6.890 – Learning-Augmented Algorithms Spring 2019

Project - Towards Learned Binary Heaps

Adib Hasan *Angelos Pelecanos*

Abstract

Recent advances in machine learning algorithms and the increase of computing power have enabled the use of machine learning techniques in various fields. We are studying the case of augmenting classical algorithms and data structures using machine learning methods. This paper will focus on augmenting the binary heap using machine learning methods.

1 Introduction

In this project, our goal is to research the possibilities of augmenting a popular data structure using machine learning methods.

This project will focus on binary heaps. The binary heap is a very famous data structure that has a wide variety of applications, from network routing to optimization. Every specific application of the binary heap has a specific type of input that might be generated from a distribution, which we expect to exploit in order to optimize the performance of the binary heap.

Our main goal is to reduce the insertion and extraction time of the binary heap. In the classical binary heap implementation, new values are inserted in the leaves and are propagated at the top of the heap until they do not violate the heap property. This operation needs $O(\log n)$ comparisons and swaps, for a heap with n values.

Our contribution is the introduction of some "shortcut" nodes that will help make insertion and extraction faster for values that will end up in the top layers. The way that we will be mapping values to these "shortcut" nodes will be through a machine learning model, which our data structure considers as a black box and has no guarantees regarding its structure or error rate.

Another goal is to develop a data structure that satisfies key properties introduced by [PSK18], which are independence, consistency and robustness. In this paper we present the new Learning-Augmented Binary Heap we developed that satisfies the above key properties.

The paper starts off with a short review of how the original Binary Heap works. The subsequent sections explain the features introduced in the Augmented Binary Heap and several problems that arise with their respective solutions. We include a final section with some experiments done in order to compare empirically the performance of the original Binary Heap with the performance of our Augmented one. Our metric was the number of comparisons required for both data structures for the whole operation. As it turns out, our results are promising, as our data structure's performance is at most a small constant worse than the original Binary Heap and in the case of strong structure in the data it has the potential of improving its performance significantly.

2 Binary Heaps

In this section, we will review the original Binary Heap data structure. The main operations that we will consider in this paper are insertion and extraction of an element. This section is devoted to reviewing how the usual implementation of a Maximum Binary Heap works.

2.1 Structure

A binary heap is a binary tree where every node satisfies the *heap property*. The (max) heap property states that every node must have a value greater than or equal to its children.

In this paper, we will assign to the root of the binary tree the index 1. If a node has been assigned index x, then its left child (if it exists) is assigned index $2x$ and its right child has the index $2x + 1$. Thus for a binary heap of N nodes, all nodes are assigned a distinct index from 1 to N.

This resembles how Binary Heaps are implemented in software, as the whole tree is stored as an array and traversing the tree involves changing the index of our current element by doubling or halving.

2.2 Insertion

Assume we have a binary heap with N nodes. Insertion of a value v happens at the position that corresponds to index $N + 1$. Then this value v is swapped upwards with its parent until it satisfies the heap property. More specifically, while the current parent p of v has a value less than v, then v and p are swapped.

2.3 Extraction

From the construction of the binary heap and the heap property, the maximum-value node is the one at the root. Given that we have a binary heap with N elements, we swap the root with the element assigned number N (the last leaf). Then the new root is swapped down with the maximum-value child until the heap property is again satisfied.

More specifically, let's give the last node the value v. Then while the node corresponding to value v (which was moved to the root) is not largest than both of its children, that node is swapped down with its child with the maximum value. This process continues recursively until the heap property is satisfied.

3 General Idea and Motivation

Currently, the original Binary Heap implementation requires $O(\log N)$ comparisons and swaps for inserting an element to the heap. Extracting also takes $O(\log N)$ comparisons and swaps. Additionally, due to the structure of the Binary Heap around 87.5% elements are in the bottom three layers, but $< 1\%$ are in the top few layers. Thus, insertions in the top few layers is very costly.

Our contribution is interested in optimizing the insertions at the top, which can be rare, but can also happen often if the distribution is changing, for example in the case of an A^* algorithm, if the potential value of states increases as we approach the target.

The way we are doing this, is by keeping some placeholder values that act like "shortcuts" to insert values to the top layers.

4 Introducing Dummy Nodes

The placeholder nodes mentioned in the previous section will be named *Dummy Nodes*. Dummy Nodes are just nodes with an assigned "superficial" value. This value does not correspond to an actual value inserted, but it is used to satisfy the heap property and determine the place of the Dummy Node in the Binary Heap.

In order to build an Augmented Binary Heap with N elements, we will include in the building set K additional Dummy Nodes with their values drawn from the learned distribution of our N elements. After building, we will create a *Dummy List*, a list that keeps the dummy nodes in order of increasing index in the Augmented Binary Heap. Thus the values of the dummy nodes in the dummy list should be fairly sorted, meaning that the number of inversions should be low.

Now we need to learn a model $\mathcal{M}$ that takes in a value to insert and returns an index of the dummy list. This model is trained using as training data the values of the dummy nodes and their respective index in the dummy list. Then the value is inserted in the node corresponding to the given index of the dummy list. As our model is not perfect, we expect that the heap property might not be satisfied if we replace the dummy node with a real node of a different value. Thus to ensure correctness, we will have to do some additional swapping up or down to preserve the heap property.

These swap up or down actions are precisely the same with the regular Binary Heap ones. For swapping up, we swap with a parent while the parent has a larger value. For swapping down, while the node does not have a largest value than both of the children, we swap the node with the maximum-value child.

Note that the number of swaps comparisons is still upper-bounded by the height of the tree, thus the performance of our data structure remains bounded at all times, with the probability of no additional swaps if our prediction is perfect.

4.1 Dealing with failure

By introducing Dummy Nodes, a problem arises. Consider the following scenario. We have a perfect model $\mathcal{M}$ and two dummy nodes with values v and $v + 1$. If we get the insert operation with value v twice, then the first insert will be completed successfully. During the second insert, our model $\mathcal{M}$ will predict the index of the already occupied dummy node. But this index does not correspond to an available dummy node and thus our new value cannot be inserted at this dummy node.

This scenario is essentially covering the fact that we need to remove the replaced dummy nodes from the dummy list. Otherwise, we will need to invalidate them in some way in order to avoid

them being chosen again by the model.

To solve that, we need a quick data structure to help us find the next available dummy. This is the Union-Find data structure which will be built on top of our dummy list. More specifically, every dummy node in the dummy list will have a parent and a next available dummy. Initially the parent of every dummy node is itself and the next available dummy is also itself. When a dummy is chosen to be replaced by a real value, this dummy is connected in a Union-Find way to the dummy in the next index (or the first one if this dummy is the last). Thus the next available dummy is the next available dummy of the parent of the predicted dummy.

The complexity of the Union-Find data structure is $O(\alpha(K))$ amortized, where K is the number of elements in the dummy list. This is indeed an added complexity, it is however asymptotically faster than the $O(\log N)$ run-time of the insertion operation on a binary heap with N elements. Additionally, these operations are done on the indices of the elements and not their actual values, which scales well in real-life applications. This is because in real-life applications, the items stored in the Binary Heap can be complex and their comparison can be an expensive operation, whereas comparing integers is a primitive operation.

5 Extracting Maximum Value

We now turn our attention to how Extract should work for the Augmented Binary Heap. When an Extract operation is received, there is a chance that the root of the Binary Heap is a Dummy Node, which does not carry any actual value and thus cannot be returned.

To resolve this issue, we keep an extra field for every node of the Binary Heap called *argmax* (or argmin in the case of the Minimum Binary Heap). More specifically, for node u, u.argmax will keep the index of the node that is not a dummy node and has the maximum value in the subtree of u.

As a result, our extraction changes to swapping the last element with the argmax of the root. This has the benefit of potentially decreasing the comparisons and swaps of the last element, as it gets swapped with another node at a layer lower than the first.

In order to maintain the correct argmax values for all nodes, extra care has to be taken. More specifically, after every series of swap operations that happen after the insertion or the extraction of a value, we have to revisit the nodes that were swapped and update their argmax according to the new values of their children in a bottom-up fashion. This operation is just a constant factor of the operation that was completed and thus does not change our asymptotic analysis.

6 Hybrid Insertion

This section is related to an optimization that we developed. This optimization is essentially allowing our Augmented Binary Heap to fall back to the operation of a normal Binary Heap if a bad prediction is received from the model.

To gain more insight, let us consider that we are given a value to insert v and it is predicted to go to dummy node v^*, which is at the first level of the tree. However, this value v is actually a very small value and after the swaps it will end up on the second to last level of the tree. In this case,

the original Binary Heap would need 2 comparisons and swaps, whereas our Augmented Binary Heap will need $\Omega(\log N)$ such operations. This scenario corresponds to a bad prediction, where the Binary Heap outperforms our Augmented Binary Heap.

How can we detect efficiently if such a bad prediction is received from our model? We can just enforce the rule that after insertion at a dummy node, the dummy node cannot be swapped down. Swapping down would mean that we "overshoot" our prediction and placed the inserted value too high in the heap. Although swapping down a few times might still be beneficial, we decided to avoid swapping down altogether, to make sure that we never cancel any of the comparisons that we gained by choosing a dummy node.

What do we do if an inserted node replacing the predicted dummy node has to be swapped down? We do not replace the dummy node and fall back to the regular Binary Heap insert.

7 Improved Method for Dummy Insertion

During our first experiments, we realized that our Augmented Binary Heap was using consistently more comparisons than the original Binary Heap. This was due to the fact that by drawing uniformly at random from the input distribution that was used to build the heap, then most of the dummy nodes would end up in the lower layers of the Binary Heap. This was introducing extra comparisons, as if an input was matched to a dummy node in the low levels, this would result to our dummy node not being a shortcut, but also needing extra comparisons from the Binary Heap.

Thus we decided to enforce more structure to the location of the dummy nodes in our tree. More specifically, if we control the value of the dummy nodes we can also control their location in the tree, something that will help us avoid many comparisons.

7.1 Adding Dummy above a Node

To resolve this issue, we defined the following operation: Adding a dummy above a node v. What this will do is it will create a dummy node v' with the same value as v. Then, it will push v' to the position of v and then v can be put to any of its children.

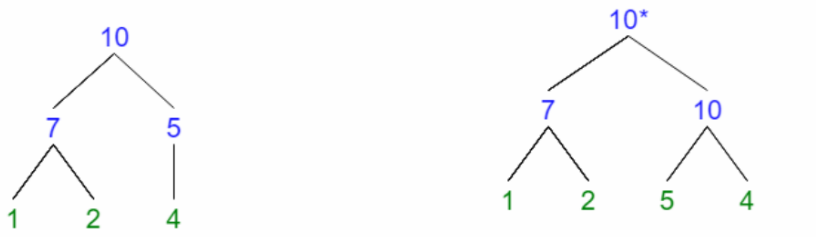


Figure 1: A dummy 10* is added at top position in a max heap. 10 is pushed down to 5, and 5 to 4. This displaces 4 from the heap, but it was from the last layer, so re-inserting it will be cheap. This operation introduces a dummy 10* which can be used to insert a large value with almost no cost.

We choose the child randomly (or we can try to balance the number of times we pushed to each

child) to make sure that we are pushing the dummy nodes uniformly throughout the tree.

This procedure continues with moving a chain of nodes one level lower until potentially a leaf is extracted completely from the tree. This leaf is then added through a normal insert into the Augmented Binary Heap. As this node used to be a leaf, we expect its insertion to not take many comparisons and swaps, as it should be located in the lower layers.

Now, the building procedure can be replaced by adding a dummy for every node in the top L layers, starting from the nodes in the lower layers and working our way up. This makes the whole procedure have a high degree of parallelism, which will be discussed below. It is still upper bounded by a constant factor of the operation of the binary heap. Also, most of the operations can be implemented as pointer swaps, which can be faster than item comparisons.

7.2 When Dummy Nodes run out

In this setting, when the number of dummy nodes reaches zero, this means that we cannot do any more "augmented" inserts using the dummy nodes. In that case, a new rebuild of the Augmented Binary Heap is required, along with the generation of new dummy nodes, dummy list and model $\mathcal{M}$.

As we have seen above, this might be a costly operation and thus we would like to avoid rebuilding our binary heap often. For this to happen, we should make sure that the number of dummies in every tree is a constant factor of the total number of items in the tree. This will make the number of rebuilds logarithmic and will help upper bound our execution time. Additionally, we can use a similar argument to a paper by Kraska et al [KBC18], where they underline how efficient CPUs and GPUs are at processing commands in parallel. In our case, the building process has a high degree of parallelism, as the adding of a dummy above a node is independent for two nodes on the same level. Additionally, most machine learning models have also a high degree of parallelism, which can make such an idea in practice significantly faster.

8 Experiments

8.1 General Setup

For our experiments, we implemented both the original Binary Heap and our Augmented Binary Heap as described above. Both implementations were made in Python, mainly for its efficiency in developing code and prototyping.

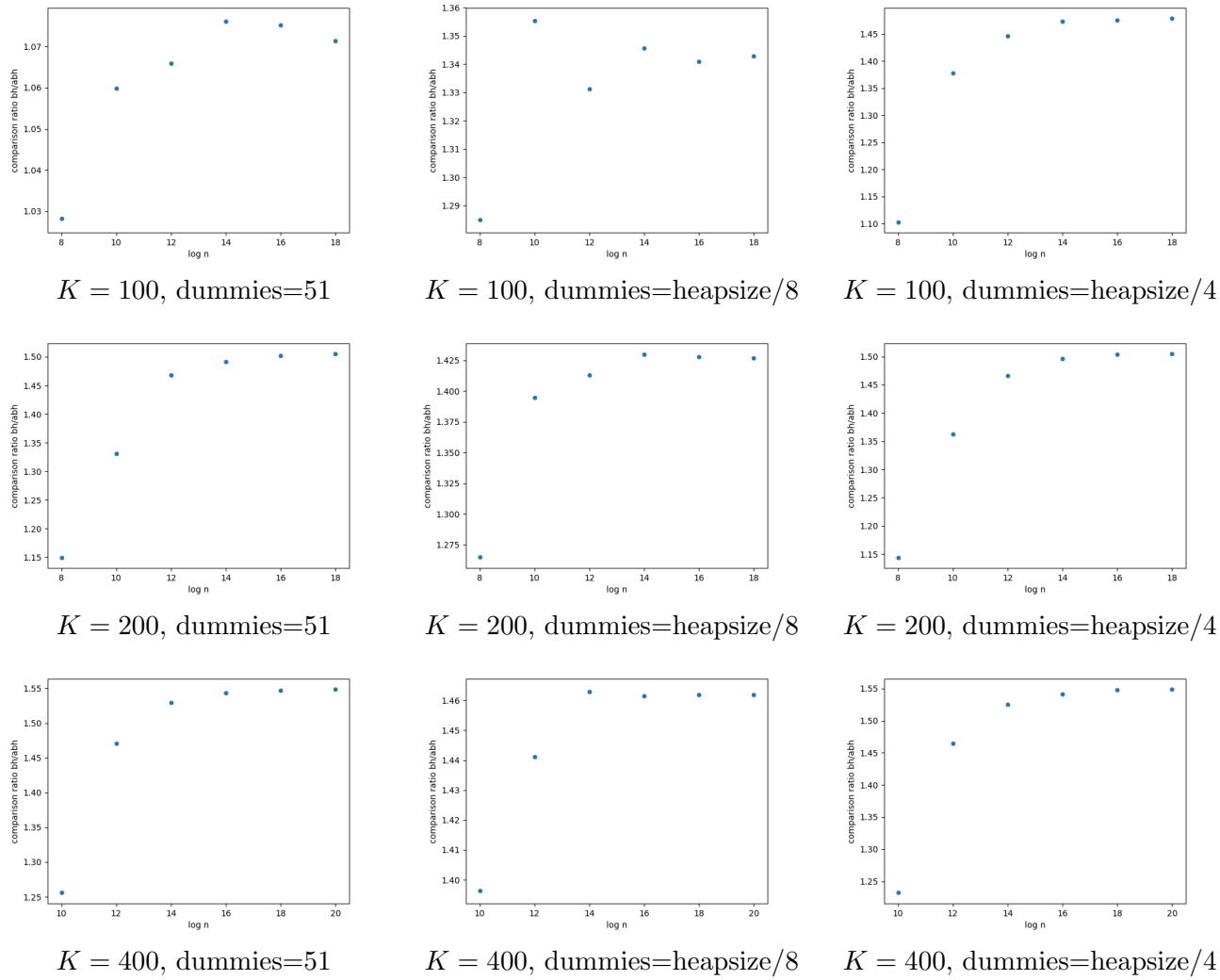
Of course, Python is not the best language for benchmarks and performance tests. Additionally, we didn't want to measure directly the running time of training the model, as the model is treated as a black-box to our analysis.

We decided to compare the two implementations on the number of comparisons performed between two elements inside the data structure. In some real-world applications, the elements stored in binary heaps are complex and thus comparing them to satisfy the heap property is an expensive operation.

8.2 K-way Merge

K-way merge is used to merge K sorted lists into one list. This is useful in the case when the input list is too large to sort in the single run. So, the list is split into K sublists, which are sorted in parallel and then merged using K-way merge.

K-way merge algorithm first initializes a heap of maximum element from each sublist. It also keeps a pointer for each sublist. The element is extracted from the heap is the maximum among all the remaining elements. Then the algorithm decreases the sublist pointer corresponding to the extracted element, and inserts this new element to this heap. The algorithm continues to do this until the heap is empty.



$K = 100$, dummies=51 $K = 100$, dummies=heapsize/8 $K = 100$, dummies=heapsize/4

$K = 200$, dummies=51 $K = 200$, dummies=heapsize/8 $K = 200$, dummies=heapsize/4

$K = 400$, dummies=51 $K = 400$, dummies=heapsize/8 $K = 400$, dummies=heapsize/4

Figure 2: Comparison ratio for number of comparisons made by augmented heap vs classical binary heap. For instance, a ratio of 1.55 means augmented heap did 55% less comparisons than the binary heap. $\log n$ is 2 based logarithm of the number of elements in the list.

In our experiments, we used linear regression [scikit19] for scikit learn as the machine learning

model. We varied K and number of dummies. It seems the algorithm did up to 55% less comparisons than the classical heap as K increases.

8.3 A^* Search Algorithm

One application that involves the use of binary heaps is the A^* Search Algorithm. The A^* search algorithm is essentially a modified Breadth-First Search algorithm that prioritizes the exploration of states that have the least value according to some metric.

The binary heap is used to keep track of the states to be explored and "pop" the state with the min/max value at the beginning of each round efficiently. As the binary heap has a central role in this algorithm, this seemed like a great real-world application to test.

Algorithm 1 A^* Search Algorithm

```
1: procedure A-starSearch(s, t)                ▷ start state, end state
2:     bh ← BinaryHeap({s})                    ▷ bh is initialized containing only the starting state
3:     while bh.size() > 0 do                  ▷ While the binary heap is not empty
4:         c ← bh.ExtractMax()                 ▷ Get and remove the current state with maximum metric
5:         for c' : c → c' do                  ▷ Iterate over all possible reachable states from c
6:             bh.Insert(c')
7:         end for
8:     end while
9: end procedure
```

For our experiments, we used the A^* search algorithm to solve the general version of a game called 15-Puzzle, which involves arranging $N^2 - 1$ tiles in an $N \times N$ board by sliding. The original game has $N = 4$ and hence the number 15 in the name.

For our experiments the heuristic metric we tried and worked best in terms of number of visited states before termination was of the form

$$h(s) = -\left(f(s) + g(s)\right)$$

For a given state of the board s, $h(s)$ represents the metric of the heuristic and the binary heap will have the ones with the highest value of the metric first. $f(s)$ will represent the "distance" of the current state from the ending state and $g(s)$ will be the number of steps required to go from the start state to s. More specifically, if the tuple (r_x, c_x) represents the row and column (0-based indexing) of the number x for $1 \leq x \leq N^2 - 1$, then the "distance" of state from the ending state is

$$f(s) = \sum_{1 \leq x \leq N^2 - 1} \left| r_x - \left\lfloor \frac{x-1}{N} \right\rfloor \right| + |c_x - (x-1) \bmod N|$$

For this experiment, our model was a simple Linear Regression model from python's package scikit-learn [scikit19]. We experimented with other models that gave similar results. This is probably due to the simple structure and noisy nature of the position of the dummy nodes. More specifically, we ideally would like to learn a smooth version of the CDF of the distribution of our dummy nodes and to avoid overfitting to the noise of our training data, we want our model to be simple. Our hyperparemeter was the fraction of elements of the heap that were dummy nodes. The less dummy

Figure 3: The goal of the game is to sort the numbers in a row-by-row fashion by sliding the tiles and positioning the empty tile at the lower right corner.

N	ABH Comparisons	BH Comparisons	Ratio
3	136	155	1.34
3	256	345	**1.35**
3	26	25	.96
4	259	280	1.08
4	392	425	**1.09**
4	229	192	.84
5	127	583	**4.55**
5	2728	2642	0.97
5	830	1016	1.22

Table 1: Results of experiments with creating an Augmented Binary Heap with 6.25% dummies

nodes there are, there is a greater reduction of comparisons of an augmented insert compared to a normal one. This is because by our algorithm, the dummy nodes stay at higher levels and thus the average number of levels "skipped" is higher. On the other side, having a lower number of dummy nodes, means that we can accommodate a lower number of insertions before running out of dummies and needing to rebuild our whole heap. Rebuilding the heap is an expensive operation and this there is a tradeoff that creates the need to tune the fraction of dummy nodes in the heap.

We created multiple starting positions for the game by randomly swapping tiles from the terminal state. Then we let the algorithm run, one of them using the original binary heap and the other one using the Augmented Binary Heap. Some representative results can be found in Tables 1 and 2.

As we can see from these results, it seems that keeping 3.125% of the heap as dummies gives the best results, with our Augmented Binary Heap being consistently more efficient in terms of comparisons with the original Binary Heap.

N	ABH Comparisons	BH Comparisons	Ratio
3	640	804	1.25
3	230	305	**1.33**
3	28	29	1.03
4	2101	2754	1.32
4	2931	3943	**1.35**
4	154	162	1.05
5	300	308	1.03
5	123	146	1.19
5	745	898	**1.20**

Table 2: Results of experiments with creating an Augmented Binary Heap with 3.125% dummies

9 Conclusion & Further Work

In this paper, we have presented a novel augmentation of the binary heap that uses machine learning methods to optimize and improve the runtime of its insert and extract operations. Our experiments in this paper were based on a Python implementation of our data structure and thus comparing total execution time would not be informative due to multiple layers of abstraction. This is why to compare the two different methods we used the number of comparisons performed between two items in the binary heap.

In some real-world applications of binary heaps, the items stored in binary heaps are not primitive types, but more complex data structures. Examples include packets that arrive to a router that implements a load balancing policy or the state of a problem that is being solved using a heuristic search algorithm (such as A^*).

Thus, our claim is that if the comparisons between the items of the binary heap is not a trivial operation and the number of items is sufficiently high, then our insert and extract operations could outperform the respective operations of the original binary heap.

A more detailed experiment setup would be to code our data structure in a low-level language such as C/C++, along with an efficient implementation of a machine learning model in the same language. Then measuring total time for both data structures would be an even more informative metric. This is what the authors plan on doing for future work.

Regardless, our Augmented Binary Heap does satisfy the three key properties introduced by [PSK18]. As we can see our data structure is

- Independent. It makes no assumptions regarding the type of the predictor or its error rate.
- Consistency. Its performance can improve dramatically with good predictions.
- Robustness. Its performance is always upper-bounded even with a predictor acting as an adversary.

10 Acknowledgements & Previous Work

When trying to find a subject to work on for this project, we met with Prof. Kraska and discussed ideas. We first want to thank Prof. Kraska for his time, ideas and constructive feedback. One of the problems he was working on, was to learning-augment the insert of B-trees, as it is a quite slow operation in practice.

We decided to focus on another data structure with similar usage and structure, but was easier to implement and test: the binary heap. The first idea proposed by Prof. Kraska when it comes to the Binary Heap, was the use of an insertion table with empty spaces according to the learned distribution of the input. This way, the insertions would not need to move large portions of the array to create empty space for the inserted element. Although a good idea, we found that this idea was hard to guarantee Robustness, meaning that the data structure has a bounded worst-case performance. Thus we decided to modify the binary heap structure directly.

The authors are aware that the techniques developed in this paper can be used to develop other Learning-Augmented Data Structures, such as B-trees and other Balanced Trees.

References

[KBC18] Tim Kraska and Alex Beutel and Ed H Chi and Jeffrey Dean and Neoklis Polyzotis. The case for learned index structures Proceedings of the *2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[PSK18] Manish Purohit, Zoya Svitkina, Ravi Kumar. Improving Online Algorithms via ML Predictions. Advances in *Neural Information Processing Systems 31*. NeurIPS, 2018.

[scikit19] Joris Van den Bossche, Loc Estve, Thomas J Fan. Logistic Regression `sklearn.linear_model.LogisticRegression`