At first, both heaps are empty.

```
l: {}
r: {}
```

Now, let's add some locations and update the heaps:

```
add("A", 10)
l: {(A, 10)}
r: {}

add("B", 5)
l: {(A, 10), (B, 5)}
r: {}

add("C", 7)
l: {(A, 10), (C, 7)}
r: {(B, 5)}
```

Let's get the top location:

```
get() -> "A"
l: {(C, 7)}
r: {(B, 5)}
```

C++ Solution

```cpp
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

struct Location {
  string name;
  int score;
  Location(const string& name, int score) : name(move(name)), score(score) {}
};

class SORTracker {
 public:
  void add(const string& name, int score) {
    l.emplace(name, score);
    if (l.size() > k + 1) {
      const Location location = l.top();
      l.pop();
      r.emplace(location.name, location.score);
    }
  }

  string get() {
    const string name = l.top().name;
    if (!r.empty()) {
      const Location location = r.top();
      r.pop();
      l.emplace(location.name, location.score);
    }
    ++k;
    return name;
  }

 private:
  struct CompareLeftMinHeap {
    bool operator()(const Location& a, const Location& b) {
      return a.score == b.score ? a.name < b.name : a.score > b.score;
    }
  };

  struct CompareRightMaxHeap {
    bool operator()(const Location& a, const Location& b) {
      return a.score == b.score ? a.name > b.name : a.score < b.score;
    }
  };

  priority_queue<Location, vector<Location>, CompareLeftMinHeap> l;
  priority_queue<Location, vector<Location>, CompareRightMaxHeap> r;
  int k = 0;
};
```

Note that the provided solution only covers the C++ language. Please implement the solution in Python, JavaScript, and C# languages as required.## Python Solution

```
python
import heapq

class Location:
    def __init__(self, name:str, score:int):
        self.name = name
        self.score = score

    def __lt__(self, other):
        return (self.score, self.name) < (other.score, other.name)

    def __eq__(self, other):
        return (self.score, self.name) == (other.score, other.name)

class SORTracker:
    def __init__(self):
        self.l = []
        self.r = []
        self.k = 0

    def add(self, name: str, score: int):
        heapq.heappush(self.l, Location(name, score))
        if len(self.l) > self.k + 1:
            loc = heapq.heappop(self.l)
            heapq.heappush(self.r, (-loc.score, loc))
        return self

    def get(self) -> str:
        name = self.l[0].name
        if self.r:
            loc = heapq.heappop(self.r)
            heapq.heappush(self.l, (loc[1]))
        self.k += 1
        return name
```

JavaScript Solution

```
javascript
class Location {
    constructor(name, score) {
        this.name = name;
        this.score = score;
    }
}

class MinHeap {
    constructor(compare) {
        this.compare = compare;
        this.heap = [];
    }

    push(val) {
        this.heap.push(val);
        this.bubbleUp(this.heap.length - 1);
    }

    pop() {
        let res = this.heap[0];
        this.heap[0] = this.heap[this.heap.length - 1];
        this.heap.pop();
        this.bubbleDown(0);
        return res;
    }

    top() {
        return this.heap[0];
    }

    size() {
        return this.heap.length;
    }

    bubbleUp(index) {
        let parent = Math.floor((index - 1) / 2);
        while (parent >= 0 && this.compare(this.heap[index], this.heap[parent])) {
            this.swap(parent, index);
            index = parent;
            parent = Math.floor((index - 1) / 2);
        }
    }

    bubbleDown(index) {
        while (true) {
            let min = index;
            let left = index * 2 + 1;
            let right = index * 2 + 2;

            if (left < this.heap.length && this.compare(this.heap[left], this.heap[min])) {
                min = left;
            }

            if (right < this.heap.length && this.compare(this.heap[right], this.heap[min])) {
                min = right;
            }

            if (min !== index) {
                this.swap(min, index);
                index = min;
            } else {
                break;
            }
        }
    }

    swap(i, j) {
        let temp = this.heap[i];
        this.heap[i] = this.heap[j];
        this.heap[j] = temp;
    }
}

class SORTracker {
    constructor() {
        this.l = new MinHeap((a, b) => a.score === b.score ? a.name < b.name : a.score > b.score);
        this.r = new MinHeap((a, b) => a.score === b.score ? a.name > b.name : a.score < b.score);
        this.k = 0;
    }

    add(name, score) {
        this.l.push(new Location(name, score));
        if (this.l.size() > this.k + 1) {
            let loc = this.l.pop();
            this.r.push(new Location(loc.name, -loc.score));
        }
    }

    get() {
        let name = this.l.top().name;
        if (this.r.size() > 0) {
            let loc = this.r.pop();
            this.l.push(new Location(loc.name, -loc.score));
        }
        this.k += 1;
        return name;
    }
}
```

Java Solution

```
java
import java.util.*;

class Location {
    String name;
    int score;

    public Location(String name, int score) {
        this.name = name;
        this.score = score;
    }
}

class SORTracker {
    private PriorityQueue<Location> l;
    private PriorityQueue<Location> r;
    private int k;

    public SORTracker() {
        this.l = new PriorityQueue<>(new Comparator<Location>() {
            @Override
            public int compare(Location a, Location b) {
                return a.score == b.score ? a.name.compareTo(b.name) : b.score - a.score;
            }
        });

        this.r = new PriorityQueue<>(new Comparator<Location>() {
            @Override
            public int compare(Location a, Location b) {
                return a.score == b.score ? b.name.compareTo(a.name) : a.score - b.score;
            }
        });
        this.k = 0;
    }

    public void add(String name, int score) {
        l.offer(new Location(name, score));
        if (l.size() > k + 1) {
            Location location = l.poll();
            r.offer(location);
        }
    }

    public String get() {
        Location top = l.poll();
        String name = top.name;
        if (!r.isEmpty()) {
            Location location = r.poll();
            l.offer(location);
        }
        k += 1;
        return name;
    }
}
```