Analysis of Heap Sorting Methods Implemented with the Heal Point Ranking System

Willie Stevenson and Sylvia Allain
December 13, 2012

Table of Contents

Analysis of Heap Sorting Methods Implemented with the Heal Point Ranking System

1.0 Introduction

The Heal Point Ranking System, tests the running times of insertion and deletion techniques of the binary heap, the pairing heap and the skew heap. The data that is being inserted into the heaps is the TI of various high school teams in Maine as they compete against each other in twelve preliminary matches. Deletion is used to rank the teams from largest TI to smallest. The TI is calculated by an algorithm which involves the division of opposing teams that a team has defeated.

In our study, we will record the running times in nanoseconds of each of these three heaps over several combinations of preliminary outcomes between the teams and record the average running time for each heap. Our prediction is that the binary heap will be the most efficient heap to implement for this project due to its logarithmic worst case time for both insertion and deletion.

2.0 Hypothesis

The binary heap would be the most suitable heap for this project. The binary heap is a balanced heap with logarithmic worst-case time (O(log N)) for both insertion and deletion. It allows for simple methods for insertion while maintaining a balanced structure.

The pairing heap is structurally unconstrained which allows for a simple insertion method that takes constant time. This is an advantage to insertion since no balancing techniques are used, however it has a linear worst-case time (O(N)) where each element is a child of the root to be deleted. In order to rank all teams, every team must be added to the heap, then every team must be removed in order to descending TI. Thus, the worst-case running time of insertion and deletion is O((N + 1)/2).

The skew heap is not suitable for this program since one element is inserted at a time, thus unnecessary recursive balancing methods are used in order to merge the element with the tree. The structure of the skew heap is similar to that of the binary heap, but since the skew heap had no balancing condition, the depth of the tree is not guaranteed to be logarithmic. In a best-case scenario, the tree depth would be logarithmic, however the tree may have a linear worst-case time (O(N)). Thus, we predict that the skew heap would be the least suitable heap for this project.

3.0 Heal Point Ranking System

For this project, we sort ten teams in order of rank after entering the scores for each preliminary round of games. The teams we used were the local high school teams of Maine. These teams are ranked by TI from highest to lowest score. In order to calculate TI, each team is given a rank, “A”, “B”, or “C”, where Bangor, Brewer, Hampden, Old Town, Nokomis, and John Bapst are rank “A” teams, Orono, Bucksport and Ellsworth are rank “B” teams, and Central is a rank “C” team. For each preliminary round, the team's PI is calculated. The PI is calculated by adding up the total points a team receives from defeating teams. These points are given to a team based on the rank of the defeated team —40 points are given to the defeater of a rank “A” team, 35 for a rank “B” team and 30 for a rank “C” team. These points are then divided by the number of games played in order to get a team's PI. The TI is then calculated by adding up all the PI's of every team that the given team has defeated, dividing it by the number of preliminaries, then multiplying the result by ten.

4.0 The Heaps

4.1 The Binary Heap

The binary heap is implemented by using a modified version of the PriorityQueue class in Weiss' book, “Data Structures and Problem Solving Using Java” [1]. The tree itself is implemented as a max heap using an array rather than implementing a node class which connects nodes to their children. The add method is used to insert a team into the heap. This is implemented by adding a “hole” to the last element of the array, then comparing the team to be inserted to the “parent” of the hole located in the array. If the new team's TI is greater than the parent's, the parent and the hole trades places in the array. This is done until the new team's TI is less than that of the parent. The team is then added to the hole.

When the team with the greatest TI is removed from the beginning of the array, the team is added to the end of the array and replaced with an empty “hole”. The percolate down method is called and the child of the hole with the greatest TI is switched with the hole. This is done until the hole has no children. A method, returnRankList, removes each element of the array by removing maximum element until the heap is empty. The ordered rank list is returned.

4.2 The Pairing Heap

The pairing heap was also implemented using a modified version of the class in Weiss' book [1].

The tree is structured using a nested class to implement the nodes of the tree. Each node contains information about the team it contains, it's TI, left child, right sibling and previous sibling. Each team is inserted by creating a new node, then checking to see if there is a root. If a root does not exist, the new node becomes the root. If the root does exist, the TI's of the root and new node are compared and linked using the method, compareAndLink. The node with the lower TI becomes the left child of the node with the greater TI and the previous left child (if it exists) becomes the lesser node's right sibling.

In order to delete the node with the greatest TI, the node is found by a method that returns the root, if it exists. If the child of the root is null, the root becomes null and the tree is empty. Otherwise, the combineSiblings method is called. If there is one sibling, that sibling is returned. Otherwise, the link from each sibling to its next is set to null. Then, each pair of siblings(treeArray[i] and treeArray[i+1] while i increments by two for every pair to be combined) are compared and linked. Then, each new tree is compared and linked to the last tree. The root is returned.

4.3 The Skew Heap

The skew heap was implemented adopting ideas from the pairing heap class Weiss' book [1], modifying the merge method as well as the structure of the nested node class. The tree is structured as a binary tree of sorts. This is used to model each node of the tree. Each node contains information about the team it contains, it's TI, left child, and right child. Each team is inserted by creating a new node, then checking to see if there is a root. If a root does not exist, the new node becomes the root. This logic is very similar to the pairing heap. However, if a root does exist, the newly created node is merged with the root. The size of the tree is then incremented after this merge. The merge compares the newly created nodes current TI with that of the root, recursively iterating down the tree, switching the bigger node's right child with its left child. If there was no right child, the smaller of the two TI contained in the nodes becomes the left child. If there was a right child, the node with the smaller TI is merged with the right child. The root then becomes the left child. The skew node with the larger TI is returned at the end.

In order to delete the node with the greatest TI, we first assign the root (containing the greatest TI) to null. If the root has no children, we are done and the root is assigned null. Otherwise merge the root's left child with the right child. The new root will then be assigned the next largest TI.

5.0 The Project

5.1 The Teams

At the heart of the project is the team class. A given team, at instantiation, contains a team name, a rank (division of school that is predefined), points, which contains the running sum of points received from beating other teams. Next are two crucial pieces of information, the PI and TI. The PI is a preliminary index (PI) which is calculated after each prelim. It assigns a predefined number of points to the winning team of a game based on their division (the summation of this value if contained in the variable points). The PI variable contains the quotient of the accumulated points divided by the number of prelims a team has played thus far. This is used to calculate TI. TI, stored in a separate variable, is calculated by adding the PIs of the schools a team has defeated (defeatedTeam contains a list of these PIs) and dividing by the team's current prelim count, or in other words, how many prelims a team has played. This value is then multiplied by 10 to achieve a relatively round number resulting in the TI. This ranks a team's seasonal standing at any given point in the season. The higher the TI, the higher the standing.

Other small, but crucial information about this implementation include how the PI is calculated. If a given team never wins any games throughout the season (twelve prelims), that team's PI remains at 1. A team then has a calculated TI of 0. Points are rewarded based on the rank of the opposing team. 40 points are awarded for defeating any division A team, 35 for defeating any division B team, and 30 for a division C.

5.2 The Moderator

The Mods class handles the life of the program. At the instantiation of a Mods object through the GUI class, all ten teams and their names are defined as well as each of the tweleve prelim matches designated by each of tweleve array lists. The distribute team scores method ensures that each of the ten teams, whether opposing or current, is taken care of when points are being assigned for a win. The TI is then calculated at the end for the team that called the method.

Each get ranking method (binary, pairing, skew) is meant to calculate the time each data structure takes to add each team to the heap after their TI has been calculated. An array list conting the rank of each team is then returned to the GUI.

In order to properly output each team's history information, a to string method is needed to properly display this information and link it with the GUI. This method outputs the essential

information each team has accumulated over the course of the season. The team's division, teams that a team has defeated, a list of the teams that a team has defeated, and the team in question's TI are outputted to the GUI.

Lastly, after prelim winners are submitted, a new prelim list appears in the GUI. This is the work of the getNextPrelimList method. It simply just increments the current prelim by one, returning the next prelim list. When prelim twelve has arrived and finished, there season is over and there are no prelims left to be had. This, null is returned.

This finalizes the information that the Mods class handles.

5.3 The GUI

The GUI is the interface where the user is able to input team scores to be calculated into TI's. Teams playing against each other in a preliminary round are listed with editable text boxes where the user may enter the scores. The “Submit” button orders the GUI to give the team scores to the moderator to calculate into TI's. The GUI receives a list of teams in order of descending TI in order to output these teams under “Ranking”. The “Submit” button also updates the latest running times of the heaps and outputs them into a text box at the bottom of the window.

The “Go” button is correlated to a pull-down menu that contains the names of the teams. When the “Go” button is pressed, the information of the team that is selected in the pull-down menu appears in the text box below it. The information includes the division of the team, the teams it has defeated and its current TI.

6.0 Procedures

In order to test our hypothesis, we ran the program through all 12 preliminaries five times, collecting 60 running time values for each heap. In order to input scores for the teams, we created a program using Python to generate random scores for each team between 1 and 100. These scores are used to calculate the winners of each prelim since the scores themselves do not affect the computation of the TI. Each running time was recorded for each heap. These were sorted into two lists for each heap: one which includes the running times of the first preliminaries of each tournament and one which excludes them. This exclusion is due to the heaps a dramatically larger running time during its first use. Both types of lists were averaged and placed in a chart. The lists which exclude running times of the first preliminaries were placed in a box-and-whisker plot to represent the distribution of the heaps.

7.0 Results

Our results represented in figure 7.1 show that when the first preliminary is either included or excluded, the binary heap has the lowest running time on average. When the first preliminary is included, the skew heap's average running time is greater than both of the other heaps; however when the first preliminary is excluded, its average running time rivals that of the binary heap and is less than that of the pairing heap.

	Binary Heap Time	Pairing Heap Time	Skew Heap Time
Average Including First Preliminary	38783.33	54866.66	843300
Standard Deviation	12714.72	39839.71	4477053.7
Average Excluding First Preliminary	35418.18	43309.09	35818.18
Standard Deviation	5397.84	7320.37	6257.12

Figure 7.1 Average Running Times and Standard Deviation of Data Heaps in the Heal Point Ranking System

The standard deviation of the pairing heap is the greatest and the binary heap is lowest when initialization is not a factor. However, when initialization is taken into consideration, the skew heap's standard deviation is approximately 352 times that of the binary heap and 112 times that of the pairing heap. As shown in figure 7.2, when the initial heap is excluded, the pairing heap has the greatest range of values while the binary heap has the smallest range.

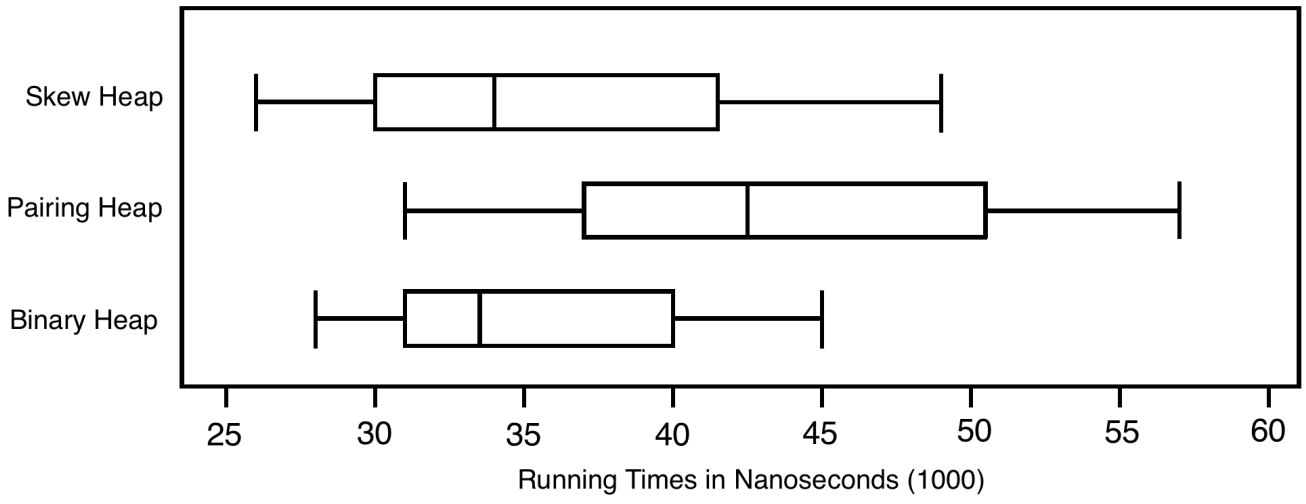

Figure 7.2 Distribution of Running Times of the Heaps without First Preliminary Data

8.0 Analysis

8.1 Binary Heap Analysis

As predicted, the binary heap proves to has the lowest average running time. The balanced nature of this heap allows the running time of each deletion to be limited by the trees' maximum height of three. This is because the percolate down method that is called when a node is deleted is only able to move the empty hole from the root down to the existing leaves. Also when a node is added, the node is added to the bottom of the heap and its traversal up the tree to its new position is limited by the height.

8.2 Skew Heap Analysis

The skew heap has the highest average running time from the initialization of the program until the end. This is as predicted; however it is not due to the large worst-case running time, but the unexpectedly high running time during the first preliminary. The low average running times for the skew heap without the times from the first preliminary may have been due to a high distribution below the average running time. As can be seen in figure 7.2, the median of the skew heap falls below the average of 35818.18 nanoseconds. This may be because the merging process of the tree encourages balance, although it doesn't guarantee it. Although the worst-case running time is linear, the chances of this occurring is unlikely.

The distribution of running times of the skew heap is similar to that of the binary heap, except with a wider range. This may be because the merge of trees tend to be between the longest path of each tree's right-most subtrees. In this program, when a node is inserted into this tree it is inserted with a height of zero. When the node is merged with the tree, there is a chance that the node will be greater than the root, thus it will take one step for the root to become a child of the node. Otherwise, the node will have to search the root's right-most subtrees for its spot in the tree. The average running time for this process is likely to be the length of this path divided by two. In summation, although the worst-case scenario is greater than each of the other trees, a much lower running time is likely due to its algorithm for merging.

8.3 Pairing Heap Analysis

The pairing heap has the greatest average running time when the running time of the first prelim is omitted. No balancing methods are used for the insertion and deletion of nodes, so best and worst-

case scenarios are equally likely. This can be seen in figure 7.2 since the running time are more equally distributed than the other, more balanced, trees. Although the merge algorithm's best and worst-case scenario is the same constant time, the number of times the children of a deleted node are merged is anywhere between one and n times.

8.4 First Preliminary Run Time

It is unclear why exactly the running time of all three heaps are dramatically greater during the computation of the first preliminary results. The system time that is taken into consideration occurs immediately before adding each team into the heap until directly after each one is removed and placed into a list.

8.5 Afterthought

If we were to repeat this experiment, we would gather more data regarding the running times of the heaps during the first preliminary round. Doing so would allow a better representation of average running time during the first preliminary and would allow a more accurate average running time for all heaps. Also, the binary heap represented in the program should be implemented to be more uniform with the skew heap and pairing heap in order to eliminate functions that may affect the running time of the data differently than the other heaps. This uniformity may be achieved through the use of a nested node class rather than an array. Further studies may be focused on the reason for the heaps' large running times during the first preliminary round.

9.0 Conclusion

Although the run times of the methods of the skew heap rival that of the binary heap, the run time of the first preliminary round greatly increases the average run time making this tree inappropriate for this program. Although the pairing heap has the most efficient adding implementation, this data structure would not be efficient due to the need to delete as many times as items are added. In conclusion, the binary heap proves to be an appropriate choice to sort rankings of a constant number of teams. The simple sorting methods are suitable for the insertion and deletion of single elements from the tree.

References

[1] M. A. Weiss, *Data Structures and Problem Solving Using Java: 4th Edition*. Addison-Wesley, 2010.