

Exercise 2 Spec (25su)

The objectives of this exercise are:

- Understand the binary heap data structure by implementing it
- See how to modify the binary heap data structure to add new operations efficiently
- Apply the binary heap data structure to create a new data structure

Overview

This exercise consists of the following parts:

1. Build a binary min heap
 - In this section, we will provide descriptions of methods you must implement for a binary min heap data structure, as well as the requirements for that implementation
2. Build a binary max heap
 - After building your binary min heap, you will modify that implementation in order to implement a binary max heap
3. Chess Leaderboard
 - You will use your binary min heap and binary max heap implementations to create a new data structure that we're calling a TopKHeap. We'll use this as a data structure for tracking the best chess players of all time!

Motivation: Chess Ranking

The [Elo rating system](#) was created by Physicist Arpad Elo as a method for ranking chess players in the United States Chess Federation. The challenge with chess ranking at the time was that not all chess players can directly play each other directly. This means that to provide a complete ranking of all players, we need some way of determining the relative quality of players who have never actually played head-to-head in a game.

The Elo system solves this problem by assigning a score to each player. This score is then used to predict the likelihood of each player winning in a hypothetical head-to-head. The player with the larger score is considered to be more likely to win the match, and the greater the magnitude of the score difference, the more likely that outcome should be. For example, if Player A has an Elo score that is 100 points higher than Player B, then Player A should win 64% of head-to-head games played. If the difference is instead 200 points, then Player A should win 76% of the time. The details of how the points map to probabilities are not important, so long as there exists a mapping.

Because each game gives us information about a player's skill level, every outcome will change each player's score. The more surprising the outcome, the more a player's score is going to change. For example, if Player A has a score that is 200 higher than Player B does, and they play 100 head-to-head games, neither player's score should change if Player A wins 76 of them. That's exactly what we expected to happen! If Player A wins 80 of them, then Player A's score will increase slightly, and Player B's score will decrease, because the result was very slightly different from our expectation. If Player A wins only 25 of them, then Player A's score will decrease by a lot, and Player B's score will increase substantially. By updating the scores after each match, the Elo rating system can be used to predict who would win in any hypothetical match, even if those two players never played each other, and [even if they were not active or alive at the same time](#) (with some caveats)!

Today, the Elo system is used for much more than chess. For example, it was used for NCAA football bowl selections from 1998-2013, a modified version is presently used for Women's FIFA rankings, many esports communities use it for ranking players, Topcoder and CodeForce use it for online coding competitions, and it can be used for [comparing AI models](#).

While in this assignment, you will not be implementing Elo ratings, we will be writing a data structure that might help us to keep track of the top contenders according to their Elo scores. Considering the chess example, on any particular day, there may be hundreds, thousands, or millions! of chess games going on. If each game adjusts every player's Elo rating, then we want to have a data structure to help us manage the ever-changing rankings.

If we only wanted to keep track of the best player, a heap might be a good choice. If we wanted a perfect ranking of all chess players, we'd probably want to maintain a sorted list. In this assignment, we'll create a data structure that allows us to efficiently maintain the top k players for any k we choose. We will call this data structure a TopKHeap.

Implementation Guidelines

Your implementations in this assignment must follow these guidelines

- You may not add any import statements beyond those that are already present. This course is about learning the mechanics of various data structures so that we know how to use them effectively, choose among them, and modify them as needed. Java has built-in implementations of many data structures that we will discuss. In general, you should not use them.
- Your Heap implementations must store all items in arrays. These arrays should use index 0 for storing items. You should only create a list for the toList method.
- Do not have any package declarations in the code that you submit. The Gradescope autograder may not be able to run your code if it does.
- Remove all print statements from your code before submitting. These could interfere with the Gradescope autograder (mostly because printing consumes a lot of computing time).
- Your code will be evaluated by the Gradescope autograder to assess correctness. It will be evaluated by a human to verify the running time. Please write your code to be readable. This means adding comments to your methods and removing unused code (including “commented out” code).
- For the sake of running times, you may consider all HashMap/HashSet operations to run in constant time.

For this assignment, there is **partial credit for passing only a few test cases**. For example, if you failed on any test case on BinaryMaxHeap, you will receive partial credits corresponding to that test case. BinaryMaxHeap and BinaryMinHeap both worth 30 points, and TopKHeap worth 25 points. There are still 5 points of manual inspection on all three classes; thus, there are 85 points of autograder and 15 points of manual inspection. Meanwhile, you will be able to see the details on the testcases you failed, and there are no hidden tests on this assignment.

Provided Code

Several Java classes have been provided for you in the zip file. Here is a description of each.

- **MyPriorityQueue**
 - A priority queue interface. It is called `MyPriorityQueue` because Java already has a built-in interface called `PriorityQueue`, and so this avoids any kind of naming issues.
 - *Do not submit this file*
- **BinaryMinHeap**
 - You will create a Binary Min Heap data structure that implements the `MyPriorityQueue` interface. Each method should satisfy the requirements described in the spec and the comments in the `MyPriorityQueue` interface.
 - ***You will submit this file to Gradescope***
- **BinaryMaxHeap**
 - You will create a Binary Max Heap data structure that implements the `MyPriorityQueue` interface. It will be equivalent to `BinaryMinHeap`, except that it's a max heap!
 - ***You will submit this file to Gradescope***
- **TopKHeap**
 - This data structure makes it efficient to insert items, update the priorities of items, and find the top-k items (where k is a parameter given in the constructor). This will be used for the “Chess Players” problem.
 - ***You will submit this file to Gradescope***
- **Pair**
 - For testing purposes, we will be inserting these `Pair` objects into the data structures above. Each object has a first field K (key) representing the key and a comparable V (value) representing the priority corresponding to that key. While ordering the objects, you should order them according to their V values. We have also implemented an `equals` method which considers two `Pairs` to be equal if they have both the same key and the same priority. There is also a `hashCode` method implemented for efficient use of `HashMaps`.
 - Specifically, for the sake of context of this exercise, K will always be `String` and V would always be `Integer` on autograder.
 - *Do not submit this file*
- **Client**
 - This class performs some basic testing of some methods within the above classes. The Gradescope autograder will do more precise testing than this.
 - *Do not submit this file*

Part 1: BinaryMinHeap

The TopKHeap data structure that we create will make use of binary heaps, like we discussed in class. To get started, implement a binary min heap data structure as described here.

The BinaryMinHeap data structure uses Java generics to store any classes that implement the Comparable interface, guaranteeing that they have a compareTo method. Recall that to use the compareTo method, you invoke it as an instance method of an object and provide a second object as an argument. The result is then an integer which indicates which object is “greater”. In particular, if we call `thingA.compareTo(thingB)` we expect the result to be a negative integer if `thingA < thingB`, it should be a positive integer if `thingA > thingB`, and it should be 0 if `thingA == thingB`. To keep it straight in my head, `thingA.compareTo(thingB)` gives the sign of `thingA - thingB`.

Meanwhile, please note that the idea of PriorityQueue/Heap **does** allow duplicated items and duplicated priorities, as it only aims to find the maximum/minimum amongst all elements. However, for the sake of the content in this exercise, and since we are using a HashMap to store the corresponding index of items, we will **NOT** allow duplicate keys in this assignment. Therefore, you can assume that the K values are unique among all elements.

A few functions have implementations provided. These are intended to be helpful, but you may alter them if you would prefer they work differently.

Part 2: BinaryMaxHeap

The TopKHeap data structure will actually be using BOTH a max heap and a min heap. Modify your BinaryMinHeap implementation to additionally implement a BinaryMaxHeap. You’ll see that almost all of the methods will be identical for the max heap. If you would prefer, you’re welcome to refactor your implementations to have a shared abstract class for those identical methods. This is not required, though. You’d be welcome to just copy-paste those implementations. If you do use an abstract class, make sure you submit it to Gradescope!

Part 3: TopKHeap

Now, the moment we’ve been waiting for, the TopKHeap! In this data structure, we make use of a max heap and a min heap to maintain a collection of the top k chess players. Implement all methods provided. See the method comments for the expected behavior and required running times. **We strongly recommend you start doing this portion after you passed all tests in BinaryMinHeap and BinaryMaxHeap, as this portion strongly relies on the previous portion.**

Note that to achieve tie breaking between elements and make our autograder easier to check potential errors, we do **NOT** allow duplicated priorities to happen in TopKHeap. Thus, we are using a HashSet to store the distinct values. Since BinaryMaxHeap and BinaryMinHeap already

forbids duplicated keys to occur at the same time, you can assume that all K and V are unique in TopKHeap.