ON THE POWER OF LEARNING-AUGMENTED SEARCH TREES

Anonymous authors
Paper under double-blind review

ABSTRACT

We study learning-augmented binary search trees (BSTs) via Treaps with carefully designed priorities. The result is a simple search tree in which the depth of each item x is determined by its predicted weight w_x. Specifically, each item x is assigned a composite priority of $-\lfloor \log\log(1/w_x) \rfloor + U(0,1)$ where $U(0,1)$ is the uniform random variable. By choosing w_x as the relative frequency of x, the resulting search trees achieve static optimality. This approach generalizes the recent learning-augmented BSTs [Lin-Luo-Woodruff ICML'22], which only work for Zipfian distributions, by extending them to arbitrary input distributions. Furthermore, we demonstrate that our method can be generalized to a B-Tree data structure using the B-Treap approach [Golovin ICALP'09]. Our search trees are also capable of leveraging localities in the access sequence through online self-reorganization, thereby achieving the working-set property. Additionally, they are robust to prediction errors and support dynamic operations, such as insertions, deletions, and prediction updates. We complement our analysis with an empirical study, demonstrating that our method outperforms prior work and classic data structures.

1 INTRODUCTION

The development of machine learning has sparked significant interest in its potential to enhance traditional data structures. First proposed by Kraska et al. (2018), the notion of *learned index* has gained much attention since then (Kraska et al., 2018; Ding et al., 2020; Ferragina & Vinciguerra, 2020). Algorithms with predictions have also been developed for an increasingly wide range of problems, including shortest path (Chen et al., 2022a), network flow (Polak & Zub, 2022; Lavastida et al., 2020), matching (Chen et al., 2022a; Dinitz et al., 2021; Chen & Indyk, 2021), spanning tree (Erlebach et al., 2022), and triangles/cycles counting (Chen et al., 2022b), with the goal of obtaining algorithms that get near-optimal performances when the predictions are good, but also recover prediction-less worst-case behavior when predictions have large errors (Mitzenmacher & Vassilvitskii, 2020). The problem of using learning to accelerate search trees, as in the original learned index question, has been widely studied in the field of data structures, focusing on developing data structures optimal to the input sequence. Mehlhorn (1975a) showed that a nearly optimal static tree can be constructed in linear time when estimates of key frequencies are provided. Extensive work on this topic culminated in the study of dynamic optimality. Tango trees (Demaine et al., 2007) achieve a competitive ratio of $O(\log\log n)$ while splay trees (Sleator & Tarjan, 1985) and Greedy BSTs (Lucas, 1988; Munro, 2000; Demaine et al., 2009) are conjectured to be within constant factors of optimal.

Treaps, introduced by Aragon & Seidel (1989), is a class of balanced BSTs distinguished by its use of randomization to maintain a low tree height. Each node in a Treap is assigned not only a key but also a randomly generated priority value. This design enables Treaps to satisfy the *Heap* property, ensuring that every node has a lower priority than its parent. In general, Treaps use randomness to ensure a low height instead of balancing the tree preemptively. More recently, Lin et al. (2022) introduced a learning-augmented Treap, demonstrating stronger guarantees compared to traditional Treaps. However, it relies on the strong assumption of the Zipfian distribution.

Inspired by this line of work, our research is driven by a series of critical questions.

- Whether a more general learning-augmented BST exists and achieves static optimality?
- Can such BST also obtain good guarantees under the dynamic settings?
- Are they robust to the errors caused by the prediction oracles?

This paper addresses the questions affirmatively by developing new learning-augmented Treaps with carefully designed priority scores, which are applicable to arbitrary input distributions in both static and dynamic settings. In the static setting, we show that our learning-augmented Treaps are within a constant factor of the static optimal cost when incorporating a prediction oracle for the frequency of each item. The proposed Treaps are robust to predicted errors, where the additional cost induced by the inaccurate prediction grows linearly with the KL divergence between the relative frequency and its estimation. For the dynamic setting, where the trees can undergo changes after each access,

we show that given a prediction oracle for the time interval until the next access, our data structure can achieve the working-set bound. This bound can be viewed as a strengthening of the static optimality bound that takes temporal locality of keys into account. Such dynamic BSTs are robust to the prediction oracle as well, where the performance degrades smoothly with the mean absolute error between the logarithm of the generated priorities and the ground truth priorities. Additionally, under the external memory model, our learning-augmented BST can be naturally extended to a B-Tree version via B-Treaps. Experimental results demonstrate that the proposed Treap outperforms other data structures, even when the predictions are inaccurate.

1.1 Overview

Learning-Augmented Treaps via Composite Priority Functions. The Treap is a tree-balancing mechanism initially designed around randomized priorities (Aragon & Seidel, 1989). When the priorities are assigned randomly, the resulting tree is balanced with high probability. Intuitively, this is because the root is likely to be picked among the middle elements. However, if some node is accessed very frequently (e.g. 10% of the time), it's natural to assign it a larger priority. Therefore, setting the priority to be a function of access frequencies, as in Lin et al. (2022), is a natural way to obtain an algorithm more efficient on more skewed access patterns. However, when the priority is set exactly as the access frequency, some nodes would have super-logarithmic depth: if each element i is accessed i times, setting priority exactly as the frequencies results in a path of size n. The total time for processing this access sequence of size $O(n^2)$ degrades to $\Omega(n^3)$. Partly as a result of this, the analysis in Lin et al. (2022) was limited only to when frequencies are under the Zipfian distribution.

Building upon these ideas, we introduce a composite priority function, a mixture of the randomized priority function from Aragon & Seidel (1989) and the frequency-based priority function from Lin et al. (2022). This takes advantage of the balance coming from the randomness and manages to work without the strong assumption from Lin et al. (2022). Specifically, we show in Theorem 2.4 that by setting the composite priority function to be

$$-\left\lfloor \log\log \frac{1}{w_x} \right\rfloor + U\left(0,1\right), \tag{1}$$

the expected depth of node x is $O(\log(1/w_x))$. The predicted score $w_x \in (0,1)$, for instance, can be set as the relative frequency or probability of each item.

Our Treap-based scheme generalizes to B-Trees, where each node has B instead of 2 children. These trees are highly important in external memory systems due to the behavior of cache performances: accessing a block of B entries has a cost comparable to the cost of accessing $O(1)$ entries. By combining the B-Treaps by Golovin (2009) with the composite priorities, we introduce a new learning-augmented B-Tree that achieves similar bounds under the *External Memory Model*. We show in Theorem 3.1 that for any weights over elements $\boldsymbol{w}$, by setting the priority to

$$-\lfloor \log_2 \log_B \frac{1}{w_x} \rfloor + U(0,1), \tag{2}$$

the expected depth of node x is $O(\log_B(1/w_x))$. It is natural to see that our proposed data structures unify BSTs and B-Trees. For simplicity, we provide the results of B-Trees in the remaining content.

Static Optimality of Learning-Augmented Search Trees. We can construct static optimal B-Trees if we set $\boldsymbol{w}$ to be the marginal distribution of elements in the access sequence. That is, if we know the frequencies $f_1, f_2, \ldots, f_n$ of each element that appears in the access sequence, and let $m = \sum_i f_i$ to be the length of the access sequence, then we set the score $w_x = f_x/m$ in Equation (2) and the corresponding B-Tree has a total access cost that achieves the static optimality

$$\sum_{i\in[n]} f_i \log_B \frac{m}{f_i}.$$

Dynamic Learning-Augmented Search Trees. We also consider the dynamic setting in which we continually update the priorities of a subset of items along with the sequence access. Rather than a fixed priority for each item, we allow the priorities to change as keys get accessed. The setting has a wide range of applications in the real world. For instance, consider accessing data in a relational database. A sequence of access will likely access related items one after another. So even if the entries themselves get accessed with fixed frequencies, the distribution of the next item to be accessed can be highly dependent on the set of recently accessed items. Consider the access sequence

$$4, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5, 1, 4$$

versus the access sequence

$$5, 2, 4, 2, 1, 4, 4, 5, \mathbf{3}, \mathbf{3}, \mathbf{3}, 4, 5, 4, 2$$

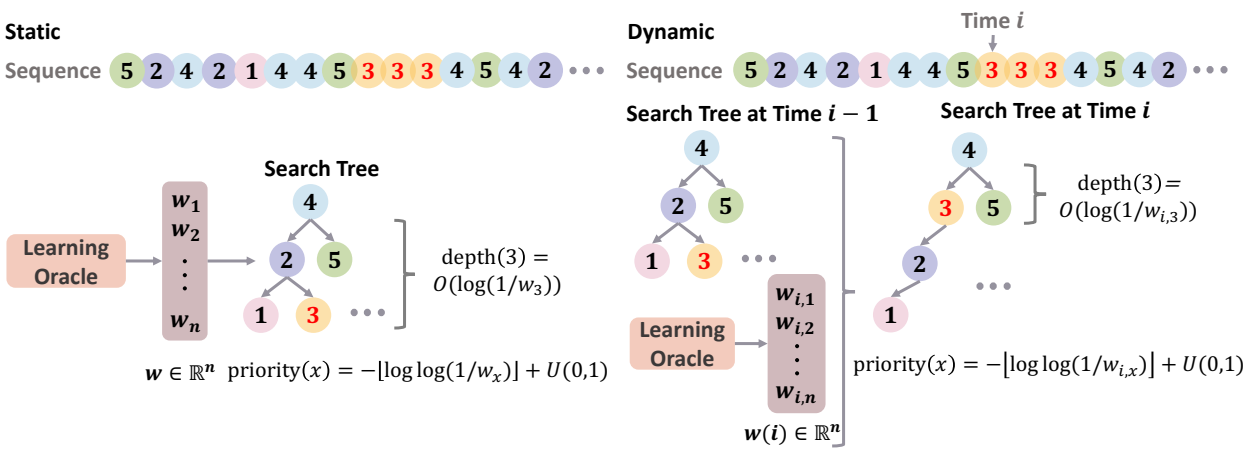

Figure 1: Sketch for static and dynamic learning augmented search trees. Since item 3 has a higher frequency around time i, the dynamic search trees adjust the priority accordingly.

In both sequences, the item 4 is accessed the most frequently. So input dependent search trees should place it near the root. However, in the second sequence, the item 3 is accessed three consecutive times around the middle. An algorithm that's allowed to modify the tree dynamically can then modify the tree to place 3 closer to root during those calls. An illustration of this is in Figure 1. Note that we pay cost both when accessing the items and updating the trees. Hence, there is a trade-off between the costs of updating items' scores and the benefits of time-varying scores.

We study ways of designing composite priorities that cause this access cost to match known sequence-dependent access costs of binary trees (and their natural generalizations to B-Trees). Here, we focus on the *working-set bound*, which says that the cost of accessing an item should be, at most, the logarithm of the number of distinct items until they get accessed again. To obtain this bound, we propose a new composite priority named *working-set priority*, based on the number of distinct elements between two occurrences of the same item accessed at step i. We give the guarantees for the dynamic Treaps with the working-set priority in Theorem 4.4. The dynamic search Treaps further demonstrate the power of learning scores from data. While we have more data, we can quantify the dynamic environment in a more accurate way and thus improve the efficiency of the data structure.

Robustness to Prediction Inaccuracy. Finally, we show the robustness of our data structures with inaccuracies in prediction oracles. In the static case, we can directly relate the overhead of having inaccurate frequency predictions to the KL divergences between the true relative frequencies p_x and their estimates q_x. This is because our composite priority can take any estimate. So plugging in the estimates q_x gives the overall access cost

$$m \cdot \sum_x p_x \log_2 \left(\frac{1}{q_x} \right),$$

which is exactly the cross entropy between $\boldsymbol{p}$ and $\boldsymbol{q}$. On the other hand, the KL divergence between $\boldsymbol{p}$ and $\boldsymbol{q}$ is exactly the cross entropy minus the entropy of $\boldsymbol{p}$. So we get that the overhead of building the tree using noisy estimators $\boldsymbol{q}$ instead of the true frequencies $\boldsymbol{p}$ is exactly m times the KL divergence between $\boldsymbol{p}$ and $\boldsymbol{q}$. We formalize the argument above in Section 2.3. We also achieve robustness results in the dynamic setting in Appendix C.2.

In all, our contributions can be summarized as follows:

- We introduce composite priorities that integrate learned advice into Treaps. The BSTs and B-Trees constructed via these priorities are within constants of the static optimal ones for arbitrary distributions (Section 2, Section 3).
- When allowing updating trees along with accessing items, we design a working-set priority function, and the corresponding Treaps with composite priorities can achieve the working-set bound (Section 4).
- Both static and dynamic learning-augmented search trees are robust to predictions (Section 2.3, Appendix C.2).
- Our experiments show favorable performance compared to prior work (Section 5).

1.2 Related Work

In recent years, there has been a surge of interest in integrating machine learning models into algorithm designs. A new field called *Algorithms with Predictions* (Mitzenmacher & Vassilvitskii, 2020) has garnered considerable attention,

particularly in the use of machine learning models to predict input patterns to enhance performance. Examples of this approach include online graph algorithms with predictions (Azar et al., 2022), improved hashing-based methods such as Count-Min (Cormode & Muthukrishnan, 2005), and learning-augmented k-means clustering (Ergun et al., 2021). Practical oracles for predicting desired properties, such as predicting item frequencies in a data stream, have been demonstrated empirically (Hsu et al., 2019; Jiang et al., 2020).

Capitalizing on the existence of oracles that predict the properties of upcoming accesses, researchers are now developing more efficient learning-augmented data structures. Index structures in database management systems are one significant application of learning-augmented data structures. One key challenge in this domain is to create search algorithms and data structures that are efficient and adaptive to data whose nature changes over time. This has spurred interest in incorporating machine learning techniques to improve traditional search tree performance.

The first study on learned index structures (Kraska et al., 2018) used deep-learning models to predict the position or existence of records as an alternative to the traditional B-Tree or hash index. However, this study focused only on the static case. Subsequent research (Ferragina & Vinciguerra, 2020; Ding et al., 2020; Wu et al., 2021) introduced dynamic learned index structures with provably efficient time and space upper bounds for updates in the worst case. These structures outperformed traditional B-Trees in practice, but their theoretical guarantees were often trivial, with no clear connection between prediction quality and performance. More recently, Lin et al. (2022) proposed a learning-augmented BST via Treaps that works under the Zipfian distribution.

Other related work on BSTs analyses and B-Trees under the external memory model is included in Appendix A.

2 Learning-Augmented Binary Search Trees

In this section, we show that the widely taught Treap data structure can, with small modifications, achieve the static optimality conditions sought after in previous studies of learned index structures (Lin et al., 2022; Hsu et al., 2019). We start with definitions and basic properties of Treaps.

Definition 2.1 (Treap (Aragon & Seidel, 1989))**.** *Let T be a BST over $[n]$ and $\mathsf{priority} \in \mathbb{R}^n$ be a priority assignment on $[n]$. We say $(T, \mathsf{priority})$ is a* Treap *if $\mathsf{priority}_x \le \mathsf{priority}_y$ whenever x is a descendent of y in T.*

Given a priority assignment priority, one can construct a BST T such that $(T, \mathsf{priority})$ is a Treap as follows. Take any $x^* \in \arg\max_x \mathsf{priority}_x$ and build Treaps on $[1, x^* - 1]$ and $[x^* + 1, n]$ recursively using priority. Then, we just make x^* the parent of both Treaps. Notice that if $\mathsf{priority}_x$'s are distinct, the resulting Treap is unique.

Observation 1. *Let $\mathsf{priority} \in \mathbb{R}^n$, which assigns each item x to a unique priority. There is a unique BST T such that $(T, \mathsf{priority})$ is a Treap.*

From now on, we always assume that priority has distinct values. Therefore, when priority is defined from the context, the term *Treap* refers to the unique BST T. For each node $x \in [n]$, we use $\mathsf{depth}(x)$ to denote its depth in T, i.e., the number of vertices on the path from the root to x.

Given any two items $x, y \in [n]$, one can determine whether x is an ancestor of y in a Treap without traversing the tree.

Observation 2. *Given any $x, y \in [n]$, x is an ancestor of y if and only if $\mathsf{priority}_x = \max_{z \in [x,y]} \mathsf{priority}_z$.*

Classical results from Aragon & Seidel (1989) state that if the priorities are randomly assigned, the depth of the Treap cannot be too large. Also, Treaps can be made dynamic and support operations such as insertions and deletions.

Lemma 2.2 (Aragon & Seidel (1989))**.** *Let $U(0,1)$ be the uniform distribution over the real interval $[0,1]$. If $\mathsf{priority} \sim U(0,1)^n$, each Treap node x has depth $\Theta(\log_2 n)$ with high probability.*

Proof. Notice that $\mathsf{depth}(x)$, the depth of item x in the Treap, is the number of ancestors of x in the Treap. Linearity of expectation yields

$$\mathbb{E}[\mathsf{depth}(x)] = \sum_{y \in [n]} \mathbb{E}[1 \text{ if } y \text{ is an ancestor of } x \text{ or } 0 \text{ otherwise}] = \sum_{y \in [n]} \Pr\left(y \text{ is an ancestor of } x\right)$$

$$= \sum_{y \in [n]} \Pr\left(\mathsf{priority}_y = \max_{z \in [x,y]} \mathsf{priority}_z\right) = \sum_{y \in [n]} \frac{1}{|x - y + 1|} = \Theta(\log_2 n).$$

□

Lemma 2.3 (Aragon & Seidel (1989))**.** *Given a Treap T and some item $x \in [n]$, x can be inserted to or deleted from T in $O(\mathsf{depth}(x))$-time.*

2.1 LEARNING-AUGMENTED TREAPS

In this section, we present the construction of composite priorities and prove the following theorem.

Theorem 2.4 (Learning-Augmented Treap via Composite Priorities)**.** *Denote $\boldsymbol{w} = (w_1, \cdots, w_n) \in (0,1)^n$ as a score associated with each item in $[n]$ such that $\|\boldsymbol{w}\|_1 = O(1)$. Consider the following priority assignment of each item:*

$$\mathsf{priority}_x \stackrel{\text{def}}{=} -\left\lfloor \log_2 \log_2 \frac{1}{w_x} \right\rfloor + \delta_x, \tag{3}$$

where δ_x is drawn independently uniformly from $(0,1)$. The expected depth of any item $x \in [n]$ is $O(\log_2(1/w_x))$.

Moreover, our proposed learning-augmented treap supports efficient updates, where we can use rotations to do insertions, deletions, and weight changes. The following corollary follows naturally by Theorem 2.4 and Lemma 2.3.

Corollary 2.5. *The data structure supports insertions and deletions naturally. Suppose the score of some node x changes from w to w' and a pointer to the node is given, the Treap can be maintained with $O(|\log_2(w'/w)|)$ rotations in expectation.*

Proof Idea. Note that the priority in Equation (3) consists of two terms. We define x's tier as $\tau_x := \left\lfloor \log_2 \log_2 \frac{1}{w_x} \right\rfloor = -\lfloor \mathsf{priority}_x \rfloor$. Let $S_t = \{x \in [n] \mid \tau_x = t\}$ be the number of items whose tiers are equal t. We assume wlog that $\tau_x \geq 0$ for any x. Otherwise, $\tau_x < 0$ implies $w_x = \Omega(1)$, which can hold for only a constant number of items. So, we can always put them at the top of the Treap, which increases the depths of other items by a constant.

The expected depth of x is the number of its ancestors. We show in Lemma 2.6 that the number of items at tier t is bounded by $|S_t| = 2^{O(2^t)}$. Furthermore, for each tier, the ties are broken randomly due to the random offset $\delta_x \sim U(0,1)$. Then, as we show in Lemma 2.7, any item has $O(\log_2 |S_t|) = O(2^t)$ ancestors with tier t in expectation. Therefore, the expected depth $\mathbb{E}[\mathsf{depth}(x)]$ can be bound by $O(2^0 + 2^1 + \ldots + 2^{\tau_x}) = O(2^{\tau_x}) = O(\log_2(1/w_x))$.

Lemma 2.6. *For any non-negative integer $t \geq 0$, $|S_t| = 2^{O(2^t)}$.*

Proof. Observe that $x \in S_t$ if and only if

$$t \leq \log_2 \log_2(1/w_x) < t+1, \text{ and } 2^{2^t} \leq \frac{1}{w_x} < 2^{2^{t+1}}.$$

Since the total score $\|\boldsymbol{w}\|_1 = O(1)$, there are only $\mathrm{poly}(2^{2^{t+1}}) = 2^{O(2^t)}$ such items. □

Next, we bound the expected number of ancestors of item x in every S_t such that $t \leq \tau_x$.

Lemma 2.7. *Let $x \in [n]$ be any item and $t \leq \tau_x$ be a non-negative integer. The expected number of ancestors of x in S_t is at most $O(\log_2 |S_t|)$.*

Proof. First, we show that any $y \in S_t$ is an ancestor of x with probability no more than $1/|S_t \cap [x,y]|$. Observation 2 says that y must have the largest priority among items $[x,y]$. Thus, a necessary condition for y being x's ancestor is that y has the largest priority among items in $S_t \cap [x,y]$. However, priorities of items in $S_t \cap [x,y]$ are i.i.d. random variables of the form $-t + U(0,1)$. Thus, the probability that $\mathsf{priority}_y$ is the largest among them is $1/|S_t \cap [x,y]|$.

Now, we can bound the expected number of ancestors of x in S_t as follows:

$$\mathbb{E}[\text{number of ancestors of } x \text{ in } S_t] = \sum_{y \in S_t} \Pr(y \text{ is an ancestor of } x) \leq \sum_{y \in S_t} \frac{1}{|S_t \cap [x,y]|} \leq 2 \cdot \sum_{u=1}^{|S_t|} \frac{1}{u} = O(\log_2 |S_t|),$$

where the second inequality comes from the fact that for a fixed value of u, there are at most two items $y \in S_t$ with $|S_t \cap [x,y]| = u$ (one with $y \leq x$, the other with $y > x$). □

Now we are ready to prove Theorem 2.4.

Proof of Theorem 2.4. By Lemma 2.7 and Lemma 2.6, the expected depth of x can be bounded by

$$\mathbb{E}[\mathsf{depth}(x)] = \sum_{t=0}^{\tau_x} \mathbb{E}[\text{number of ancestors of } x \text{ in } S_t] \leq O\left(\sum_{t=0}^{\tau_x} \log_2 |S_t|\right) \leq O\left(\sum_{t=0}^{\tau_x} 2^t\right) \leq O\left(2^{\tau_x}\right).$$

We conclude the proof by observing that

$$\tau_x \leq \log_2 \log_2 \frac{1}{w_x} \leq \tau_x + 1, \text{ and } 2^{\tau_x} \leq \log_2 \frac{1}{w_x}.$$

□

2.2 STATIC OPTIMALITY

We present a priority assignment for constructing statically optimal Treaps given item frequencies. Given any access sequence $\boldsymbol{X} = (x(1), \ldots, x(m))$, we define f_x for any item x, to be its *frequency* in $\boldsymbol{X}$, i.e. $f_x := |\{i \in [m] \mid x(i) = x\}|, x \in [n]$. For simplicity, we assume that every item is accessed at least once, i.e., $f_x \geq 1, x \in [n]$. We prove the following result as a simple application of Theorem 2.4:

Theorem 2.8 (Static Optimality)**.** *For any item $x \in [n]$, we set its priority as*

$$\mathsf{priority}_x := -\left\lfloor \log_2 \log_2 \frac{m}{f_x} \right\rfloor + \delta_x, \delta_x \sim U(0,1).$$

In the corresponding Treap, each node x has expected depth $O(\log_2(m/f_x))$. Therefore, the total time for processing the access sequence is $O(\sum_x f_x \log_2(m/f_x))$, which matches the performance of the optimal static BSTs up to a constant factor.

Proof. Given item frequencies , we define the following $\boldsymbol{w}$ assignment:

$$w_x := \frac{f_x}{m},\ x \in [n]. \tag{4}$$

One can verify that $\|\boldsymbol{w}\|_1 = O(1)$. By Theorem 2.4, the expected depth of each item x is $O(\log_2(m/f_x))$. □

2.3 ROBUSTNESS GUARANTEES

In practice, one could only estimate $q_x \approx p_x = f_x/m, x \in [n]$. A natural question arises: how does the estimation error affect the performance? In this section, we analyze the drawbacks in performance given the estimation errors. As a result, we will show that our Learning-Augmented Treaps are robust against noise and errors.

For each item $x \in [n]$, define $p_x = f_x/m$ to be the relative frequency of item x. One can view $\boldsymbol{p}$ as a probability distribution over $[n]$ such that $\boldsymbol{p}(x) = p_x$. Then we can restate the expected depth of each item in Theorem 2.8 using the notion of entropy. We define the entropy as follows and state the corollary in Corollary 2.10.

Definition 2.9 (Entropy)**.** *Given a probability distribution $\boldsymbol{p}$ over $[n]$, define its* Entropy *as* $\mathrm{Ent}(\boldsymbol{p}) := \sum_x p_x \log_2(1/p_x) = \mathbb{E}_{x\sim\boldsymbol{p}}[\log_2(1/p_x)]$.

Corollary 2.10. *In Theorem 2.8, the expected depth of each item x is $O(\log_2(1/p_x))$ and the expected total cost is $O(m \cdot \mathrm{Ent}(\boldsymbol{p}))$, where $\mathrm{Ent}(\boldsymbol{p}) = \sum_x p_x \log_2(1/p_x)$ measures the entropy of the distribution $\boldsymbol{p}$.*

Now we consider the case when we cannot access the relative frequency p_x. Instead, we are given p_x's estimator, q_x, and construct the data-augmented BST with q_x. Similarly, we view $\boldsymbol{q}$ as a data distribution over $[n]$ such that $\boldsymbol{q}(x) = q_x$. Then we show that the total access of the treap built with q_x equals the total access number m times the cross entropy of $\boldsymbol{p}$ and $\boldsymbol{q}$ in Theorem 2.13. We start with some definitions.

Definition 2.11 (Cross Entropy)**.** *Given two distributions $\boldsymbol{p}, \boldsymbol{q}$ over $[n]$, define its* Cross Entropy *as* $\mathrm{Ent}(\boldsymbol{p}, \boldsymbol{q}) := \sum_x p_x \log_2(1/q_x) = \mathbb{E}_{x\sim\boldsymbol{p}}[\log_2(1/q_x)]$.

Definition 2.12 (KL Divergence)**.** *Given two distributions $\boldsymbol{p}, \boldsymbol{q}$ over $[n]$, define its* KL Divergence *as* $D_{\mathrm{KL}}(\boldsymbol{p}, \boldsymbol{q}) = \mathrm{Ent}(\boldsymbol{p}, \boldsymbol{q}) - \mathrm{Ent}(\boldsymbol{p}) = \sum_x p_x \log_2(p_x/q_x)$.

Next we analyze the run time given frequency estimations $\boldsymbol{q}$.

Theorem 2.13. *Given a distribution $\boldsymbol{q}$, an estimate of the true relative frequencies distribution $\boldsymbol{p}$. For any item $x \in [n]$, we draw a random number $\delta_x \sim U(0,1)$ and set its priority as*

$$\mathsf{priority}_x := -\left\lfloor \log_2 \log_2 \frac{1}{q_x} \right\rfloor + \delta_x.$$

In the corresponding Treap, each node x has expected depth $O(\log_2(1/q_x))$. Therefore, the total time for processing the access sequence is $O(m \cdot \mathrm{Ent}(\boldsymbol{p}, \boldsymbol{q}))$.

Proof. Define the weights $w_x = q_x$ for each item $x \in [n]$. Clearly, $\|\boldsymbol{w}\|_1 = 1$ and we can apply Theorem 2.4 to prove the bound on the expected depths. The total time for processing the access sequence is, by definition,

$$O\left(\sum_{x\in[n]} f_x \log_2 \frac{1}{q_x}\right) = O\left(m \cdot \sum_{x\in[n]} p_x \log_2 \frac{1}{q_x}\right) = O\left(m \cdot \mathrm{Ent}(\boldsymbol{p}, \boldsymbol{q})\right).$$

□

2.4 Analysis of Other Priority Assignments

In this section, we discuss two different priority assignments. For each assignment, we design an input distribution that results in a greater expected depth than the expected depth with our priority assignment stated in Theorem 2.8. We define the distribution $\boldsymbol{p}$ as $\boldsymbol{p}(x) = p_x = f_x/m, x \in [n]$. We use $f \gtrsim g$ to indicate that f is greater or equal to g up to a constant factor.

The first priority assignment is used in Lin et al. (2022). They assign priorities according to p_x entirely, i.e., $\mathsf{priority}_x = p_x, x \in [n]$. Assuming that items are ordered randomly, and $\boldsymbol{p}$ is a Zipfian distribution, Lin et al. (2022) shows *Static Optimality*. However, it does not generally hold, and the expected access cost could be $\Omega(n)$, while our data structure (Theorem 2.4) has a $O(\log_2 n)$ worst-case depth bound.

Theorem 2.14. *Consider the priority assignment that assigns the priority of each item to be* $\mathsf{priority}_x \coloneqq p_x, x \in [n]$. *There is a distribution* $\boldsymbol{p}$ *over* $[n]$ *such that the expected access time,* $\mathbb{E}_{x\sim\boldsymbol{p}}[\mathsf{depth}(x)] = \Omega(n)$.

Proof. We define for each item x, $p_x \coloneqq \frac{2(n-x+1)}{n(n+1)}$. One could easily verify that $\boldsymbol{p}$ is a distribution over $[n]$. In addition, the smaller the item x, the larger the priority $\mathsf{priority}_x$. Thus, by the definition of Treaps, item x has depth x. The expected access time of x sampled from $\boldsymbol{p}$ can be lower bounded as follows:

$$\begin{aligned}\mathbb{E}_{x\sim\boldsymbol{p}}[\mathsf{depth}(x)] &= \sum_{x\in[n]} p_x \cdot \mathsf{depth}(x) = \sum_{x\in[n]} \frac{2(n-x+1)}{n(n+1)} \cdot x = \frac{2}{n(n+1)} \sum_{x\in[n]} x(n-x+1) \\ &\gtrsim \frac{2}{n(n+1)} \cdot n^3 \gtrsim n.\end{aligned}$$

□

Next, we consider the priority assignment $\mathsf{priority}_x \coloneqq -\lfloor \log_2 1/p_x \rfloor + \delta_x, \delta_x \sim U(0,1)$.

Theorem 2.15. *Consider the following priority assignment that sets the priority of each node* x *as* $\mathsf{priority}_x \coloneqq -\lfloor \log_2 1/p_x \rfloor + \delta_x, \delta_x \sim U(0,1)$. *There is a distribution* $\boldsymbol{p}$ *over* $[n]$ *such that the expected access time,* $\mathbb{E}_{x\sim\boldsymbol{p}}[\mathsf{depth}(x)] = \Omega(\log_2^2 n)$.

Proof. We assume WLOG that n is an even power of 2. Define $K = \frac{1}{2}\log_2 n$. We partition $[n]$ into $K+1$ segments $S_1, \ldots, S_K, S_{K+1} \subseteq [n]$. For $i = 1, 2, \ldots, K$, we add $2^{1-i} \cdot n/K$ elements to S_i. Thus, S_1 has n/K elements, S_2 has $n/2K$, and S_K has $\sqrt{n}/K$ elements. The rest are moved to S_{K+1}.

Now, we can define the distribution $\boldsymbol{p}$. Elements in S_{K+1} have zero-mass. For $i = 1, 2, \ldots, K$, elements in S_i has probability mass $2^{i-1}/n$. One can directly verify that $\boldsymbol{p}$ is indeed a probability distribution over $[n]$.

In the Treap with the given priority assignment, S_i forms a subtree of expected height $\Omega(\log_2 n)$ since $|S_i| \geq n^{1/3}$ for any $i = 1, 2, \ldots, K$ (Lemma 2.2). In addition, every element of S_i passes through $S_{i+1}, S_{i+2}, \ldots, S_K$ on its way to the root since they have strictly larger priorities. Therefore, the expected depth of element $x \in S_i$ is $\Omega((K-i)\log_2 n)$. One can lower bound the expected access time (which is the expected depth) as:

$$\begin{aligned}\mathbb{E}_{x\sim\boldsymbol{p}}[\mathsf{depth}(x)] &\gtrsim \sum_{i=1}^{K} \sum_{x\in S_i} p_x \cdot (K-i) \cdot \log_2 n = \sum_{i=1}^{K} \boldsymbol{p}(S_i) \cdot (K-i) \cdot \log_2 n \\ &= \sum_{i=1}^{K} \frac{1}{K} \cdot (K-i) \cdot \log_2 n \gtrsim K \log_2 n \gtrsim \log_2^2 n,\end{aligned}$$

where we use $\boldsymbol{p}(S_i) = |S_i| \cdot 2^{i-1}/n = 1/K$ and $K = \Theta(\log_2 n)$. That is, the expected access time is at least $\Omega(\log_2^2 n)$. □

3 Learning-Augmented B-Trees

We now extend the ideas above, specifically the composite priority notions, to B-Trees in the *External Memory Model*. The main results are shown as follows. Full details are included in Appendix B. We show that the learning-augmented B-Treaps (Appendix B.1) obtain static optimality (Appendix B.2) and is robust to the noisy predicted scores (Appendix B.3).

Theorem 3.1 (Learning-Augmented B-Treap via Composite Priorities). *Denote $\boldsymbol{w} = (w_1, \cdots, w_n) \in (0,1)^n$ as a score associated with each element of $[n]$ such that $\|\boldsymbol{w}\|_1 = O(1)$ and a branching factor $B = \Omega(\ln^{1/(1-\alpha)} n)$, consider the following priority assignment scheme:*

$$\mathsf{priority}_x \coloneqq -\lfloor \log_2 \log_B \frac{1}{w_x} \rfloor + \delta_x,\ \delta_x \sim U(0,1).$$

There is a randomized data structure that maintains a B-Tree T^B over U such that

1. *Each item x has expected depth $O(\frac{1}{\alpha} \log_B(1/w_x))$.*
2. *Insertion or deletion of item x into/from T touches $O(\frac{1}{\alpha} \log_B(1/w_x))$ nodes in T^B in expectation.*
3. *Updating the weight of item x from w to w' touches $O(\frac{1}{\alpha} |\log_B(w'/w)|)$ nodes in T^B in expectation.*

In addition, if $B = O(n^{1/2-\delta})$ for some $\delta > 0$, all above performance guarantees hold with high probability $1 - \delta$. If we are given the frequency f_x for each x and set the priority as

$$\mathsf{priority}_x \coloneqq -\left\lfloor \log_2 \log_B \frac{m}{f_x} \right\rfloor + \delta_x, \delta_x \sim U(0,1).$$

Then the total access cost $O(\sum_x f_x \log_B(m/f_x))$ achieves the static optimality.

4 Dynamic Learning-Augmented Search Trees

In this section, we investigate the properties of dynamic search trees that permit modifications concurrent with sequence access. Prioritizing items that are anticipated to be accessed in the near future to reside at lower depths within the tree can significantly reduce access times. Nonetheless, updating the B-trees introduces additional costs. The overarching goal is to minimize the composite cost, which includes both the access operations across the entire sequence and the modifications to the B-trees. In the main content, we specifically concentrate on the study of locally dynamic B-trees, which are characterized by the restriction that tree modifications are limited solely to the adjustment of priorities for the items being accessed.

Here, we construct a dynamic learning-augmented B-trees that achieves the *working-set property*. In data structures, the working set is the collection of data that a program uses frequently over a given period. This concept is important because it helps us understand how a program interacts with memory and thus enables us to design more efficient data structures and algorithms. For example, if a program is sorting a list, the working set might be the elements of the list it is comparing and swapping right now. The size of the working set can affect how fast the program runs. A smaller working set can make the program run faster because it means the program doesn't need to reach out to slower parts of memory as often. In other words, if we know which parts of a data structure are used most, we can organize the data or even the memory in a way that makes accessing these parts faster, which can speed up the entire program.

We define the *working-set size* as the number of distinct items accessed between two consecutive accesses. Correspondingly, we design a time-varying score, *working-set score*, as the reciprocal of the square of one plus working-set size. We will show that the working-set score is locally changed and there exists a data structure that achieves the *working-set property*, which states that the time to access an element is a logarithm of its working-set size.

The formal definitions and the main theorems in this section are presented as follows. We include more general results for dynamic B-trees and omitted proofs in Appendix C.

Definition 4.1 (Previous and Next Access $\mathsf{prev}(i,x)$ and $\mathsf{next}(i,x)$). *Let $\mathsf{prev}(i,x)$ be the previous access of item x at or before time i, i.e, $\mathsf{prev}(i,x) \coloneqq \max\{i' \le i \mid x(i') = x\}$. Let $\mathsf{next}(i,x)$ to be the next access of item x after time i, i.e, $\mathsf{next}(i,x) \coloneqq \min\{i' > i \mid x(i') = x\}$.*

Definition 4.2 (Working-set Size $\mathsf{work}(i,x)$). *Define the* working-set size $\mathsf{work}(i,x)$ *to be the number of distinct items accessed between the previous access of item x at or before time i and the next access of item x after time i. That is,*

$$\mathsf{work}(i,x) \stackrel{\text{def}}{=} |\{x(\mathsf{prev}(i,x)+1), \cdots, x(\mathsf{next}(i,x))\}|.$$

If x does not appear after time i, we define $\mathsf{work}(i,x) \coloneqq n$.

Definition 4.3 (Working-set Score $\omega(i,x)$). *Define the time-varying score as the reciprocal of the square of one plus working-set size. That is,*

$$\omega(i,x) = \frac{1}{(1+\mathsf{work}(i,x))^2}$$

Theorem 4.4 (Dynamic Search Tree with Working-set Priority)**.** *With the working-set size* $\mathsf{work}(i,x)$ *known and the branching factor* $B = \Omega(\ln^{1.1} n)$*, there is a randomized data structure that maintains a B-tree* T^B *over* $[n]$ *with the priorities assigned as*

$$\mathsf{priority}(i,x) = -\lfloor \log_2 \log_B (1+\mathsf{work}(i,x))^2 \rfloor + U(0,1).$$

Upon accessing the item x *at time* i*, the expected depth of item* x *is* $O(\log_2(1+\mathsf{work}(i,x))$*. The expected total cost for processing the whole access sequence* $\boldsymbol{X}$ *is the following. The data structure satisfies the working-set property.*

$$\mathsf{cost}(\boldsymbol{X},\omega) = O\left(n\log_B n + \sum_{i=1}^{m} \log_B(1+\mathsf{work}(i,x))\right).$$

In particular, if $B = O\left(n^{1/2-\delta}\right)$ *for some* $\delta > 0$*, the guarantees hold with probability* $1-\delta$*.*

Remark. Consider two sequences with length m, $\boldsymbol{X}_1 = (1,2,\cdots,n,1,2,\cdots,n,\cdots,1,2,\cdots,n)$, $\boldsymbol{X}_2 = (1,1,\cdots,1,2,2,\cdots,2,\cdots,n,n,\cdots,n)$. Two sequences have the same total cost if we have a fixed score. However, X_2 should have less cost because of its repeated pattern. Given the frequency freq as a time-invariant priority, by Theorem 3.1, the optimal static costs are

$$\mathsf{cost}(\boldsymbol{X}_1,\mathsf{freq}) = \mathsf{cost}(\boldsymbol{X}_2,\mathsf{freq}) = O(m\log_2 n).$$

But for the dynamic BSTs, with the working-set score, we calculate both costs from Theorem 4.4 as

$$\mathsf{cost}(\boldsymbol{X}_1,\omega) = O(m\log_2(n+1)), \mathsf{cost}(\boldsymbol{X}_2,\omega) = O(n\log_2 n + m\log_2 3).$$

This means that our proposed priority can better capture the timing pattern of the sequence and thus can even do better than the optimal static setting.

Finally, we use the following theorem to show the robustness of the results when the scores are inaccurate.

Theorem 4.5 (Dynamic Search Tree with Working-set Priority)**.** *Given the predicted locally changed working-set score* $\widetilde{\omega}(i) \in (0,1)^n$ *satisfying* $\|\widetilde{\omega}(i)\|_1 = O(1)$*,* $\widetilde{\omega}_{i,j} \geq 1/\mathrm{poly}(n)$ *and the branching factor* $B = \Omega(\ln^{1.1} n)$*, there is a randomized data structure that maintains a B-Tree over the* n *keys such that the expected total cost for processing the whole access sequence* $\boldsymbol{X}$ *is*

$$\mathsf{cost}(\boldsymbol{X},\widetilde{\omega}) = \mathsf{cost}(\boldsymbol{X},\omega) + O\left(\sum_{i=1}^{m} \left|\log_B \omega_{i,x(i)} - \log_B \widetilde{\omega}_{i,x(i)}\right|\right).$$

In particular, if $B = O(n^{1/2-\delta})$ *for some* $\delta > 0$*, the guarantees hold with probability* $1-\delta$*.*

5 Experiments

In this section, we give experimental results that compare our learning-augmented Treap with prior work Lin et al. (2022) and classical search tree data structures including Red-Black Trees, AVL Trees, Splay Trees, B-Trees of order 3, and randomized Treaps. Experiments are conducted in a similar manner in Lin et al. (2022): **(1)** All keys are inserted in a random order to avoid insertion order sensitivity. **(2)** The total access cost is measured by the total number of comparisons needed while accessing keys.

We consider a synthetic data setting, with n unique items appearing in a sequence of length m. We assume the input data is drawn from a specified distribution. In our experiments, we explore three types of data distributions: the Zipfian distribution, the distribution described in Theorem 2.14, and the uniform distribution. We define the frequency of each item i as f_i and its relative frequency as $p_i = f_i/m$. In all the experiments, we take m to be one of [2000, 6000, 10000, 16000, 20000], and choose n to be 1000. The x-axis represents the number of unique items, and y-axis is the number of comparisons that we make, which indicates the access cost. All results are based on ten independent trials.

Zipfian Distribution. As a starting point, we consider Zipfian distribution, which is required by the data structure proposed in Lin et al. (2022). The Zipfian distribution with parameter α has relative frequencies $p_i = \frac{1}{i^\alpha H_{n,\alpha}}$, where $H_{n,\alpha} = \sum_{i=1}^{n} \frac{1}{i^\alpha}$ is the n^{th} generalized harmonic number of order α. In the experiments, we choose $\alpha = 1$.

Adversarial Distribution. In the proof of Theorem 2.14, we construct a distribution with relative frequency given by $p_i = \frac{2(n-i+1)}{n(n+1)}$. We prove that using the priority assignment as in Lin et al. (2022), the expected depth is $\Omega(n)$.

Uniform Distribution. We also consider uniform distribution, where the relative frequency of each item is $p_i = \frac{1}{n}$.

5.1 Perfect Prediction Oracle on Frequency

We first assume that we are given a perfect prediction oracle on the item frequency.

For the Zipfian distribution, as shown in Figure 2, our Treaps beat all other data structures except Lin et al. (2022), which requires Zipfian distribution assumption. Specifically, our data structure has 6-32% less cost than Splay Trees, 13-49% than AVL Trees and Red-Black Trees, and 43-103% than B-Trees of order 3 and randomized Treaps.

As shown in Figure 3 and Figure 4, for the adversarial distribution and the uniform distribution, our learning-augmented Treaps outperform all other data structures, including Lin et al. (2022). Specifically, our learning-augmented Treaps outperforms Lin et al. (2022) by 32-85% in both datasets. The experiments verify that our data structure works for arbitrary distribution while prior work (Lin et al., 2022) only works for Zipfian distribution.

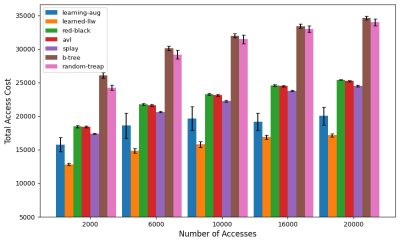

Figure 2: Zipfian distribution, $\alpha = 1$.

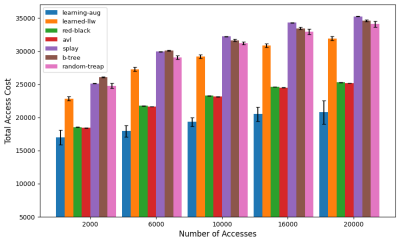

Figure 3: Adverserial Distribution.

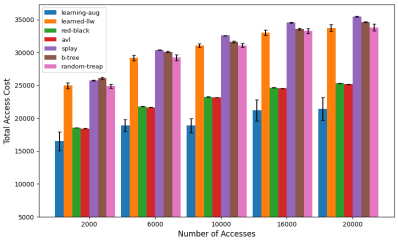

Figure 4: Uniform distribution.

5.2 Inaccurate Prediction Oracle on Frequency

We consider the scenario where our learning-augmented Treaps are constructed based on an inaccurate prediction of item frequencies. Specifically, we assume that the predicted frequency follows the adversarial distribution. However, the actual access sequence is generated by a mixture of two distributions: with probability w, the item follows the adversarial distribution and with probability $1 - w$, it follows the Zipfian distribution (Figure 5, Figure 6, Figure 7) or uniform distribution (Figure 8, Figure 9, Figure 10). In both settings, we compare our Treaps against Splay trees and randomized Treaps. Our experiments show that our Treaps outperform both alternatives, even when 75% of the data comes from an unknown distribution (either Zipfian or uniform). Furthermore, as the prediction quality decreases, the performance of our Treaps remains stable, demonstrating their robustness.

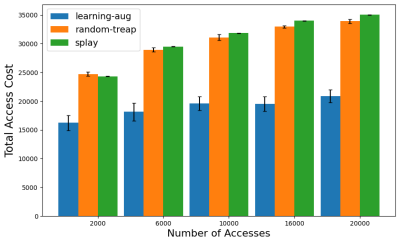

Figure 5: Mixture distribution, Zipfian has mixing weight 25%.

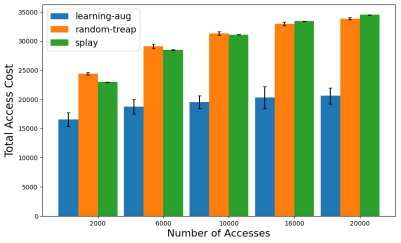

Figure 6: Mixture distribution, Zipfian has mixing weight 50%.

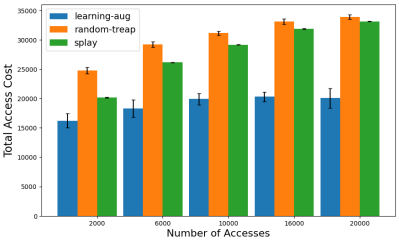

Figure 7: Mixture distribution, Zipfian has mixing weight 75%.

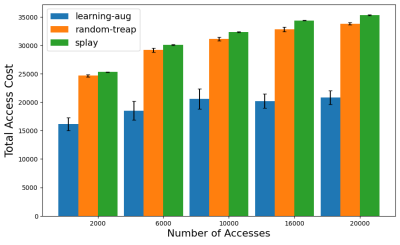

Figure 8: Mixture distribution, uniform has mixing weight 25%.

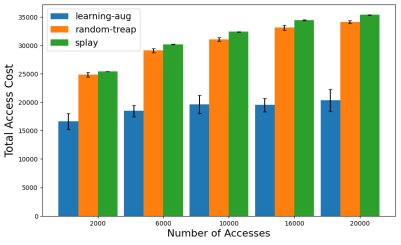

Figure 9: Mixture distribution, uniform has mixing weight 50%.

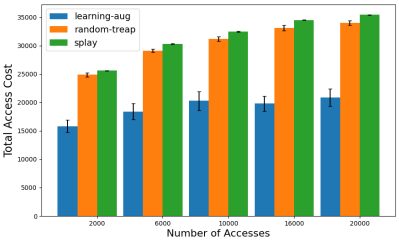

Figure 10: Mixture distribution, uniform has mixing weight 75%.

REFERENCES

M Adelson-Velskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, Joint Publications research service Washington DC, 1963.

Brian Allen and J. Ian Munro. Self-organizing binary search trees. *J. ACM*, 25(4):526–535, 1978.

Cecilia R Aragon and Raimund Seidel. Randomized search trees. In *FOCS*, volume 30, pp. 540–545, 1989.

Yossi Azar, Debmalya Panigrahi, and Noam Touitou. Online graph algorithms with predictions. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 35–66. SIAM, 2022.

Mihai Bădoiu, Richard Cole, Erik D Demaine, and John Iacono. A unified access bound on comparison-based dynamic dictionaries. *Theoretical Computer Science*, 382(2):86–96, 2007.

Michael A Bender, Roozbeh Ebrahimi, Haodong Hu, and Bradley C Kuszmaul. B-trees and cache-oblivious b-trees with different-sized atomic keys. *ACM Transactions on Database Systems (TODS)*, 41(3):1–33, 2016.

Prosenjit Bose, Karim Douieb, and Stefan Langerman. Dynamic optimality for skip lists and b-trees. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 1106–1114. Citeseer, 2008.

Prosenjit Bose, Jean Cardinal, John Iacono, Grigorios Koumoutsos, and Stefan Langerman. Competitive online search trees on trees. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1878–1891. SIAM, 2020.

Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pp. 546–554, 2003.

Trevor Brown. B-slack trees: Space efficient b-trees. In *Scandinavian Workshop on Algorithm Theory*, pp. 122–133. Springer, 2014.

Trevor Brown. B-slack trees: Highly space efficient b-trees. *arXiv preprint arXiv:1712.05020*, 2017.

Adam L Buchsbaum, Michael H Goldwasser, Suresh Venkatasubramanian, and Jeffery R Westbrook. On external memory graph traversal. In *SODA*, pp. 859–860, 2000.

Clément L Canonne. A short note on learning discrete distributions. *arXiv preprint arXiv:2002.11457*, 2020.

Parinya Chalermsook, Julia Chuzhoy, and Thatchaphol Saranurak. Pinning down the strong wilber 1 bound for binary search trees. *arXiv preprint arXiv:1912.02900*, 2019.

Justin Chen, Sandeep Silwal, Ali Vakilian, and Fred Zhang. Faster fundamental graph algorithms via learned predictions. In *International Conference on Machine Learning*, pp. 3583–3602. PMLR, 2022a.

Justin Y Chen and Piotr Indyk. Online bipartite matching with predicted degrees. *arXiv preprint arXiv:2110.11439*, 2021.

Justin Y Chen, Talya Eden, Piotr Indyk, Honghao Lin, Shyam Narayanan, Ronitt Rubinfeld, Sandeep Silwal, Tal Wagner, David P Woodruff, and Michael Zhang. Triangle and four cycle counting with predictions in graph streams. *arXiv preprint arXiv:2203.09572*, 2022b.

Richard Cole. On the dynamic finger conjecture for splay trees. part ii: The proof. *SIAM Journal on Computing*, 30 (1):44–85, 2000.

Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN 978-0-262-03384-8. URL http://mitpress.mit.edu/books/introduction-algorithms.

Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

Erik D Demaine, Dion Harmon, John Iacono, and Mihai Patraşcu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

Erik D Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Patraşcu. The geometry of binary search trees. In *Proceedings of the 20th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pp. 496–505. SIAM, 2009.

Jonathan C Derryberry and Daniel D Sleator. Skip-splay: Toward achieving the unified bound in the bst model. In *Workshop on Algorithms and Data Structures*, pp. 194–205. Springer, 2009.

Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 969–984, 2020.

Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. Faster matchings via learned duals. *Advances in Neural Information Processing Systems*, 34:10393–10406, 2021.

Jon Ergun, Zhili Feng, Sandeep Silwal, David P Woodruff, and Samson Zhou. Learning-augmented k-means clustering. *arXiv preprint arXiv:2110.14094*, 2021.

Thomas Erlebach, Murilo Santos de Lima, Nicole Megow, and Jens Schlöter. Learning-augmented query policies for minimum spanning tree with uncertainty. *arXiv preprint arXiv:2206.15201*, 2022.

Rolf Fagerberg, David Hammer, and Ulrich Meyer. On optimal balance in b-trees: What does it cost to stay in perfect shape? In *30th International Symposium on Algorithms and Computation (ISAAC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, 13(8):1162–1175, 2020.

Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. Why are learned indexes so effective? In *International Conference on Machine Learning*, pp. 3123–3132. PMLR, 2020.

Daniel Golovin. *Uniquely represented data structures with applications to privacy*. PhD thesis, Carnegie Mellon University, 2008.

Daniel Golovin. B-treaps: A uniquely represented alternative to b-trees. In *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*, pp. 487–499. Springer, 2009.

Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pp. 8–21. IEEE, 1978.

Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. Learning-based frequency estimation algorithms. In *International Conference on Learning Representations*, 2019.

TC Hu and AC Tucker. Optimum binary search trees. Technical report, WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER, 1970.

John Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pp. 516–522, 2001.

John Iacono. Key-independent optimality. *Algorithmica*, 42(1):3–10, 2005.

John Iacono. In pursuit of the dynamic optimality conjecture. In *Space-Efficient Data Structures, Streams, and Algorithms*, pp. 236–250. Springer, 2013.

HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, pp. 16–25, 1997.

Chris Jermaine, Anindya Datta, and Edward Omiecinski. A novel index supporting high volume data warehouse insertion. In *VLDB*, volume 99, pp. 235–246, 1999.

Tanqiu Jiang, Yi Li, Honghao Lin, Yisong Ruan, and David P Woodruff. Learning-augmented data stream algorithms. *ICLR*, 2020.

Marek Karpinski, Lawrence L Larmore, and Wojciech Rytter. Sequential and parallel subquadratic work algorithms for constructing approximately optimal binary search trees. In *SODA*, pp. 36–41. Citeseer, 1996.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 international conference on management of data*, pp. 489–504, 2018.

Thomas Lavastida, Benjamin Moseley, R Ravi, and Chenyang Xu. Learnable and instance-robust predictions for online matching, flows and load balancing. *arXiv preprint arXiv:2011.11743*, 2020.

Honghao Lin, Tian Luo, and David Woodruff. Learning augmented binary search trees. In *International Conference on Machine Learning*, pp. 13431–13440. PMLR, 2022.

Joan Marie Lucas. *Canonical forms for competitive binary search tree algorithms*. Rutgers University, Department of Computer Science, Laboratory for Computer . . . , 1988.

Giorgos Margaritis and Stergios V Anastasiadis. Efficient range-based storage management for scalable datastores. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):2851–2866, 2013.

Kurt Mehlhorn. Best possible bounds for the weighted path length of optimum binary search trees. In H. Barkhage (ed.), *Automata Theory and Formal Languages, 2nd GI Conference, Kaiserslautern, May 20-23, 1975*, volume 33 of *Lecture Notes in Computer Science*, pp. 31–41. Springer, 1975a.

Kurt Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5(4):287–295, 1975b.

Michael Mitzenmacher and Sergei Vassilvitskii. Algorithms with predictions. *arXiv preprint arXiv:2006.09123*, 2020.

J Ian Munro. On the competitiveness of linear search. In *European symposium on algorithms*, pp. 338–345. Springer, 2000.

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

Adam Polak and Maksym Zub. Learning-augmented maximum flow. *arXiv preprint arXiv:2207.12911*, 2022.

Arnold L Rosenberg and Lawrence Snyder. Time-and space-optimality in b-trees. *ACM Transactions on Database Systems (TODS)*, 6(1):174–193, 1981.

Roodabeh Safavi and Martin P Seybold. B-treaps revised: Write efficient randomized block search trees with high load. *arXiv preprint arXiv:2303.04722*, 2023.

Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32 (3):652–686, 1985.

Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proceedings of the VLDB Endowment*, 14(8):1276–1288, 2021.

F Frances Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4):532–540, 1982.

Ke Yi. Dynamic indexability and the optimality of b-trees. *Journal of the ACM (JACM)*, 59(4):1–19, 2012.