On the Power of Learning-Augmented BSTs

Jingbang Chen
Georgia Tech
chenjb@gatech.edu

Li Chen
Georgia Tech
lichen@gatech.edu

November 18, 2022

Abstract

We present the first Learning-Augmented Binary Search Tree (BST) that attains Static Optimality and Working-Set Bound given rough predictions. Following the recent studies in algorithms with predictions and learned index structures, Lin, Luo, and Woodruff (ICML 2022) introduced the concept of Learning-Augmented BSTs, which aim to improve BSTs with learned advice. Unfortunately, their construction gives only static optimality under strong assumptions on the input.

In this paper, we present a simple BST maintenance scheme that benefits from learned advice. With proper predictions, the scheme achieves Static Optimality and Working-Set Bound, respectively, which are important performance measures for BSTs. Moreover, the scheme is robust to prediction errors and makes no assumption on the input.

1 Introduction

In recent years, there is a great amount of research on incorporating machine learning models into algorithm designs. A new field named Algorithms with Predictions [MV20] has attracted people’s attention. In particular, machine learning models are used to predict input patterns for improving performance. For example, [APT22] gives online graph algorithms with predictions; [CM05] improves the hashing-based method Count-Min; [EFSWZ21] introduces a k-means clustering that is learning-augmented, etc. Oracles that predict desired properties are crucial in these works and do exist in practice. For example, there are some empirical results on predicting item frequencies in a data stream [HIKV19; JLLRW20].

Assuming an oracle predicts properties of the upcoming accesses, one can leverage the information and build more efficient data structures, named the Learning-Augmented Data Structures. The Index structure is an important instance of Learning-Augmented Data Structures, which is very useful in the database management system. The learned index structures were studied first in [KBCDP18], which uses deep-learning models to predict the position or existence of records instead of the traditional B-Tree or Hash-Index. [KBCDP18] only deals with the static case. Later, [FV20b; DMYWDLZCGK+20; WZCWX21] propose dynamic learned index structures with provably efficient time and space upper bounds of updates in the worst case. These structures also outperform the traditional B-Tree empirically.

While scientists believe that data structures can be replaced by a learned model and an auxiliary structure, some traditional data structures have the potential to adapt the data pattern and give better performance. The binary search tree (BST), a fundamental index structure, demonstrates such a possibility. Introduced in 1985, the Splay tree [ST85] is able to adapt data patterns naturally and give performance bounded by the entropy of the data distribution. However, other balanced
BSTs are unlikely to leverage the input patterns to improve performance further due to their strict adjusting mechanism. One might ask: Is it possible to adapt the adjusting mechanism of some balanced BSTs to be learning-augmented? Some recent works show it is possible to do so.

Treap [AS89] is another kind of balanced binary search tree that uses randomization to keep the height low. It assigns a random value to every node as the priority, denoted as $\text{pri}_x$, in addition to the key. In addition to being a binary search tree, Treaps satisfy the Heap property, i.e., any node has a smaller priority than its parent. In general, Treap simply uses randomness to ensure a low height instead of balancing the tree preemptively. One could guess that Treaps can be made learning-augmented with a twist in the priority assignments. The work in [LLW22] confirms such adaptation is possible. However, their result is limited as they assume strong properties on the input. Following the line of work, some questions are now on the table:

- Whether a more general learning-augmented Treap exists?
- Do they have similar performance guarantees like those conjectured of dynamic optimality?

In this paper, we answer both questions affirmatively. We develop a new variant of learning-augmented Treap that is capable of adapting access patterns without strong assumptions on the input.

Our starting point is a Score-Respecting Treap. In general, a learned oracle would output an importance score $s(x)$ for each item $x$. The smaller the score, the more important the item. The score may change over time. Given these predicted importance scores $s(x)$, a learning-augmented BST should respect the score, i.e., the smaller the score, the easier to access the item.

One of our main technical results is the following theorem that constructs a Treap given importance scores $s(x)$. We will prove the theorem in Section 3.

**Theorem 1.1.** [Score-Respecting Treap] We say a score function $s(\cdot)$ is smooth if for any $c > 0$, there are only $O(\text{poly}(c))$ items with score at most $c$. Given a smooth score $s : [n] \rightarrow \mathbb{R}_{\geq 1}$, consider the following priority assignment for Treap nodes:

$$\text{pri}_x := -\lfloor \log_2 \log_2 s(x) \rfloor + \delta_x, \quad \delta_x \sim U(0, 1), \quad x \in [n].$$

The expected depth of any item $x \in [n]$ is $O(\log s(x))$.

**Remark 1.2.** Score-respecting Treaps support insertions and deletions naturally. Suppose the score of some node $x$ changes from $s(x)$ to $s'(x)$, we simply delete the node and insert it back using typical Treap maintenance. The expected update time is proportional to the sum of $x$’s depth before and after the update, which is $O(\log s(x) + \log s'(x))$.

Combining the theorem with learned oracles that predict proper scores, we construct learning-augmented Treaps that admit Static Optimality and Working Set Theorem which are fundamental performance measures for binary search trees.

**Static Optimality** Suppose each item $x$ is accessed $f_x$ times among a total of $m$ accesses. The minimum cost for a static binary search tree to process all $m$ accesses is $\Theta(\sum_x f_x \log(m/f_x))$. In particular, each item $x$ has depth $\Theta(\log(m/f_x))$ in the optimal static BST.

Given a learned oracle that predicts item frequencies $f_x$ of each item $x$. We can construct a statically optimal BST using score-respecting Treaps. This construction makes no assumptions about the input, unlike [LLW22], which requires random item order and Zipfian distribution. We will prove the theorem in Section 4 as a direct application of Theorem 1.1.
Informal Theorem 1.3. [Static Optimality] Consider the following priority assignment for Treap nodes:

$$\text{pri}_x := -[\log_2 \log_2 (m/f_x)] + \delta_x, \delta_x \sim U(0,1), \ x \in [n],$$

where $m$ is the total number of access, and $f_x$ is the number of times item $x$ is accessed.

For any access sequence, the expected depth of each item $i$ is $O(\log(m/f_x))$ in the treap using the above priorities.

We show that the performance is robust under errors in the frequency predictions (Section 4.1). In addition, we use this example to demonstrate the need for $\log_2 \log_2 s(x)$ in the node priority (Section 4.2). In particular, we show that setting priorities using $f_x$ or $\log_2 (m/f_x)$ would have large expected depth for some input distribution. The case of $f_x$ is exactly the one used in [LIW22], which bypasses our counter-example because of the strong input assumptions.

Working-Set Theorem Let $X = x(1), x(2), \ldots, x(m)$ be the access sequence. Conceptually, if the user only accesses a small set of items recently, a good binary search tree should keep these items close to the root for faster access time. This observation motivates the Working-Set Theorem, a BST performance measure first proved true for Splay trees.

Define the Working Set Size at time $i$, denoted by $\text{work}(i)$, to be the number of distinct items accessed since the previous access of the same item $x(i)$. Intuitively, a good BST should have faster access time when $\text{work}(i)$ is small. Formally, a BST satisfies the Working-Set Theorem if the cost to process $X$ is $O(n \log n + \sum_i \log(\text{work}(i) + 1))$.

We consider two types of learned advice that help us to achieve the Working-Set Theorem. First, we consider the case where the learned oracle predicts working set sizes in the future. Define the Future Working Set Size at time $i$, denoted by $\text{future}(i)$, is the number of distinct items accessed until the next access of the same item $x(i)$. That is, if $x(i)$ is accessed next at time $j$, the future working set size $\text{future}(i)$ is $\text{work}(j)$, the working set size at time $j$. Given an oracle that predicts the value of $\text{future}(i)$ crudely, we can combine it with a score-respecting Treap to achieve the Working-Set Theorem.

The following theorem will be proved in Section 5.

Informal Theorem 1.4. [Working-Set Theorem with Future Information] Given an oracle that predicts $s(i) \approx \text{future}(i)$ at any time $i$. Consider the following priority assignment scheme: At time $i$, after accessing item $x(i)$, we update the priority of $x(i)$ to be:

$$\text{pri}_{x(i)} := -[\log_2 \log_2 (s(i) + 1)] + \delta, \ \delta \sim U(0,1).$$

When accessing $x(i)$, the expected depth of item $x(i)$ is $O(\log(\text{work}(i) + 1))$. The total cost is $O(n \log n + \sum_i \log(\text{work}(i) + 1))$ in expectation.

Next, we show that the Working-Set Theorem can be achieved if the oracle only maintains estimations of the Past Working Set Size. Formally, we define the working set size of $x$ before time $i$, denoted by $\text{work}(i, x)$, as the number of distinct items accessed since the previous access of the item $x$. Combining the score-respecting Treaps with oracles that maintain crude estimations of $\text{work}(i, x)$, we also achieve the Working-Set Theorem.

The following theorem will be proved in Section 5.

Informal Theorem 1.5. [Working-Set Theorem with Past Working Set Sizes] Given an oracle that maintains $s(i, x)$, a crude approximation to $\text{work}(i, x)$, at any time $i$ for any item $x$. Consider
the following priority assignment scheme: At time \( i \), after accessing item \( x(i) \), we maintain the priority of each item \( x \) to be

\[
\text{pri}_x := -\lfloor \log_2 \log_2 (s(i, x) + 1) \rfloor + \delta_x, \quad \delta_x \sim U(0, 1).
\]

When accessing \( x(i) \), the expected depth of item \( x(i) \) is \( O(\log(\text{work}(i)+1)) \). The total cost, excluding the one for updating priorities, is \( O(n \log n + \sum_i \log(\text{work}(i)+1)) \) in expectation.

Outline We provide an overview in Section 1.1 on Learning-augmented algorithms, balanced BSTs and the Dynamic Optimality Conjecture. In Section 2, we give necessary notations as well as basic properties of T reaps. In Section 3, we formally introduce our learning-augmented T reap as the Score-Respecting T reap framework. In Section 4 and 5, we analyze how to achieve the Static Optimality and Working-Set properties with our T reap.

1.1 Related Works

Learning-augmented Algorithms Learning-augmented algorithms take advantage of prediction oracles to improve performance, such as the running time or the output quality. In addition to the problem instance, learning-augmented algorithms have access to an oracle that outputs learned predictions. Some classic online problems have attracted great interest from such scheme, including the ski rental problem [MV20; PSK18; GP19; WLW20; Ban20], the job scheduling problem [MV20; PSK18; Mit19], the online knapsack problem [IKMP21; BFL22; ZSHW21], the generalized sorting problem [LRSZZ21], and the online Steiner tree problem [XM22]. Meanwhile, some learning-augmented algorithms in graph theory also get developed: shortest path [CSVZZ22], network flow [PZ22; LMRX20], matching [CSVZZ22; DLIMV21; CI21], spanning tree [ELMS22], and triangles/cycles counting [CEILNRSWWZ22]. Learning-Augmented Data structure is also an essential field [FV20a]: Besides [LLW22] on BSTs, there are works in learned index structures [FV20b; KBCDP18], and hashing-based methods [CM05; CCF02]. Towards the heated trend in developing learning-augmented algorithms, [FLV21] discusses the performance of learned data structures and [ELMS20] studies how to utilize predictions in learning-augmented algorithms. Learning-augmented algorithms’ performance is highly dependent on the quality of predictions. Predictive modeling is extensively studied in machine learning [Fin14; Gki17]. In developing learning-augmented online algorithms, online frequency prediction is often used. Works including [HIKV19; KBT22; JLLRW20] provide such estimators with analysis on the errors.

Balanced Binary Search Tree There have been many studies on analyzing binary search trees under different scenarios. J.Culberson, J.I.Munro [CM89; CM90] studied the BST behavior after a sufficient number of updates and concluded that if these updates are produced randomly, the average search cost will degrade to \( \Theta(n^{1/2}) \). One could improve the performance of binary search trees if we know the update/query sequence. One can construct the optimal static BST in \( O(n^2) \) time on \( n \) items [HT70] and a nearly optimal BST in \( O(n \log n) \) [Meh75]. However, these do not support updates such as insertions or deletions and require knowledge of the input. The first balanced binary search tree was invented in 1962 by Georgy Adelson-Velsky and Evgenii Landis [AL63], called the AVL tree. The AVL tree controls heights by restricting the balance factor for each node. Whenever a node becomes unbalanced after an update, the AVL tree will restore the balance using tree rotations. The mechanism gives a worst-case \( O(\log n) \)-update/query time. Following Rudolf Bayer’s 1972 work [Bay72], the Red-black tree is officially proposed in 1978 by L.J.Guibas and R.Sedgewick [GS78]. The Red-Black tree has worst-case \( O(\log n) \)-time
and amortized constant time per update/query via rotations and coloring mechanism. In 1985, the renowned Splay tree \cite{ST85} was invented by Daniel Sleator and Robert Tarjan. Its balance mechanism is simple: whenever a node is accessed, rotate it to the root. Splay trees attain an amortized $O(\log n)$-time per query/update/access and various performance theorems. In addition, there are many other balanced binary search trees. Weight-balanced trees \cite{NR72} maintain the balance according to the given weights on nodes. Tango trees \cite{DHIP07} are the first $O(\log \log n)$-competitive dynamic BSTs and they work by partitioning the tree into a set of preferred paths.

**Dynamic Optimality Conjecture** In the past few decades, many works focus on studying the Dynamic Optimality Conjecture \cite{ST85} i.e., performance no worse than any other dynamic BSTs up to a constant factor, which remains unsolved. We refer most contents in the following to \cite{Iac13}, a survey on this topic by John Iacono. For any sufficiently long access sequence $X$, we define $\text{OPT}(X)$ as the minimum cost to process $X$ by any binary search tree. The very first known result is by Knuth \cite{HT70}, giving an entropy bound. This bound is expected to be $O(\text{OPT}(X))$ if each access is randomly sampled from a fixed distribution on items. The conjecture in \cite{ST85} says that the Splay Tree’s execution time is $O(\text{OPT}(X))$. People also attempt to provide non-trivial lower bounds of $\text{OPT}(X)$. There are several known bounds: Independent Rectangle Bound \cite{DHIP09}, Alternation Bound \cite{Wil89}, and the Funnel Bound. The latter two bounds are asymptotically implied by the Independent Rectangle Bound. There are more works on giving an upper bound. In the original work \cite{ST85}, the Working-Set upper bounds are proved, which says an access is fast if it was accessed recently. In 2000, Richard Cole proves the Dynamic Finger theorem in \cite{CMSS00, Col00}. The Dynamic Finger property says an access is fast if it is close in the key value compared to the previous access. John Iacono combines two bounds into a unified access bound \cite{BCDI07}. He also defines a key-independent optimality property for binary search trees \cite{Iac05}, which is asymptotically equivalent to the Working Set theorem. One can interpret the key-independent optimality that if key values are assigned arbitrarily, the Splay Tree is optimal. However, all these bounds are not tight. No BST is known to be better than $O(\log n)$-competitive for a long time until the Tango tree \cite{DHIP07}, which is the first $O(\log \log n)$-competitive dynamic BSTs. In addition to the Tango tree, there are some data structures giving a better upper bound, including the unified structure \cite{Iac01, BD04, DS09}. If the rotations of tree nodes are free, \cite{BC03} gives an online BST data structure whose access cost is $O(\text{OPT}(X))$. A recent work \cite{BCIKL20} inspired by the Tango tree is also an online $O(\log \log n)$-competitive search tree data structure.

## 2 Preliminaries

In this section, we introduce notations in use and present some basic properties of Treaps. We assume that every BST node has its item from the ordered universe $[n]$. We use $x$ to identify the unique tree node with item $x$ when the BST is clear from the context. We say $a \preceq b$ if there exists a universal constant $L > 0$ such that $a \leq Lb$ always holds.

In this paper, we use $x, y, z$ to denote items from $[n]$. Given any two items $x, y \in [n]$, we use $[x, y]$ to denote the subset $\{x, x + 1, \ldots, y\}$ or $\{y, y + 1, \ldots, x\}$ if $y \leq x$. We use $X = (x(1), x(2), \ldots, x(m)) \in [n]^m$ to denote the access sequence and $m$ to denote the length. We use $i, j,$ and $k$ to index the access sequence and to denote time. In particular, we put superscript $u^{(t)}$ to denote the state of the variable $u$ after processing the access $x(t)$.

**Definition 2.1** (Treap, \cite{AS89}). Let $T$ be a Binary Search Tree over $[n]$ and $\text{pri} \in \mathbb{R}^n$ be a priority assignment on $[n]$. We say $(T, \text{pri})$ is a Treap if $\text{pri}_x \preceq \text{pri}_y$ whenever $x$ is a descendant of $y$ in $T$. 


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

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

From now on, we always assume that $\mathbf{p}$ has distinct values. Therefore, when $\mathbf{p}$ is clear from the context, the term Treap is referred to the unique BST $T$.

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

A classical result of [AS89] states that if priorities are randomly assigned, the depth of the Treap cannot be too large.

**Lemma 2.2 ([AS89]).** Let $U(0,1)$ be the uniform distribution over the real interval $[0,1]$. If $\mathbf{p} \sim U(0,1)^n$, each treap node $x$ has expected depth $\Theta(\log n)$.

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

$$
\mathbb{E}[\text{depth}(x)] = \sum_{y \in [n]} \mathbb{E}[1 \text{ if } y \text{ is an ancestor of } x \text{ or } 0] = \sum_{y \in [n]} \Pr(y \text{ is an ancestor of } x) = \sum_{y \in [n]} \Pr \left( \mathbf{p}_y = \max_{z \in [x,y]} \mathbf{p}_z \right) = \sum_{y \in [n]} \frac{1}{|x - y + 1|} = \Theta(\log n).
$$

Treaps can be made dynamic and support operations such as insertions and deletions.

**Lemma 2.3 ([AS89]).** Given a Treap $T$ and some item $x \in [n]$, $x$ can be inserted to or deleted from $T$ in $O(\text{depth}(x))$-time.

### 3 Score-Respecting Treaps

In this section, we prove Theorem 1.1. First, let us formally define the class of smooth scores.

**Definition 3.1 (Smooth Score).** Consider a score function $s$ that maps each item $x \in [n]$ to a positive number $s(x)$. Define another function $s_{\leq}(k)$ for any positive number $k$ to be the number of items whose score is at most $k$. We say $s$ is smooth if $s_{\leq}(k) = O(\text{poly}(k))$.

Recall the statement of Theorem 1.1.
**Theorem 1.1.** [Score-Respecting Treap] We say a score function $s(\cdot)$ is smooth if for any $c > 0$, there are only $O(\text{poly}(c))$ items with score at most $c$. Given a smooth score $s : [n] \to \mathbb{R}_{\geq 1}$, consider the following priority assignment for Treap nodes:

$$\text{pri}_x := -\lfloor \log_2 \log_2 s(x) \rfloor + \delta_x, \quad \delta_x \sim U(0, 1), \quad x \in [n].$$

The expected depth of any item $x \in [n]$ is $O(\log s(x))$.

For any item $x \in [n]$, we define $x$’s weight, $w_x$, to be the integral part of its priority, i.e.,

$$w_x := \lfloor \log_2 \log_2 s(x) \rfloor = -\lfloor \text{pri}_x \rfloor.$$  \hfill (1)

In addition, we define $S_w = \{x \in [n] \mid w_x = w\}$ for any integer $w$. For simplicity, we assume that $w_x \geq 0$ for any $x$ WLOG. This is because $w_x < 0$ implies $s(x) = O(1)$ which can hold for only a constant number of items. We can always make them stay at the top of the Treap. This only increases the depths of others by a constant.

Let $x$ start at any node and approach the root. The weight $w_x$ decreases. In other words, the smaller the $w_x$, the smaller the depth $\text{depth}(x)$. However, there may be many items with identical weights $w_x$. The ties are broken randomly. Similar to the ordinary Treaps, where $w_x$’s are all zero, any item has $O(\log |S_w|)$ ancestors with weight $w$. To prove the desired bound, we will show that $\log |S_w| = O(2^w)$. Since each ancestor of item $x$ has weight at most $w_x$, the expected depth $\mathbb{E}[\text{depth}(x)]$ can be bound by $O(2^0 + 2^1 + \ldots + 2^{w_x}) = O(2^{w_x}) = O(\log(s(x)))$. First, let us bound the size of each $S_w$.

**Lemma 3.2.** For any non-negative integer $w \geq 0$, $|S_w| = O(2^{O(2^w)})$.

**Proof.** Observe that $x \in S_w$ if and only if

$$w \leq \log_2 \log_2 s(x) \leq w + 1, \quad \text{and} \quad 2^{2^w} \leq s(x) \leq 2^{2^{w+1}}.$$  

However, there are only $O(\text{poly}(2^{2^w+1})) = O(2^{O(2^w)})$ items with score at most $2^{2^w+1}$ because the score is smooth (Definition 3.1). \hfill $\square$

Next, we bound the expected number of ancestors of item $x$ in $S_w, w \leq w_x$.

**Lemma 3.3.** Let $x \in [n]$ be any item and $w \leq w_x$ be a non-negative integer. The expected number of ancestors of $x$ in $S_w$ is at most $O(\log |S_w|)$.

**Proof.** First, we show that any $y \in S_w$ is an ancestor of $x$ with probability no more than $1/|S_w \cap [x, y]|$. 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 $S_w \cap [x, y]$. However, priorities of items in $S_w \cap [x, y]$ are i.i.d. random variables of the form $-w + U(0, 1)$. Thus, the probability that $\text{pri}_y$ is the largest among them is $1/|S_w \cap [x, y]|$.

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

$$\mathbb{E}[\text{number of ancestors of } x \text{ in } S_w] = \sum_{y \in S_w} \Pr(y \text{ is an ancestor of } x)$$

$$\leq \sum_{y \in S_w} \frac{1}{|S_w \cap [x, y]|}$$

$$\leq 2 \cdot \sum_{u=1}^{|S_w|} \frac{1}{u} = O(\log |S_w|),$$

where the second inequality comes from the fact that for a fixed value of $u$, there are at most two items $y \in S_w$ with $|S_w \cap [x, y]| = u$ (one with $y \leq x$, the other with $y > x$). \hfill $\square$
Now we are ready to prove Theorem 1.1.

Proof of Theorem 1.1. Let \( x \in [n] \) be any item. Linearity of expectation yields

\[
E[\text{depth}(x)] = \sum_{w=0}^{w_x} E[\text{number of ancestors of } x \text{ in } S_w] \\
\lesssim \sum_{w=0}^{w_x} \log |S_w| \\
\lesssim \sum_{w=0}^{w_x} 2^w \lesssim 2^{w_x}.
\]

We conclude the proof by observing that

\[
w_x \leq \log \log 2 \log 2 \left( \frac{m}{f_x} \right) \leq w_x + 1, \text{ and } 2^{w_x} \leq \log s(x).
\]

4 Static Optimality given Item Frequencies

In this section, we prove Theorem 1.3.

Informal Theorem 1.3. [Static Optimality] Consider the following priority assignment for Treap nodes:

\[
\text{pri}_x := -\left\lfloor \log \log 2 \left( \frac{m}{f_x} \right) \right\rfloor + \delta_x, \text{ } \delta_x \sim U(0, 1), \text{ } x \in [n],
\]

where \( m \) is the total number of access, and \( f_x \) is the number of times item \( x \) is accessed.

For any access sequence, the expected depth of each item \( i \) is \( O(\log(\frac{m}{f_x})) \) in the treap using the above priorities.

We present a priority assignment for constructing statically optimal Treaps given item frequencies. Given any access sequence \( X = (x(1), \ldots, x(m)) \), we define \( f_x \) for any item \( x \), to be its frequency in \( X \), 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 technical result which is a simple application of Theorem 1.1:

Theorem 4.1 (Formal Version of Theorem 1.3). For any item \( x \in [n] \), we set its priority as

\[
\text{pri}_x := -\left\lfloor \log \log 2 \left( \frac{m}{f_x} \right) \right\rfloor + \delta_x, \text{ } \delta_x \sim U(0, 1).
\]

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

Proof. Given item frequencies \( f \), we define the following score function \( s \) which will be proved to be smooth:

\[
s(x) := \frac{m}{f_x}, \text{ } x \in [n].
\]

For any \( k > 0 \), there are at most \( k \) items with score at most \( k \) because any such item appears at least \( m/k \) times. Thus, \( s \) is a smooth score (Definition 3.1). Theorem 1.1 yields that the expected depth of each item \( x \) is \( O(\log(\frac{m}{f_x})) \).
4.1 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 drawback in performance given 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 \( p_x \) as a probability distribution over \( [n] \). Using the notion of entropy, one can express Theorem 4.1 as the following claim:

**Definition 4.2 (Entropy).** Given a probability distribution \( p \) over \( [n] \), define its Entropy as

\[
\text{Ent}(p) := \sum_x p_x \log(1/p_x) = \mathbb{E}_{x \sim p}[\log(1/p_x)].
\]

**Corollary 4.3.** In Theorem 4.1, the expected depth of each item \( x \) is \( O(\log(1/p_x)) \) and the expected total cost is \( O(m \cdot \text{Ent}(p)) \), where \( \text{Ent}(p) = \sum_x p_x \log(1/p_x) \) measures the entropy of the distribution \( p \).

The appearance of entropy in the runtime bound suggests that some more related notations would appear in the analysis. Let us present several related notions.

**Definition 4.4 (Cross Entropy).** Given two distributions \( p, q \) over \( [n] \), define its Cross Entropy as

\[
\text{Ent}(p, q) := \sum_x p_x \log(1/q_x) = \mathbb{E}_{x \sim p}[\log(1/q_x)].
\]

**Definition 4.5 (KL Divergence).** Given two distributions \( p, q \) over \( [n] \), define its KL Divergence as

\[
D_{\text{KL}}(p, q) = \text{Ent}(p, q) - \text{Ent}(p) = \sum_x p_x \log(p_x/q_x).
\]

First, we analyze the run time given frequency estimations \( q \).

**Theorem 4.6.** Given an estimation \( q \) on the relative frequencies \( p \). For any item \( x \in [n] \), we draw a random number \( \delta_x \sim U(0, 1) \) and set its priority as

\[
\text{pri}_x := -\left\lfloor \log_2 \log_2 \frac{1}{q_x} \right\rfloor + \delta_x.
\]

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

**Proof.** Define score \( s(x) = 1/q_x \) for each item \( x \in [n] \). Clearly, \( s \) is smooth and we can apply Theorem 1.1 to prove the bound on the expected depths. The total time for processing the access sequence is, by definition,

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

Using the theorem, one can relate the drawback in the performance given \( q \) in terms of KL Divergence.

**Corollary 4.7.** In the setting of Theorem 4.6, the extra time spent compared to using \( p \) to build the Treap is \( O(m \cdot D_{\text{KL}}(p, q)) \) in expectation.
4.2 Analysis of Other Variations

In this section, we discuss two different priority assignments. For each assignment, we construct an input distribution that creates a larger expected depth than Theorem 1.3. We define the distribution \( p \) as \( p_x = f_x/m, x \in [n] \).

The first priority assignment is used in \[LLW22\]. They assign priorities according to \( p_x \) entirely, i.e., \( \text{pri}_x = p_x, x \in [n] \). Assuming that items are ordered randomly, and \( p \) is a Zipfian distribution, \[LLW22\] shows Static Optimality. However, it does not generally hold, and the expected access cost could be \( \Omega(n) \).

**Theorem 4.8.** Consider the priority assignment that assigns the priority of each item to be \( \text{pri}_x := p_x, x \in [n] \). There is a distribution \( p \) over \( [n] \) such that the expected access time, \( \mathbb{E}_{x \sim p}[\text{depth}(x)] = \Omega(n) \).

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

\[
\mathbb{E}_{x \sim p}[\text{depth}(x)] = \sum_{x \in [n]} p_x \cdot \text{depth}(x)
= \sum_{x \in [n]} \frac{2(n-x+1)}{n(n+1)} \cdot x
= \frac{2}{n(n+1)} \sum_{x \in [n]} x(n-x+1)
\geq \frac{2}{n(n+1)} \cdot n^3 \geq n.
\]

Next, we consider a very similar assignment to ours.

**Theorem 4.9.** Consider the following priority assignment that sets the priority of each node \( x \) as \( \text{pri}_x := -[\log 1/p_x] + \delta_x, \delta_x \sim U(0,1) \). There is a distribution \( p \) over \( [n] \) such that the expected access time, \( \mathbb{E}_{x \sim p}[\text{depth}(x)] = \Omega(\log^2 n) \).

**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 \( p \). 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 \( p \) is indeed a probability distribution over \([n]\).

In the Treap with the given priority assignment, \( S_i \) forms a subtree of expected height \( \Omega(\log n) \) 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 \( \Omega((K-i) \log n) \). One can lower bound the expected access
time (which is the expected depth) as:

\[ E_{x \sim p}[^{\text{depth}}(x)] \geq \sum_{i=1}^{K} \sum_{x \in S_i} p_x \cdot (K - i) \cdot \log n \]

\[ = \sum_{i=1}^{K} p(S_i) \cdot (K - i) \cdot \log n \]

\[ = \sum_{i=1}^{K} \frac{1}{K} \cdot (K - i) \cdot \log n \gtrsim K \log n \gtrsim \log^2 n, \]

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

\[ \square \]

5 Working-Set Theorem given Working Set Sizes

In this section, we prove the Working-Set Theorem previously described in Theorem 1.4 and 1.5. First, let us introduce some formal notations common in both settings.

**Definition 5.1 (Previous and Next Access prev(i, x) and next(i, x)).** Given an access sequence \( X = (x(1), \ldots, x(m)) \), we can define \( \text{prev}(i, x) \) to be the previous access of item \( x \) before time \( i \), i.e., \( \text{prev}(i, x) := \max \{ j < i \mid x(j) = x \} \). We also define \( \text{next}(i, x) \) to be the next access of item \( x \) after time \( i \), i.e., \( \text{next}(i, x) := \min \{ j > i \mid x(j) = x \} \).

**Definition 5.2 (Working Set Size work(i, x)).** Given any access sequence \( X = (x(1), \ldots, x(m)) \), we define the Working Set Size \( \text{work}(i, x) \) to be the number of distinct items accessed between time \( i \) and the previous access of the item \( x \). That is,

\[ \text{work}(i, x) := | \{ x(j + 1), \ldots, x(i - 1) \} |, \text{ where } j = \text{prev}(i, x). \]

If \( x \) does not appear before time \( i \), we define \( \text{work}(i, x) := n \). We use \( \text{work}(i) \) to denote \( \text{work}(i, x(i)) \).

As previously mentioned, both settings use score-respecting Treaps (Theorem 1.1) as important building blocks. To use the data structure, one needs to specify the score of each item and certify that the score is smooth. In the case of proving Working-Set Theorems, we use some approximations of the working set sizes \( \text{work}(i, x) \) as scores. We justify the use by observing that \( \text{work}(i, x) \) is a smooth score when \( i \) is fixed.

**Observation 3.** Fix a timestamp \( i \), \( \text{work}(i, x) + 1 \) is a smooth score on \([n]\).

**Proof.** Recall for each item \( x \), \( \text{prev}(i, x) \) is the previous access of item \( x \) before time \( i \). \( \text{work}(i, x) \) is exactly the number of items \( y \) such that \( \text{prev}(i, x) < \text{prev}(i, y) \). Let \( \pi_1, \pi_2, \ldots, \pi_n \) be a permutation of \([n]\) in the order of decreasing \( \text{prev}(i, x) \). For any item \( x \), \( \text{work}(i, x) + 1 \) is simply the index of \( x \) in \( \pi \).

\[ \square \]

5.1 Given Predicted Future Working Set Sizes

Given an oracle predicting future working-set sizes, we can maintain a score-respecting Treap (Theorem 1.1) that achieves the Working-Set Theorem.
Informal Theorem 1.4. [Working-Set Theorem with Future Information] Given an oracle that predicts $s(i) \approx \text{future}(i)$ at any time $i$. Consider the following priority assignment scheme: At time $i$, after accessing item $x(i)$, we update the priority of $x(i)$ to be:

$$\text{pri}_{x(i)} := -\lfloor \log_2 \log_2 (s(i) + 1) \rfloor + \delta, \ \delta \sim U(0, 1).$$

When accessing $x(i)$, the expected depth of item $x(i)$ is $O(\log(\text{work}(i) + 1))$. The total cost is $O(n \log n + \sum_i \log(\text{work}(i) + 1))$ in expectation.

First, let us formally define the notion of future working set size.

Definition 5.3 (Future Working Set Size future(i)). Let $X = (x(1), \ldots, x(m))$ be any access sequence. We define the Future Working Set Size future(i, x) to be the working set size of the next access of item $x$. That is,

$$\text{future}(i, x) := \text{work}(j), \ \text{where } j = \text{next}(i, x).$$

If $x$ does not appear after time $i$, we define $\text{future}(i, x) := n$. We use $\text{future}(i)$ to denote $\text{future}(i, x(i))$.

Remark 5.4. We will use a simple observation in the following section. At any time $i$, $\text{future}(i - 1, x(i)) = \text{work}(i)$ holds by definition.

Suppose we have a perfect oracle that predicts the exact value of $\text{future}(i)$. We can apply Theorem 1.1 with the score $\text{future}(i, x) + 1$ at each time $i$. We will prove that $\text{future}(i, x)$ is a smooth score. Theorem 1.1 ensures that the expected depth of $x(i)$ is $O(\log(\text{future}(i - 1, x(i)) + 1)) = O(\log(\text{work}(i) + 1))$ by definition. Then, we update $x(i)$’s priority in time $O(\log(\text{work}(i) + 1) + \log(\text{future}(i) + 1))$. However, the next access of $x(i)$ costs $O(\log(\text{future}(i) + 1))$ in expectation. We can charge the update cost to both accesses of the item $x(i)$. The total cost is therefore $O(n \log n + \sum_i \log(\text{work}(i) + 1))$ in expectation.

In practice, one cannot hope for a perfect oracle. But as long as it outputs a crude approximation, we can still achieve the Working-Set Theorem. That is, the oracle outputs a number $s(i)$ crudely approximating $\text{future}(i)$ after $x(i)$ is accessed. We show that the Working-Set Theorem is achieved if $\log(s(i) + 1) = \Theta(\log(\text{future}(i) + 1))$.

Theorem 5.5 (Formal Version of Theorem 1.4). Let $X = (x(1), \ldots, x(m))$ be any access sequence. We are given an oracle that outputs a positive score $s(i)$ after $x(i)$ is accessed. The score $s(i)$ polynomially approximates the future working set size at time $i$, i.e.,

$$C_1 \cdot \log(\text{future}(i) + 1) \leq \log(s(i) + 1) \leq C_2 \cdot \log(\text{future}(i) + 1), \ \ i \in [m]$$

for some universal constants $C_1, C_2 > 0$. Consider the following priority assignment scheme for Treap:

Initially, we set $\text{pri}_{x(i)}(0) := -\lfloor \log_2 \log_2 (n + 1) \rfloor + \delta, \delta \sim U(0, 1)$ for any item $x \in [n]$. After $x(i)$ is accessed, we receive $s(i)$ from the oracle and update $x(i)$’s priority to be

$$\text{pri}_{x(i)}^{(i)} := -\lfloor \log_2 \log_2 (s(i) + 1) \rfloor + \delta, \delta \sim U(0, 1).$$

All other items’ priorities remain the same.

When accessing $x(i)$, the expected depth of item $x(i)$ is $O(\log(\text{work}(i) + 1))$ (Definition 5.2). The expected total cost for processing the access sequence is

$$O \left( n \log n + \sum_i \log(\text{work}(i) + 1) \right).$$
Theorem 1.1 ensures that the expected depth of item \( x(i) \) after its priority updated is \( O(\log(s(i) + 1)) \). It is \( O(\log(\text{future}(i) + 1)) \) because \( s(i) \) polynomially approximates \( \text{future}(i) \). When accessing \( x(i) \), its expected depth is \( O(\log(s(i) - 1, x(i)) + 1)) \), which is \( O(\log(\text{work}(i) + 1)) \). The equality comes from that \( s(i - 1, x(i)) \) crudely approximates \( \text{future}(i - 1, x(i)) \). \text{future}(i - 1, x(i)) \) is exactly \( \text{work}(i) \), the working set size at the next access of \( x \), which is at time \( i \).

Next, we will analyze the total cost carefully because priorities change throughout the process. When oracle outputs \( s(i) \), the T reap changes due to the priority update of item \( x(i) \). We update the T reap by removing \( x \) and inserting it back via Lemma 5.6. The expected cost for removing \( x \) is proportional to its depth, which is \( O(\log(s(i - 1, x(i)) + 1)) \). The expected cost for inserting \( x \) with the updated priority is also proportional to its depth, which is \( O(\log(s(i, x) + 1)) \). Let \( j \) be the next time oracle updates \( x \)’s score. The removing cost of \( x \) at that time is also \( O(\log(\text{work}(i) + 1)) \). Thus, one can charge the insertion cost at time \( i \) to time \( j \). For each item, the first removal and the last insertion cost \( O(\log n) \) each by Definition 5.3. Thus, one can bound the expected total cost by

\[
O \left( n \log n + \sum_i \log(\text{work}(i) + 1) \right).
\]
5.2 Given Maintained Past Working Set Sizes

Given an oracle maintaining estimations on the past working-set sizes, we can still maintain a score-respecting Treap that achieves the Working-Set Theorem.

**Informal Theorem 1.5.** [Working-Set Theorem with Past Working Set Sizes] Given an oracle that maintains $s(i, x)$, a crude approximation to $\text{work}(i, x)$, at any time $i$ for any item $x$. Consider the following priority assignment scheme: At time $i$, after accessing item $x(i)$, we maintain the priority of each item $x$ to be

$$\text{pri}_x := -\lfloor \log_2 \log_2 (s(i, x) + 1) \rfloor + \delta_x, \delta_x \sim U(0, 1).$$

When accessing $x(i)$, the expected depth of item $x(i)$ is $O(\log(\text{work}(i) + 1))$. The total cost, excluding the one for updating priorities, is $O(n \log n + \sum_i \log(\text{work}(i) + 1))$ in expectation.

One can directly get the Working-Set Theorem using Theorem 1.1 with a time-varying score $\text{work}(i, x) + 1$, which is smooth by Observation 3. At any time $i$, the expected depth of item $x$ is $O(\log(\text{work}(i, x) + 1))$. The expected cost for accessing $x(i)$ is $O(\log(\text{work}(i) + 1))$. If we ignore the time spent on maintaining the scores and the Treap priorities, the total cost is $O(n \log n + \sum_i \log(\text{work}(i) + 1))$.

However, the issue with this approach is that $\text{work}(i, x)$ changes dramatically. In the proof of Observation 3, we maintain a list $\pi$ on $[n]$ in the decreasing order of $\text{prev}(i, x)$ and claim that $\text{work}(i, x) + 1$ is exactly the index of $x$ in $\pi$. After some item $x$ is accessed at time $i$, we know that $\text{prev}(i + 1, x) = i$ which is the largest among all items. We move $x$ to the front of the list $\pi$ and increase the working-set sizes of all items in between by one. One might need to update the priorities of $\Omega(n)$ items per access.

It turns out that we don’t need exact values of $\text{work}(i, x)$ to ensure the Working-Set Theorem. Suppose there is another time-varying smooth score $s(i, x)$ that crudely approximates $\text{work}(i, x)$, i.e. $\log(s(i, x) + 1) = \Theta(\log(\text{work}(i, x) + 1))$. We can use $s(i, x)$ in Theorem 1.1 and the expected depth of each item is $O(\log(\text{work}(i, x) + 1))$ at any time $i$. Because $s(i, x)$ are only crude approximations, these scores could be lazily maintained with few updates in total.

In the following theorem, we study the total cost given an oracle that maintains a crude approximation to $\text{work}(i, x)$.

**Theorem 5.7** (Formal version of Theorem 1.5). Let $X = (x(1), \ldots, x(m))$ be any access sequence. We are also given an oracle that maintains a time-varying score $s(i, x)$ that polynomially approximates the working set size $\text{work}(i, x)$, i.e.

$$C_1 \cdot \log(\text{work}(i, x) + 1) \leq \log(s(i, x) + 1) \leq C_2 \cdot \log(\text{work}(i, x) + 1), \ i \in [m], x \in [n],$$

for some universal constants $C_1, C_2 > 0$. Consider the following priority assignment scheme for Treap:

Initially, we set $\text{pri}_x^{(0)} := -\lfloor \log_2 \log_2 (n + 1) \rfloor + \delta_x, \delta_x \sim U(0, 1)$ for any item $x \in [n]$. Upon receiving the access request $x(i)$, the oracle reports $U_i \subseteq [n]$, a subset of items whose score changes, i.e. $s(i, x) \neq s(i - 1, x), x \in U_i$. We update the priority of each item $x \in U_i$ to be:

$$\text{pri}_x^{(i)} := -\lfloor \log_2 \log_2 (s(i, x) + 1) \rfloor + \delta, \delta \sim U(0, 1).$$

All other items’ priorities remain the same.

When accessing $x(i)$, the expected depth of item $x(i)$ is $O(\log(\text{work}(i) + 1))$. The expected total cost for processing the access sequence is

$$O \left( n \log n + \sum_i \log(\text{work}(i) + 1) + \sum_i \sum_{x \in U_i} \log(s(i, x) + 1) \right).$$
The proof is an application of Theorem 1.1. In this case, the score function dynamically changes as time goes on, unlike Theorem 4.1 where the score is fixed beforehand.

First, we show that \( s(i, x) \) is smooth for fixed \( i \).

Lemma 5.8. At any time \( i \), \( s(i, x) + 1 \) is a smooth score on \([n]\).

Proof. For any \( k \geq 0 \), observe that \( s(i, x) + 1 \leq k \) implies that \( (\text{work}(i, x) + 1)^{C_1} \leq k \). This comes from the assumption on \( s(i, x) \). \( \text{work}(i, x) \) is smooth at any time \( i \) by Observation 3. Therefore, the number of items with \( (\text{work}(i, x) + 1)^{C_1} \leq k \) is \( O(\text{poly}(k^{1/C_1})) = O(\text{poly}(k)) \) since \( C_1 \) is a constant.

This also bounds the number of items with \( s(i, x) + 1 \leq k \) and concludes the proof.

Now, we are ready to prove Theorem 5.7.

Proof of Theorem 5.7. Theorem 1.1 and Lemma 5.8 ensures that the expected depth of item \( x \) after its priority updated is \( O(\log(s(i, x) + 1)) = O(\log(\text{work}(i, x) + 1)) \). When accessing \( x(i) \), the expected depth of \( x(i) \) is \( O(\log(\text{work}(i) + 1)) \) by Definition 5.2. The expected total cost for processing the access sequence is \( O(\sum_i \log(\text{work}(i) + 1)) \).

Next, we will analyze the total cost carefully because priorities change throughout the process. When oracle reports updates in scores, the Treap changes due to priorities updates. Let \( x \in U_i \) be an item whose score is updated. We update the Treap by removing \( x \) and inserting it back via Lemma 2.3. The expected cost for removing \( x \) is proportional to its depth, which is \( O(\log(s(i-1, x) + 1)) \). The expected cost for inserting \( x \) with the updated priority is also proportional to its depth, which is \( O(\log(s(i, x) + 1)) \). Let \( j \) be the next time oracle updates \( x \)’s score. The removing cost of \( x \) at time \( j \) is also \( O(\log(s(i, x) + 1)) \) because \( x \)’s priority is not yet updated. Thus, one can charge the insertion cost at time \( i \) to time \( j \). For each item, the first removal and the last insertion cost \( O(\log n) \) each by Definition 5.2. Thus, one can bound the expected total cost for maintaining the Treap by

\[
O \left( n \log n + \sum_i \sum_{x \in U_i} \log(s(i, x) + 1) \right).
\]

6 Conclusion

In this paper, we introduce score-respecting Treaps for building the first learning-augmented binary search trees against arbitrary input. With proper predictions, score-respecting Treaps achieve either static optimality or the Working-Set Theorem. Moreover, our learning-augmented BSTs work for any input and are robust to prediction errors.

Our work opens up two directions for future work. One is to investigate what other performance measures learning-augmented BSTs can achieve such as the Static/Dynamic Finger Theorem, the Unified Bound, or even the Dynamic Optimality. A key to these questions is to identify the right value or object to predict. The other direction is to design other learning-augmented data structures that work with arbitrary input.

Acknowledgments

The authors are very grateful to Richard Peng and Danny Sleator for useful discussions, suggestions, and comments on earlier drafts and the presentation of these ideas.
References

[AL63] M Adelson-Velskii and Evgenii Mikhailovich Landis. *An algorithm for the organization of information*. Tech. rep. Joint Publications research service Washington DC, 1963 (cit. on p. 4).

[APT22] 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)*. SIAM. 2022, pp. 35–66 (cit. on p. 1).

[AS89] Cecilia R Aragon and Raimund Seidel. “Randomized search trees”. In: *FOCS*. Vol. 30. 1989, pp. 540–545 (cit. on pp. 2, 5, 6).

[Ban20] Shom Banerjee. “Improving online rent-or-buy algorithms with sequential decision making and ML predictions”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 21072–21080 (cit. on p. 4).

[Bay72] Rudolf Bayer. “Symmetric binary B-trees: Data structure and maintenance algorithms”. In: *Acta informatica* 1.4 (1972), pp. 290–306 (cit. on p. 4).

[BCDI07] Mihai Bădoiu, Richard Cole, Erik D Demaine, and John Iacono. “A unified access bound on comparison-based dynamic dictionaries”. In: *Theoretical Computer Science* 382.2 (2007), pp. 86–96 (cit. on p. 5).

[BCIKL20] 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)*. SIAM. 2020, pp. 1878–1891 (cit. on p. 5).

[BCK03] Avrim Blum, Suchi Chawla, and Adam Tauman Kalai. “Static optimality and dynamic search-optimality in lists and trees”. In: *Algorithmica* 36.3 (2003), pp. 249–260 (cit. on p. 5).

[BD04] Mihai Bădoiu and Erik D Demaine. “A simplified and dynamic unified structure”. In: *Latin American Symposium on Theoretical Informatics*. Springer. 2004, pp. 466–473 (cit. on p. 5).

[BFL22] Joan Boyar, Lene M Favrholdt, and Kim S Larsen. “Online Unit Profit Knapsack with Untrusted Predictions”. In: *arXiv preprint arXiv:2203.00285* (2022) (cit. on p. 4).

[CCF02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. “Finding frequent items in data streams”. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2002, pp. 693–703 (cit. on p. 4).

[CEILNRSWWZ22] 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”. In: *arXiv preprint arXiv:2203.09572* (2022) (cit. on p. 4).

[CI21] Justin Y Chen and Piotr Indyk. “Online Bipartite Matching with Predicted Degrees”. In: *arXiv preprint arXiv:2110.11439* (2021) (cit. on p. 4).
[CM05] Graham Cormode and Shan Muthukrishnan. “An improved data stream summary: the count-min sketch and its applications”. In: *Journal of Algorithms* 55.1 (2005), pp. 58–75 (cit. on pp. 1, 4).

[CM89] Joseph Culberson and J. Ian Munro. “Explaining the Behaviour of Binary Search Trees Under Prolonged Updates: A Model and Simulations¶”. In: *The Computer Journal* 32.1 (1989), pp. 68–75 (cit. on p. 4).

[CM90] Joseph Culberson and J Ian Munro. “Analysis of the standard deletion algorithms in exact fit domain binary search trees”. In: *Algorithmica* 5.1 (1990), pp. 295–311 (cit. on p. 4).

[CMSS00] 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”. In: *SIAM Journal on Computing* 30.1 (2000), pp. 1–43 (cit. on p. 4).

[Col00] Richard Cole. “On the dynamic finger conjecture for splay trees. Part II: The proof”. In: *SIAM Journal on Computing* 30.1 (2000), pp. 44–85 (cit. on p. 5).

[CSVZ22] Justin Chen, Sandeep Silwal, Ali Vakilian, and Fred Zhang. “Faster fundamental graph algorithms via learned predictions”. In: *International Conference on Machine Learning*. PMLR. 2022, pp. 3583–3602 (cit. on p. 4).

[DHIKP09] 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)*. SIAM. 2009, pp. 496–505 (cit. on p. 5).

[DHIP07] Erik D Demaine, Dion Harmon, John Iacono, and Mihai Patraşcu. “Dynamic optimality—almost”. In: *SIAM Journal on Computing* 37.1 (2007), pp. 240–251 (cit. on p. 5).

[DILMV21] Michael Dinitz, Sungjin Im, Thomas Lavastida, Benjamin Moseley, and Sergei Vassilvitskii. “Faster matchings via learned duals”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 10393–10406 (cit. on p. 4).

[DMYWDLZCGK+20] 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*. 2020, pp. 969–984 (cit. on p. 1).

[DS09] 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*. Springer. 2009, pp. 194–205 (cit. on p. 5).

[EFSWZ21] Jon Ergun, Zhili Feng, Sandeep Silwal, David P Woodruff, and Samson Zhou. “Learning-Augmented k-means Clustering”. In: *arXiv preprint arXiv:2110.14094* (2021) (cit. on p. 1).

[ELMS20] Thomas Erlebach, Murilo S de Lima, Nicole Megow, and Jens Schlöter. “Learning-Augmented Query Policies”. In: *arXiv preprint arXiv:2011.07385* (2020) (cit. on p. 4).
[KBCDP18] 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. 2018, pp. 489–504 (cit. on pp. 1, 4).

[KBTV22] Mikhail Khodak, Maria-Florina Balcan, Ameet Talwalkar, and Sergei Vassilvitskii. “Learning predictions for algorithms with predictions”. In: arXiv preprint arXiv:2202.09312 (2022) (cit. on p. 4).

[LLW22] Honghao Lin, Tian Luo, and David Woodruff. “Learning Augmented Binary Search Trees”. In: International Conference on Machine Learning. PMLR. 2022, pp. 13431–13440 (cit. on pp. 2–4, 10).

[LMRX20] Thomas Lavastida, Benjamin Moseley, R Ravi, and Chenyang Xu. “Learnable and instance-robust predictions for online matching, flows and load balancing”. In: arXiv preprint arXiv:2011.11743 (2020) (cit. on p. 4).

[LRSZ21] Pinyan Lu, Xuandi Ren, Enze Sun, and Yubo Zhang. “Generalized Sorting with Predictions”. In: Symposium on Simplicity in Algorithms (SOSA). SIAM. 2021, pp. 111–117 (cit. on p. 4).

[Meh75] Kurt Mehlhorn. “Nearly optimal binary search trees”. In: Acta Informatica 5.4 (1975), pp. 287–295 (cit. on p. 4).

[Mit19] Michael Mitzenmacher. “Scheduling with predictions and the price of mis-prediction”. In: arXiv preprint arXiv:1902.00732 (2019) (cit. on p. 4).

[MV20] Michael Mitzenmacher and Sergei Vassilvitskii. “Algorithms with predictions”. In: arXiv preprint arXiv:2006.09123 (2020) (cit. on pp. 1, 4).

[NR72] Jürg Nievergelt and Edward M Reingold. “Binary search trees of bounded balance”. In: Proceedings of the fourth annual ACM symposium on Theory of computing. 1972, pp. 137–142 (cit. on p. 5).

[PSK18] Manish Purohit, Zoya Svitkina, and Ravi Kumar. “Improving online algorithms via ML predictions”. In: Advances in Neural Information Processing Systems 31 (2018) (cit. on p. 4).

[PZ22] Adam Polak and Maksym Zub. “Learning-Augmented Maximum Flow”. In: arXiv preprint arXiv:2207.12911 (2022) (cit. on p. 4).

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. “Self-adjusting binary search trees”. In: Journal of the ACM (JACM) 32.3 (1985), pp. 652–686 (cit. on pp. 1, 5).

[Wil89] Robert Wilber. “Lower bounds for accessing binary search trees with rotations”. In: SIAM journal on Computing 18.1 (1989), pp. 56–67 (cit. on p. 5).

[WLW20] Shufan Wang, Jian Li, and Shiqiang Wang. “Online algorithms for multi-shop ski rental with machine learned advice”. In: Advances in Neural Information Processing Systems 33 (2020), pp. 8150–8160 (cit. on p. 4).

[WZCWCX21] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. “Updatable learned index with precise positions”. In: Proceedings of the VLDB Endowment 14.8 (2021), pp. 1276–1288 (cit. on p. 1).
[XM22] Chenyang Xu and Benjamin Moseley. “Learning-augmented algorithms for online steiner tree”. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 36. 8. 2022, pp. 8744–8752 (cit. on p. 4).

[ZSHW21] Ali Zeynali, Bo Sun, Mohammad Hajiesmaili, and Adam Wierman. “Data-driven competitive algorithms for online knapsack and set cover”. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 35. 12. 2021, pp. 10833–10841 (cit. on p. 4).