The Group Access Bounds for Binary Search Trees

Parinya Chalermsook
Aalto University, Finland

Manoj Gupta
IIT Gandhinagar, India

Wanchote Jiamjitrak
University of Helsinki, Finland

Akash Pareek
IIT Gandhinagar, India

Sorrachai Yingchareonthawornchai
The Hebrew University of Jerusalem, Israel

Abstract

The access lemma (Sleator and Tarjan, JACM 1985) is a property of binary search trees (BSTs) that implies interesting consequences such as static optimality, static finger, and working set property on any access sequence $X = (x_1, x_2, \ldots, x_m)$. However, there are known corollaries of the dynamic optimality that cannot be derived via the access lemma, such as the dynamic finger, and any $o(\log n)$-competitive ratio to the optimal BST where n is the number of keys.

In this paper, we introduce the *group access bound* that can be defined with respect to a reference *group access tree*. Group access bounds generalize the access lemma and imply properties that are far stronger than those implied by the classical access lemma. For each of the following results, there is a group access tree whose group access bound

1. Is $O(\sqrt{\log n})$-competitive to the optimal BST.
2. Achieves the k-finger bound with an *additive* term of $O(m \log k \log\log n)$ (randomized) when the reference tree is an almost complete binary tree.
3. Satisfies the unified bound with an *additive* term of $O(m \log\log n)$.
4. Matches the unified bound with a time window k with an *additive* term of $O(m \log k \log\log n)$ (randomized).

Furthermore, we prove the simulation theorem: For every group access tree, there is an online BST algorithm that is $O(1)$-competitive with its group access bound. In particular, any new group access bound will automatically imply a new BST algorithm achieving the same bound. Thereby, we obtain an improved k-finger bound (reference tree is an almost complete binary tree), an improved unified bound with a time window k, and matching the best-known bound for Unified bound in the BST model. Since any dynamically optimal BST must achieve the group access bounds, we believe our results provide a new direction towards proving $o(\log n)$-competitiveness of the Splay tree and Greedy, two prime candidates for the dynamic optimality conjecture.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Dynamic Optimality, Binary Search Tree, Online Algorithm

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.38

Category Track A: Algorithms, Complexity and Games

Related Version *Full version:*: `https://arxiv.org/abs/2312.15426`

Funding This research was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 759557.

1 Introduction

In the amortized analysis of a self-adjusting binary search tree (BST), the *access lemma* [32] is perhaps the most fundamental property of a BST algorithm that allows us to prove the competitiveness against many performance benchmarks, including temporal and spatial locality. If a BST algorithm satisfies the access lemma, then, by plugging in appropriate parameters, the algorithm also satisfies many interesting corollaries of the *dynamic optimality conjecture* including, for example, *balance theorem [32], static optimality [32, 20], static finger property [32], working set property [32], and key-independent optimality [19].* For example, Splay tree [32], Greedy [14, 30], and Multi-splay tree [34] are all known to satisfy the access lemma [32, 20, 34].

Despite these many applications, several strong BST properties cannot be implied via the access lemma, including *"non-trivial" competitiveness*, *k-finger property* (or even the (weaker) dynamic finger property [13]), and the *unified bound* [24, 17]. For completeness, we discuss each one of them in turn.

Competitiveness. Dynamic optimality conjecture [32] postulates that there is an online binary search tree (BST) on n keys whose cost to perform a search (access) sequence $(x_1, \ldots, x_m) \in \{1, 2, \ldots, n\}^m$ (including the cost to adjust the internal structure in between the sequence) is at most that of the offline optimum up to a constant factor. We say that a BST algorithm is $f(n)$-*competitive* if its total cost is, at most, the cost of the offline optimum up to a factor of $f(n)$. A BST algorithm is dynamically optimal if it is $O(1)$-competitive.

Splay tree [32] and Greedy [14] are widely regarded as the prime candidates for dynamic optimality conjecture. However, the best-known competitiveness of both the algorithms remains $O(\log n)$, which can be shown from the access lemma or (via static optimality) using any balanced trees. The access lemma cannot be used to prove $o(\log n)$-competitive (for example, we cannot even derive the *sequential access theorem* [33] via the access lemma). In contrast, there are BST algorithms with $O(\log \log n)$-competitiveness. In [15], the authors presented the first $O(\log \log n)$-competitive binary search tree, which they call Tango Trees. Several subsequent results provided alternate $O(\log \log n)$-competitive algorithms [21, 35, 3, 11]. Despite achieving the best-known competitive ratios, the fact that these algorithms do not satisfy many corollaries of the dynamic optimality conjecture makes them less promising than Splay and Greedy.

***k*-Finger Bound.** Several notions of finger bounds [32, 13, 26] have been introduced to capture the "locality" of input sequences. The strongest finger bound, called k-Finger bound, was motivated by the connection between BSTs and the k-server problem [8]. It has still remained unclear whether any online BST algorithm achieves this property.

We define k-Finger bound as follows. Assume we have an almost complete binary tree where each leaf represents a distinct key from $\{1, \ldots n\}$. This tree is called the *reference tree*. Assume that there are k-fingers stationed at k arbitrary leaves in the reference tree.

▶ **Definition 1 (k-Finger Bound).** *When a key x_t is searched at time t, we must move one finger from its current position to the node containing x_t. The cost of the search is the number of nodes on the unique path connecting the finger's source to its destination. We define $F^k(X)$ as the minimum, overall finger movement strategies distance traversed by the fingers to process the sequence X when the reference tree is an almost complete binary tree.*

The classical access lemma cannot imply the k-finger property even when $k = 1$ (called the lazy finger bound [26], which generalizes the dynamic finger bound [13]). Nonetheless, it is possible to prove a non-trivial bound w.r.t. k-Finger using different techniques. In [8], the authors claimed the existence of an online BST algorithm with cost $O((\log k)^7 \mathrm{F}^k(X))$. However, this claim has an error since the algorithm employs Lee's [29] k-server result, which the author has retracted. Instead of Lee's result, we can use the k-server result of Koutsoupias and Papadimitriou [27], $(2k-1)$-competitive. By using [8], it implies that an online BST algorithm exists with a running time of $O(k\mathrm{F}^k(X))$. This is a relatively large gap when compared with the best achievable offline BST bound of $O(\log k)F^k(X)$ [8]; in fact, whether there exists a BST whose cost is additive in the k-finger bound, that is $O(F^k(X) + m \log k)$, has remained open.

Unified Bound. To describe the unified bound, we should first understand the *working set bound* [25, 32] and the *dynamic finger bound* [13, 32, 26]. If x_t is a search key, then the working set bound requires x_t to be searched in amortized time $O(\log \textsc{Ws}(x_t))$, where $\textsc{Ws}(x_t)$ is the number of distinct keys searched since the last search of x_t. The working set bound is based on temporal locality and implies many more bounds, such as the static finger bound, the static optimality bound, etc. The dynamic finger bound states that the amortized time to search x_t is $O(\log |x_t - x_{t-1}| + 2)$. The dynamic finger bound is based on spatial locality.

The *unified bound* [24] implies both the working set and the dynamic finger.

▶ **Definition 2.** *The unified bound can be defined as* $\textsc{Ub}(X) = \sum_{t=2}^{m} \log(\min_{t'<t}\{t - t' + |x_t - x_{t'}| + 2\})$.

The unified bound is stronger than both the working set and the dynamic finger, as it suggests that it is in-expensive to search a key that is close to a recently searched key. It is true that the access lemma cannot prove the unified bound, although there are data structures that satisfy the unified bound. Iacono [24] gave a comparison-based data structure called the unified structure that achieves the unified bound. In [1], the authors gave a dynamic comparison-based data structure that achieves the unified bound. Derryberry and Sleator [17] designed the first BST algorithm called Skip-Splay tree that nearly achieves the unified bound with its running time of $O(\textsc{Ub}(X) + m \log\log n)$. In [4], the authors modified Skip-Splay using layered working-set trees to get amortized $O(\textsc{Ub}(x_i) + \log\log n)$ time, where x_i is a search key at time i. In Derryberry's thesis [18], Cache trees were introduced as the first BST algorithm aiming to achieve the Unified bound. However, this assertion was later questioned by Sleator in [31]. The question of whether a BST algorithm can truly achieve the Unified bound remains open and intriguing.

Unified Bound with time window. In the unified bound, for each x_t, we find a key $x_{t'}$ that minimizes the term $(t - t' + |x_t - x_{t'}| + 2)$ where $t' < t$. We can add another condition that t' should be one of the last k searched keys before time t. Thus, t' comes from some time window. We call this variant *Unified Bound with a time window.* We formally define it as follows.

▶ **Definition 3.** *Given an integer k, the unified bound with a time window is*

$$\textsc{Ub}^k(X) = \sum_{t=2}^{m} \log\left(\min_{t' \in [t-k \dots t-1]} \{t - t' + |x_t - x_{t'}| + 2\}\right).$$

This bound can be seen as "interpolating" between the dynamic finger and the unified bounds: When $k = 1$, $\textsc{Ub}^k(X)$ is the dynamic finger bound, and when $k = m$, $\textsc{Ub}^m(X)$ is the unified bound. For $k = 1$, we know that both Splay tree and $\textsc{Greedy}$ satisfy the dynamic finger bound. The authors of [8], showed a relation between $\textsc{Ub}^k(X)$ and $\textsc{Opt}(X)$, i.e,

$$\textsc{Opt}(X) \leq \beta(k)\textsc{Ub}^k(X)$$

where $\beta(\cdot)$ is some fixed (super exponentially growing) function.

In sum, the access lemma is an intrinsic property of a BST which implies nice properties but cannot seem to imply any of the aforementioned properties.

1.1 Our New Concept: The Group Access Bounds

The main conceptual contribution of this paper is to introduce the *group access bounds* that generalize the bound from the access lemma. Informally, the access lemma states that the amortized cost to access x_t at time is $O\left(\log \frac{W}{w(x_t)}\right)$, where $w(x_t)$ is the weight of x_t and W is the sum of weights of all the keys in the BST.

We give an informal definition of the group access bounds (see Section 3 for the formal definitions). Denote $[n] = \{1, \ldots, n\}$. The key gadget to describe our bound is the notion of *group access tree*, which captures a hierarchical partition of $[n]$ until singletons are obtained. That is, in a group access tree $\mathcal{T}$, each node $v \in V(\mathcal{T})$ is associated with an "interval" $I_v \subseteq [n]$ (consecutive integers). The root $r \in V(\mathcal{T})$ has $I_r = [n]$, if a node v has children $v_1, \ldots, v_k$, then we have that $\{I_{v_1}, \ldots, I_{v_k}\}$ forms a partition of I_v. This process continues until each leaf $v \in V(\mathcal{T})$ is associated with a singleton. Note that the tree does not have to be binary. See Figure 1 for illustration.

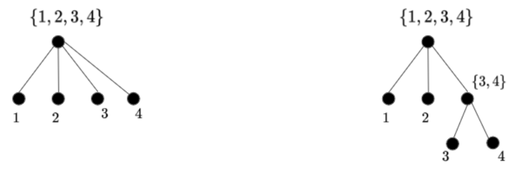

Figure 1 Examples of two group access trees. Each represents a hierarchical partition of $[n]$ until singletons are obtained. When a group access tree is a star (LHS), our bound is simply the access lemma.

Let w be a positive weight function that assigns a real-valued weight to each node in $\mathcal{T}$. We define the cost to access the tree $\mathcal{T}$ w.r.t. the weight function w as follows. For any edge (u, v) where u is the parent of v, the cost on (u, v) is $\log \frac{W(u)}{w(v)}$ where $W(u)$ is the total weight of all the children of u. The access cost of a key $a \in [n]$, denoted as $\text{cost}_{\mathcal{T},w}(a)$, is defined as the total cost of all the edges in the path P_a from the root to the leaf containing a in the group access tree $\mathcal{T}$. That is,

$$\text{cost}_{\mathcal{T},w}(a) = \sum_{(u,v)\in P_a} \log \frac{W(u)}{w(v)}.$$

Some readers may have observed the similarity between our cost function and that of the access lemma. Indeed, one can show that it generalizes the access lemma.

▶ **Observation 4.** *If the group access tree $\mathcal{T}$ is a star, then the access $a \in [n]$ on $\mathcal{T}$ gives the same (amortized) cost as the access lemma on the same weight function.*

Intuitively, the group access bound offers a "search tree" (which is not necessarily binary) where the cost of searching a is the sum of the cost of the edges on the search path P_a.

Let $\mathcal{W} = (w^{(1)}, \dots, w^{(m)})$ be a sequence of weight functions. The total cost of the group access tree $\mathcal{T}$ on an access sequence $X = (x_1, \dots, x_m)$, where $x_t \in [n]$ for all t, is

$$\mathrm{cost}_{\mathcal{T},\mathcal{W}}(X) = \sum_t \mathrm{cost}_{\mathcal{T},w^{(t)}}(x_t).$$

Similar to the access lemma, the weight functions should change in a controllable manner in order to be meaningful (e.g., the weight functions for the working set bound are changed in a structured way in the access lemma). Here, we introduce the notion of *locally bounded* weight families. We say that a sequence of weight functions $\mathcal{W} = (w^{(1)}, \dots, w^{(m)})$ is *locally bounded* if for all t, the weight increase from time t to $t+1$ can happen only at the nodes in the path from root to x_t in the group access tree $\mathcal{T}$.

Given an input sequence $X = (x_1, \dots, x_m)$, *the group access bound* $\mathsf{GAB}(\mathcal{T}, X)$ w.r.t. a group access tree $\mathcal{T}$ is

$$\mathsf{GAB}(\mathcal{T}, X) = \min_{\mathcal{W} \text{ locally bounded}} \mathrm{cost}_{\mathcal{T},\mathcal{W}}(X).$$

1.2 Our Technical Results: Deriving BST Bounds from GAB

We show that the cost of the group access tree (with respect to certain weight functions) is competitive against many strong BST bounds that are not known via the access lemma. We say a group access bound is *randomized* if the group access tree $\mathcal{T}$ is obtained by a random process that iteratively partitions $[n]$ until singletons are obtained.

▶ **Theorem 5.** *For each of the following bounds, there exists deterministic group access trees $\mathcal{T}_1, \mathcal{T}_3$ and randomized group access trees $\mathcal{T}_2, \mathcal{T}_4$ such that for all access sequences $X = (x_1, \dots, x_m)$ where $x_t \in [n]$ for all t,*

1. $\mathsf{GAB}(\mathcal{T}_1, X) = O(\sqrt{\log n}) \cdot \textsc{Opt}(X)$ *where $\textsc{Opt}(X)$ is the cost of offline optimal BST on X.*
2. $\mathsf{GAB}(\mathcal{T}_2, X) = O(F^k(X) + m \log k \log\log n)$ *(randomized). That is, it is competitive with k-finger up to an additive term when the reference tree is an almost complete binary tree.*
3. $\mathsf{GAB}(\mathcal{T}_3, X) = O(\textsc{Ub}(X) + m \log\log n)$*. That is, it is competitive with the unified bound up to an additive term.*
4. $\mathsf{GAB}(\mathcal{T}_4, X) = O(\textsc{Ub}^k(X) + m \log k \log\log n)$ *(randomized). That is, it is competitive with unified bound with a time window up to an additive term.*

The group access tree for the second and the fourth bound can be efficiently constructed from a probability distribution.

It is not immediately clear that these group access bounds can be realized by BST algorithms. We show that every group access tree $\mathcal{T}$ can be simulated by an online BST algorithm. Let $\mathcal{A}$ be a BST algorithm. We denote $\mathrm{cost}_{\mathcal{A}}(X)$ to be the cost of algorithm $\mathcal{A}$ running on a sequence $X = (x_1, \dots, x_m)$.

▶ **Theorem 6** (Simulation Theorem). *For any group access tree $\mathcal{T}$, there exists an online BST algorithm $\mathcal{A}$ such that for any sufficiently long access sequence X , $\mathit{cost}_{\mathcal{A}}(X) = O(\mathsf{GAB}(\mathcal{T}, X))$. Furthermore, if the group access tree $\mathcal{T}$ is randomized, then $\mathcal{A}$ is randomized, where the competitive ratio is measured in the oblivious adversary model.*

The Simulation Theorem (Theorem 6) signifies that one can prove new BST bounds by just proving the existence of a group access tree along with locally bounded weight families.

Table 1 Summary of the new bounds that can be derived via the group access bound. The first two columns show the best-known results prior to this paper. Asymptotic notations (Big-Oh and Big-Omega) are hidden for brevity.

	Offline Upper	**Online Upper**	**Our new bounds**
OPT	OPT $\log\log n$ [15]	OPT $\log\log n$ [15]	OPT$\sqrt{\log n}$
F^k	$F^k(X)\log k$ [8]	$kF^k(X)$ [8]	$F^k(X) + m\log k \log\log n$
UB	$\mathsf{UB}(X) + m\log\log n$ [17]	$\mathsf{UB}(X) + m\log\log n$ [17]	$\mathsf{UB}(X) + m\log\log n$
UB^k	$\beta(k)\mathsf{UB}^k(X)$ [8]	$\beta(k)\mathsf{UB}^k(X)$ [8]	$\mathsf{UB}^k(X) + m\log k\log\log n$

1.3 Significance of our results

Beyond the access lemma, group access bounds serve as the first step towards providing a unified framework for proving binary search tree bounds systematically. To resolve the dynamic optimality conjecture, a candidate algorithm must satisfy all dynamic optimality corollaries simultaneously, and we believe group access bounds are the starting point for this.

Apart from the competitiveness result, the results we present here either match the best-known bounds (such as unified bound) or provide an improvement upon even the best-known offline algorithms (k-finger bound when reference tree is an almost complete binary tree and unified bound with bounded time window) in the BST model.

We remark that even though our competitive ratio does not match the $O(\log\log n)$ best-known factor, what we prove is more general where any BST algorithm satisfying the GAB will automatically be $O(\sqrt{\log n})$- competitive. The BST algorithm we present here is called GGREEDY, a very similar algorithm to GREEDY – a prime candidate for dynamic optimality. GGREEDY resembles Greedy and inherits its conceptual simplicity. Even after a lot of work in this area [20, 26, 5, 7, 6, 8, 11, 22, 23, 17, 28, 9], only the trivial bound for GREEDY is known: GREEDY$(X) = O(\log n)$ OPT(X) for all possible X. Thus, GREEDY is $O(\log n)$-competitive. Our result raises some hope of proving $o(\log n)$-competitive ratio for Greedy by showing that Greedy satisfies the group access bound.

We illustrate the power of the group access bound by deriving two new BST bounds that have not been known for even offline BST algorithms.

▶ **Corollary 7.** *For each of the following items, there is a randomized online BST algorithm $\mathcal{A}$ such that the expected cost for any search sequence $X = (x_1, \dots, x_m)$ where $x_t \in [n]$ for all t,*

- *$cost_{\mathcal{A}}(X) = O(F^k(X) + m\log k\log\log n)$, and*
- *$cost_{\mathcal{A}}(X) = O(\mathrm{UB}^k(X) + m\log k\log\log n)$.*

The algorithms are randomized because the group access trees are chosen based on a probability distribution. The bound $O(F^k(X) + m\log k\log\log n)$ gives an improvement from the best known online BST that costs $O(kF^k(X))$ in [8] and improves upon the offline bound $O(\log k)F^k(X)$ for some range of parameter k when the reference tree is an almost complete binary tree.

From such an improvement, we can derive a new "pattern-avoiding" bound for BSTs. We say that a sequence *contains* another sequence (or pattern) π if it contains a subsequence that is order-isomorphic to π.

▶ **Corollary 8.** *Let $X \in [n]^m$ be a sequence that does not contain the pattern $(k, k-1, \dots, 1)$. Then there exists a randomized BST that accesses X with cost $O(nk + m\log k\log\log n)$.*

This bound improves the best-known online algorithm [9] that gives a bound of $O(mk^2)$ and the best-known bound on the offline optimum $O(mk)$ [8] for some range of parameters when the reference tree used to calculate $F^k(X)$ is an almost complete binary tree.

The unified bound obtained in this paper matches the best-known bound of $O(\textsc{Ub}(X) + m \log\log n)$ by Derryberry and Sleator [17]. In fact, we show that we can use the analysis of [17] as a black box once we obtain the group access tree for Unified bound.

We do expect more applications of the group access bounds in binary search trees since group access bounds are generic and yet (unlike the dynamic optimality conjecture) maintain a certain flavor of being "static" (since $\mathcal{T}$ is still fixed). We show that this static component can be algorithmically leveraged. We believe that our bound offers a "bridge" between the relatively static access lemma to the dynamic optimality conjecture, which requires a full understanding of dynamic BSTs.

Contribution to potential function analysis. Another interesting aspect of our work is that once the group access bound is formulated, the proof relies solely on the use of the standard Sum-of-logs potential function, which in general, does not seem sufficient to prove any strong bounds beyond the access lemma. We show that by "augmenting" the access lemma with a group access tree, a natural and standard sum-of-logs potential function immediately provides significantly stronger BST bounds.

In general, designing a potential function for analyzing a given algorithm is a highly innovative but rather ad-hoc task. Our work suggests that the sum-of-logs potential function on the group access tree might be a good candidate for proving that Greedy or Splay is $O(\sqrt{\log n})$-competitive.

Combining BSTs. In [16], the authors showed that different BSTs with well-known bounds can be combined into a single BST that achieves all the properties of the combined BSTs. For example, Tango trees and Skip-Splay trees can be combined to get a BST, which is $O(\log\log n)$-competitive and achieves the Unified bound with an additive term of $O(\log\log n)$. The combined BST from their approach, however, results in a different BST algorithm. Our group access bounds offer another way to combine known BST bounds, as we have illustrated in the above discussion, while retaining the original algorithm, e.g., proving that Splay satisfies the group access bound implies that Splay itself possesses all the nice properties derived from GAB.

1.4 Concluding Remarks

We propose the group access bound – a far-reaching extension of the standard access lemma and present applications in deriving new and unifying old bounds. Some of our bounds even improve the best-known upper bound on the offline optimum on some range of parameters.

An immediate (and perhaps most interesting) open question is whether Greedy or Splay satisfies the group access bound via the sum-of-logs potential function. We believe that this question is very concrete (since it involves a specific potential function), so proving or refuting it would not be beyond our reach.

Developing further understanding and finding more applications of our group access bounds are interesting directions. For instance, what are other BST bounds that can be implied by the group access bounds? Can we show that $\mathsf{GAB}(\mathcal{T}, X) \le O(\mathsf{OPT}(X))$ for some (distribution of) group access tree $\mathcal{T}$? Can one derive a non-trivial result about more general pattern-avoiding bounds [6, 23, 10, 12]? It was shown that optimal BST takes linear time for a pattern of bounded length [2]. Can we design a BST algorithm with a running time $O(\mathrm{F}^k(X) + m \log k)$ for any sequence X of length m where the k-finger bound is calculated on any arbitrary reference tree?

There are also open questions to settle the complexity of specific BST bounds both in the online and offline settings. Most notably, is there any BST data structure that satisfies the unified bound?

1.5 Organization

We introduce notation and terminology in Section 2. We formalize the description of the group access bounds in Section 3. We prove $O(\sqrt{\log n})$-competitiveness in Section 4. Owing to space limitations, the k-finger bound, unified bound and unified bound with a time window can be found in the full version of the paper. To prove the Simulation theorem (Theorem 6), we first define an algorithm, called GGREEDY, that simulates the group access tree in Section 5. We derive the group access lemma in Section 6 and finally prove the Simulation theorem (See to the full version of the paper). Certain proofs and sections are removed which can be found in the full version of the paper.

2 Preliminaries

Let $X = (x_1, x_2, \ldots, x_m)$ be a sequence of m accesses where each access is from the set $\{1, 2, ..., n\}$. This sequence can be represented as points in the plane, that is, $X_p = \{(x_t, t) : t \in [m]\} \subseteq \mathbb{R}^2$. Imagine these points on a plane with an origin and X-Y axis. Since both the coordinates of a point are positive, all points lie in the first quadrant. The positive x-axis represents the key space, and the y-axis represents time. For any two points p, q in a point-set P, if they are not in the same horizontal or vertical line, we can form a rectangle $\square pq$. A rectangle $\square pq$ is said to be *arborally satisfied* if $\exists r \in P \setminus \{p, q\}$ such that r lies in $\square pq$. [14] introduced us to the following beautiful problem:

▶ **Definition 9** (Arborally Satisfied Set). *Given a point set X_p, find a point set Y such that $|X_p \cup Y|$ is minimum and every pair of points in $X_p \cup Y$ is arborally satisfied.*

[14] showed that finding the best BST execution for a sequence X is equivalent to finding the minimum cardinality set Y such that $X_p \cup Y$ is arborally satisfied.

▶ **Lemma 10** (See Lemma 2.3 in [14]). *Let A be an online algorithm that outputs an arborally satisfied set on any input representing X. Then, there is an online BST algorithm A' such that the cost of A' is asymptotically equal to the cost of A, where the cost of A is the number of points added by A plus the size of X.*

At time t, we say that x_t is an *access key*. An algorithm adds (or touches) *points* while processing a key with the aim of making the final point set arborally satisfied. For a point p, denote $p.x$ and $p.y$ as its x-coordinate and y-coordinate, respectively. Note that $p.x$ denotes a key, and $p.y$ denotes the time when the point p was added.

Let q be a key. When we say that GREEDY (or any other algorithm) adds a *point at key* q at time t, it means that GREEDY adds a point at coordinates (q, t). Given two points p_1 and p_2, p_2 lies to the right of p_1 if $p_2.x > p_1.x$, else it lies to the left of p_1. For brevity, we will avoid using ceil and floor notation for various parameters used in this paper.

3 The Group Access Bound

In this section, we describe the *group access bound*, which generalizes the access lemma. The concept of group access bound consists of two ingredients:

- **Hierarchical partition**: An **interval partition** of $[n]$ is a partition π such that each set $S \in \pi$ is an interval (i.e. consecutive integers). Let π be an interval partition of $[n]$. We say that π is a refinement of another partition π' if for all $S \in \pi$ and $S' \in \pi'$, we have $S \subseteq S'$ or $S \cap S' = \emptyset$. For instance $\{\{1,2\},\{3,4\},\{5,6\}\}$ is a refinement of $\{\{1,2,3,4\},\{5,6\}\}$.
  A hierarchical partition is a sequence of partitions $\mathcal{P} = \{\pi_0, \pi_1, \dots, \pi_k\}$ such that (i) for all i, π_{i+1} is a refinement of π_i, (ii) $\pi_0 = \{[n]\}$ and (iii) $\pi_k = \{\{i\}\}_{i\in[n]}$ (singletons). Given such $\mathcal{P}$, each interval (set) in π_i is referred to as a **group**. The sets in π_i are called level-i groups for $\mathcal{P}$.
  A hierarchical partition $\mathcal{P}$ has a natural corresponding tree $\mathcal{T}$ where each node in $V(\mathcal{T})$ corresponds to a group. The root of $\mathcal{T}$ is $[n]$ (the group in π_0). Level-i of the tree contains nodes that have 1-to-1 correspondence with sets in π_i. Moreover, there is an edge connecting $S \in \pi_i$ to $S' \in \pi_{i+1}$ if $S' \subseteq S$.
- **Weight functions**: Given a canonical hierarchical partition $\mathcal{P}$, a $\mathcal{P}$-weight function is an assignment of positive real values to nodes in $V(\mathcal{T})$, i.e., $w : V(\mathcal{T}) \to \mathbb{R}_{>0}$.

We use the term **group access tree** to denote a hierarchical partition $\mathcal{P}$ (as well as its corresponding tree $\mathcal{T}$). See Figure 2 for illustration.

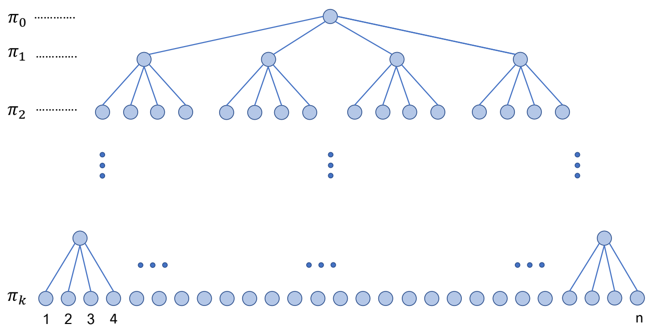

Figure 2 An illustration of group access tree.

▶ **Definition 11** (Group given a key). *Let $\mathcal{P}$ (or $\mathcal{T}$) be a group access tree and let $p \in [n]$ be a key. Then, for each j, we use $g_j(p)$ to denote the (unique) level-j group $S \in \pi_j$ in which p lies.*

▶ **Definition 12** (Level j groups of a group). *Given a level-$(j-1)$ group (interval) g, denote by $\mathsf{CG}(g)$ the set of children groups in level j that are contained in g, i.e., $\mathsf{CG}(g) = \{g' \in \pi_j : g' \subseteq g\}$. These are exactly the same as the groups that are children of node $g \in V(\mathcal{T})$.*

We are now ready to define the access cost in the group access tree. When accessing key $a \in [n]$ in the group access tree $\mathcal{T}$ and $\mathcal{P}$-weight w, denote by $\mathcal{T}(a)$ the path from root to a in the tree $\mathcal{T}$; in particular, this path contains $\mathcal{T}(a) = (g_0(a), g_1(a), \dots, g_k(a) = \{a\})$. The cost incurred on edge $e = (g_{j-1}(a), g_j(a))$ is

$$c_e(\mathcal{T}, w, a) = \log\left(\frac{W^j}{w(g_j(a))}\right)$$

where $W^j = \sum_{g' \in \mathsf{CG}(g_{j-1}(a))} w(g')$. The total access cost is $c(\mathcal{T}, w, a) = \sum_{e \in \mathcal{T}(a)} c_e(\mathcal{T}, w, a)$. Notice that this access cost is very similar to the access lemma cost. In fact, one can show that it generalizes the access lemma:

▶ **Observation 13.** *The access lemma corresponds to the cost $c(\mathcal{T}, w, a)$ when $\mathcal{T}$ is a star.*

In this way, our group access bound can be seen as an attempt to strengthen the standard access lemma by introducing hierarchical partitioning. As in the access lemma, the interesting application to BSTs happens when the changes of weights are "controllable" (e.g., in the working set bound). We introduce the concept of *locally bounded* weight families $\mathcal{W}$ to capture this property.

▶ **Definition 14.** *The weight family $\mathcal{W}$ is locally bounded if for all time t, for every group $g \notin \mathcal{T}(x_t)$, we have $w_{t+1}(g) \leq w_t(g)$. This means that the weight can increase only when $g \in \mathcal{T}(x_t)$.*

We are interested in the total access cost on a sequence $X = (x_1, \dots, x_m) \in [n]^m$ where the group access trees are allowed weight changes over time, that is, we are given a sequence of weight functions $\mathcal{W} = \{w_1, \dots, w_m\}$ where w_t denotes the weight function at time t.

The **group access bound** w.r.t. $(\mathcal{T}, X)$ is:

$$\mathsf{GAB}(\mathcal{T}, X) = \min_{\mathcal{W} \text{ locally bounded}} \sum_{t \in [m]} c(\mathcal{T}, w_t, x_t).$$

Note that the use of "minimum" in the definition of group access bound serves to select the weight family with the lowest weight among multiple locally bounded weight families.

Our main contribution is in showing that the group access bound is competitive to many strong bounds in the binary search tree model. We say that the group access bound is (α, β)-**competitive** to function $f : [n]^m \to \mathbb{R}$ if there exists a group access tree $\mathcal{T}$ such that $\mathsf{GAB}(\mathcal{T}, X) \leq \alpha f(X) + \beta|X|$.

▶ **Theorem 15.** *The group access bound is $(O(\sqrt{\log n}), O(1))$-competitive to* OPT.

The group access bound can also be used by allowing randomization in choosing the hierarchical partition $\mathcal{P}$. We say that the group access bound is randomized (α, β)-**competitive** to function $f : [n]^m \to \mathbb{R}$ if there is an efficiently computable distribution $\mathcal{D}$ that samples $\mathcal{T}$ such that

$$\mathbb{E}_{\mathcal{T} \sim \mathcal{D}}[\mathsf{GAB}(\mathcal{T}, X)] \leq \alpha f(X) + \beta|X|.$$

▶ **Theorem 16.** *The group access bound is (randomized) $(O(1), O(\log k \log \log n))$-competitive to the k-finger bound when the reference tree is an almost complete binary tree.*

▶ **Theorem 17.** *The group access bound is $(O(1), O(\log \log n))$-competitive to the unified bound.*

▶ **Theorem 18.** *The group access bound is (randomized) $(O(1), O(\log k \log \log n))$-competitive to the unified bound with a time window of size k.*

We say that a BST algorithm $\mathcal{A}$ **satisfies the group access bound** w.r.t. group access tree $\mathcal{T}$ if the cost of the algorithm is at most $O(\mathsf{GAB}(\mathcal{T}, X))$ for all input sequence X.

▶ **Proposition 19.** *If a BST algorithm satisfies the group access bound and the group access bound is (α, β)-competitive to function f, then the cost of the BST algorithm on any sequence X is at most $O(\alpha \cdot f(X) + \beta|X|)$.*

We will later show that a family of BST algorithms (named GGREEDY) satisfies the group access bound and possesses all the aforementioned properties.

4 $(O(\sqrt{\log n}), O(1))$-competitiveness

In this section, we prove Theorem 15. We will define the appropriate group access tree so that the group access bound is upper bounded by $(O(\sqrt{\log n}), O(1)) \cdot \mathsf{OPT}(X)$.

Group access tree

Define the partition $\mathcal{P}$ inductively as follows. Let $M = 2^{\sqrt{\log n}}$. First $\pi_0 = \{[n]\}$. Given π_i, we define π_{i+1} by, for each interval $S \in \pi_i$, partitioning S into $S_1, S_2, \ldots, S_M$ equal-sized intervals and adding them into π_{i+1}. This would give us a group access tree where each non-leaf node has M children and its height is at most $h = O(\sqrt{\log n})$.

Weight function

Given a sequence $X = (x_1, x_2, \ldots, x_m)$, we define a weight function $\mathcal{W}$ which is locally bounded and such that $c(\mathcal{T}, \mathcal{W}, X) = \sum_t c(\mathcal{T}, \mathcal{W}, x_t) \leq O(\sqrt{\log n}) \cdot (\mathsf{OPT}(X) + m)$. This will give us the desired result. Our weight function uses the notion of *last access*.

▶ **Definition 20.** *Consider time t and the group $g = g_{j-1}(x_t)$. Let $t' < t$ be the last time before t at which g is on the search path $\mathcal{T}(x_t)$. We say that a child $g_1 \in \mathsf{CG}(g)$ is **last accessed (child) group** of g at time t if the edge (g, g_1) is on search path $\mathcal{T}(x_{t'})$.*

Remark that each group can have at most one child in $\mathcal{T}$ that is the last access group. Now, we are ready to define the weight function:

$$w_t(g) = \begin{cases} M & \text{if } g \text{ is the last accessed group of its parent} \\ 1 & \text{otherwise} \end{cases}$$

It is easy to verify that this family of weight functions $\mathcal{W}$ is locally bounded (there is only one key whose weight can increase between time t and $(t+1)$).

▶ **Lemma 21.** *Consider edge $e = (g_{j-1}(x_t), g_j(x_t))$. We have*

$$c_e(\mathcal{T}, \mathcal{W}, x_t) = \begin{cases} O(\log M) & \textit{if } g_j(x_t) \textit{ is not the last accessed group of } g_{j-1}(x_t) \\ O(1) & \textit{otherwise} \end{cases}$$

Proof. By definition, $W^j = \sum_{g' \in \mathsf{CG}(g_{j-1}(x_t))} w_t(g') \leq 2M$ because there is only one group in $\mathsf{CG}(g_{j-1}(x_t))$ with weight M (the last accessed group) and there can be at most M groups with weight 1.

We analyze the cost in two cases: If $g = g_j(x_t)$ is the last accessed group, then we have $c_e(\mathcal{T}, \mathcal{W}, x_t) \leq \log(W^j / w_t(g)) \leq \log(2M/M) = O(1)$. Otherwise, if $g_j(x_t)$ is not the last accessed group, then we have $\log(2M) \leq O(\log M)$. ◀

Cost analysis

Consider the search path $\mathcal{T}(x_t)$. Denote by γ_t the number of groups on $\mathcal{T}(x_t)$ that are not the last accessed group of its parents. The cost $c(\mathcal{T}, w_t, x_t) = \sum_{e \in \mathcal{T}(x_t)} c_e(\mathcal{T}, w_t, x_t)$, and from the above lemma, the cost is at most $O(\log M) \cdot \gamma_t + O(h) \leq O(\sqrt{\log n})(\gamma_t + 1)$. Therefore, the total cost is $O(\sqrt{\log n}) \cdot (\sum_t \gamma_t + m)$. The sum of γ_t will be upper bounded by the Wilber bound.

The Wilber bound. Wilber [36] gave two lower bounds on the running time of any BST on a sequence X. These bounds are known as $\textsc{Wilber1}$ and $\textsc{Wilber2}$.[1] We now describe $\textsc{Wilber1}(X)$. Let R be a leaf-oriented (keys at the leaves) binary search tree, and for each $a \in [n]$, denote by $R(a)$ the search path in R of key a. When searching a sequence X in R, for each node $v \in V(R)$, the **preferred child** of node v at time t (denoted by $\textsc{Preferred-Child}_t(v)$) is the child of v on the last search path in R at time t. If node v is not on the search path $R(x_t)$, we know that the preferred child cannot change, i.e., $\textsc{Preferred-Child}_t(v) = \textsc{Preferred-Child}_{t-1}(v)$.

The Wilber bound with respect to R at time t and node v is:

$$\textsc{Wilber1}_R^t(v) = \begin{cases} 1 & \text{if } \textsc{Preferred-Child}_t(v) \neq \textsc{Preferred-Child}_{t-1}(v) \\ 0 & \text{otherwise} \end{cases}$$

The total Wilber bound of a sequence X is $\textsc{Wilber1}_R(X) = \sum_t \sum_v \textsc{Wilber1}_R^t(v)$. The Wilber bound is defined as the maximum, overall reference BST R, of $\textsc{Wilber1}_R(X)$.

▶ **Lemma 22.** *We have that $\sum_t \gamma_t \leq O(\textsc{Wilber1}(X))$*

Proof. It is sufficient to define a reference tree R that allows us to charge the cost of $\sum_t \gamma_t$ to $\textsc{Wilber1}_R(X)$. Notice that our group access tree $\mathcal{T}$ is not a binary search tree. However, it can be naturally extended into a binary search tree R as follows: We process the non-leaf nodes in $V(\mathcal{T})$ in a non-decreasing order of distance from the root. When a group $g \in V(\mathcal{T})$ is processed, we remove the edges from g to its children. Let T_g be an arbitrary BST rooted at g and leave $\mathsf{CG}(g)$; we add the tree T_g in place of the deleted edges (See Figure 3). After all vertices are processed, it is straightforward to see that the resulting tree is a (leaf-oriented) BST.

Now we claim that $\sum_t \gamma_t$ can be upper bounded by $\textsc{Wilber1}_R(X)$. Recall that γ_t is the number of groups on the search path $\mathcal{T}(x_t)$ that are not last accessed. Let $G_t \subseteq V(\mathcal{T}(x_t))$ be those groups on the search path that are not last accessed. For each such group $g_1 \in G_t$, let g be its parent, so we know that the last time g was on the search path, some other group $g_2 \in \mathsf{CG}(g)$ was instead chosen. Notice that the search paths $R(g_1)$ and $R(g_2)$ also visit g but branch away at some vertex b inside T_g (it could be that $b = g$). This would imply that $\textsc{Preferred-Child}_t(b) \neq \textsc{Preferred-Child}_{t-1}(b)$ and therefore $\textsc{Wilber1}_R^t(b) = 1$. This implies that $\gamma_t \leq \sum_v \textsc{Wilber1}_R^t(v)$ and hence the lemma. ◀

The lemma implies that the total group access bound is at most

$$O(\sqrt{\log n})(\textsc{Wilber1}(X) + m) \leq O(\sqrt{\log n}) \cdot (\mathsf{OPT}(X)).$$

See the full version of the paper for sections related to the *k-finger bound*, *Unified bound*, and *Unified bound with a time window.*

5 Ggreedy algorithm

This section will define a new BST algorithm named $\textsc{Ggreedy}$, which satisfies the *group access bound.* The $\textsc{Ggreedy}$ algorithm is very similar to $\textsc{Greedy}$. Therefore, we first describe the $\textsc{Greedy}$ algorithm and then provide some intuition about the $\textsc{Ggreedy}$ algorithm before describing it formally.

[1] They can also be derived using an elegant geometric language of [14].

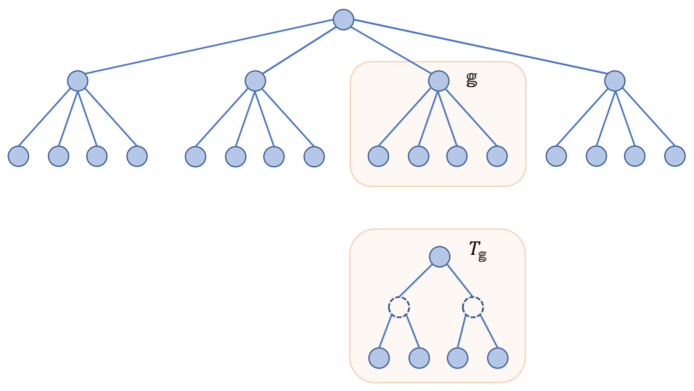

Figure 3 An example of g and T_g.

Given a search sequence $X = \{x_1, x_2, \ldots, x_m\}$, intuitively the Greedy algorithm works as follows:
At time t, the Greedy algorithm performs a horizontal line sweep of the points at $y = t$. By induction, we can assume that all the pairs of the points below the line $y = t$ are arborally satisfied. So, at time t, we find all the rectangles with one endpoint x_t and the other endpoint q, where q is a point below the sweep line. If the rectangle $\Box x_t q$ is not arborally satisfied, then add a point at the corner of the rectangle $\Box x_t q$ on the sweep line to make it arborally satisfied. See Algorithm 2 for the formal definition of Greedy (Figure 4b illustrates the execution of Greedy).

Algorithm 1 AddPoints(A,x_t).

```
1 foreach $\Box x_t q$ in $A$ do
2     Add a point at $(q.x, x_t.y)$
3 end
```

Algorithm 2 Processing Greedy at time t.

```
1 Let $A$ be the set of all the arborally
    unsatisfied rectangles with one endpoint as
    $x_t$ and another endpoint below the line
    $y = t$;
2 AddPoints($A, x_t$);
```

We will now try to provide some intuition (motivation) behind the Ggreedy algorithm. While analyzing the Greedy algorithm, we observed that if we can partition the keys into groups (containing consecutive keys) and treat these groups as keys, then we can apply the Greedy algorithm to this new set of keys. We can then recursively do this within each group, making the analysis easy.

As an example, we can divide the key space into three groups, say from $[1, \ldots, n/3]$, $[n/3 + 1, \ldots, 2n/3]$ and $[2n/3 + 1, \ldots, n]$ which we can represent as keys k_1, k_2 and k_3. We can now run the Greedy algorithm on this set of keys. When a key in the range $[n/3 + 1, \ldots, 2n/3]$ is searched at time t, we can assume that the key k_2 is searched and perform the Greedy algorithm accordingly on the keys k_1, k_2 and k_3. We can recursively do this procedure inside each group. To separate the groups and to ensure that the resultant point set is *arborally satisfied*, we add points at the boundaries of the groups.

One can observe that the recursive partition of the keys corresponds to a *group access tree*. An astute reader can see that Ggreedy is not a single algorithm but a family of BST algorithms, which depends on the group access tree.

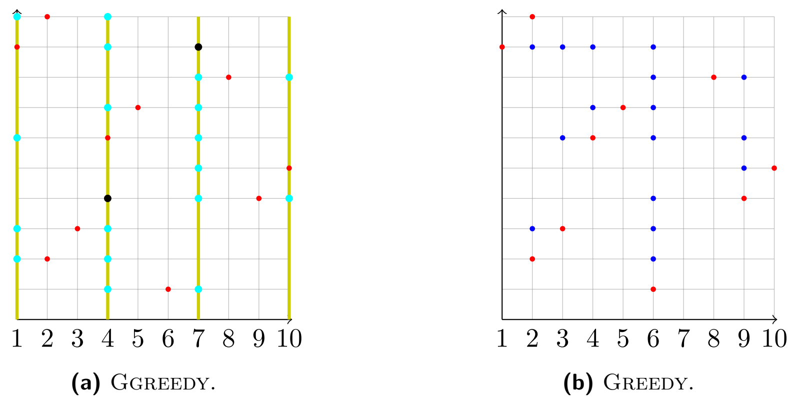

Figure 4 Ggreedy(on level 1) vs Greedy on a sequence $(6, 2, 3, 9, 10, 4, 5, 8, 1, 2)$. There are three groups $\{[1 \dots 4], [4 \dots 7], [7 \dots 10]\}$ in level 1, recursively we can define groups within each group. Red points are the searched keys. (a) Light blue points are added at the boundary of a group $g = g_1(x_t)$. Black points are added to make Ggreedy arborally satisfied with other groups in level 1 (b) Blue points are added to make Greedy arborally satisfied.

We are now ready to define the Ggreedy algorithm. Pick a group access tree $\mathcal{T}$. Let g be a group at any level in the group access tree. The group g contains a set of consecutive keys at the leaves of $\mathcal{T}$. The group g has two end keys, which we denote as the boundary keys of the group.

▶ **Definition 23** (Boundary of a group). *Let g be a group that contains consecutive keys $(a, a+1, a+2, \dots, b)$, then left$(g) = a$ is called the left boundary key of the group g. Similarly, right$(g) = b$ is called the right boundary key. Together, they are called the boundary keys of the group g.*

In Greedy, we access a search key x_t at time t, but in Ggreedy, when a key is searched at time t, we access a group per level of the group access tree $\mathcal{T}$. We define the access of a group as follows:

▶ **Definition 24** (Accessed group at time t in level j). *A group g is said to be accessed at time t in level j if $g = g_j(x_t)$.*

▶ **Definition 25** (Accessed boundary key at time t in level j). *The boundary keys of a group are accessed at time t in level j when the group g is accessed at time t in level j. We denote the boundary key access by adding points at the two boundaries of the group g at time t in level j.*

While accessing a key in Greedy at time t, the algorithm touches keys to form an arborally satisfied set. Similarly, in Ggreedy, when a group is accessed at time t, we might touch other groups. We define a touch group in Ggreedy as follows:

▶ **Definition 26** (Touched group at time t in level j). *A group g is said to be touched at time t in level j if one of the boundary keys of g is touched.*

▶ Remark 27. An accessed group is also a touch group at time t in level j, where both the boundary keys are touched.

Similar to the definition of *unsatisfied rectangle* for Greedy, we define an *unsatisfied group* in Ggreedy as follows:

▶ **Definition 28** (Unsatisfied group at time t in level j). *A group g_1 is said to be unsatisfied at time t in level j if one of the boundary keys of g_1 forms an unsatisfied rectangle with the boundary keys of $g = g_j(x_t)$ at time t. We denote the unsatisfied group by $\square g_1 g$.*

Let g_1 be a group at time t in level j, which is unsatisfied when the group g is accessed. We touch g_1 at time t in level j to make it an *arborally satisfied* group.

Let us now informally describe the GGREEDY algorithm. When a key x_t is accessed at time t, we access one group per level in the group access tree $\mathcal{T}$. While accessing the group $g = g_j(x_t)$ in level j, we find all the unsatisfied groups in level j which lie inside the group $g_{j-1}(x_t)$ and make them arborally satisfied. We recursively do this for level $j+1$ and so on. The GGREEDY algorithm can be viewed as if we are applying the GREEDY algorithm on groups at each level of the group access tree $\mathcal{T}$. Hence the name, GREEDY on groups or GGREEDY.

We now describe the GGREEDY algorithm formally (See Algorithm 4).

Algorithm 3 Touchgroups(A,g,j).

```
1 Touch $g$ in level $j$;
2 foreach $\square g_1 g$ in $A$ do
3     Touch $g_1$ in level $j$;
4 end
```

Algorithm 4 Processing of GGREEDY at time t.

```
1 foreach $j = 1$ to $k$ do
2     Let $A_j$ be the set of all arborally
        unsatisfied groups in level $j$ when
        accessing $g = g_j(x_t)$;
3     Touchgroups($A_j, g, j$);
4 end
```

In the above algorithm, at iteration j, we first add points at left($g_j(x_t)$) and right($g_j(x_t)$) (same as touching group $g_j(x_t)$). Then, we process all the arborally unsatisfied groups with one endpoint as $g_j(x_t)$ and the other endpoint inside the group $g_{j-1}(x_t)$ (See Figure 4a for the execution of GGREEDY on level 1).

▶ **Observation 29.** *Although the GGREEDY algorithm is very similar to GREEDY, it is not a candidate for dynamic optimality conjecture because of the inherent cost of the groups at each level in the group access tree $\mathcal{T}$.*

We will now try to bound the number of groups touched by GGREEDY in iteration j of Algorithm 4. Let us give it a special notation:

▶ **Definition 30.** *Let $\mathcal{T}_j(t)$ be the set of groups touched by GGREEDY in the j-th iteration of Algorithm 4 at time t.*

Touching a group is the same as touching the group's boundary key(s). Therefore, we consider the boundary keys of the group for the rest of this section to provide better-detailed proofs. We now show some essential properties of $\mathcal{T}_j(t)$.

▶ **Lemma 31.** *$\mathcal{T}_j(t)$ only contains points that are at the boundaries of level j groups that lie inside $g_{j-1}(x_t)$.*

For proof, refer to the full version.

An immediate corollary of the above lemma is:

▶ **Corollary 32.** *Let p be a point that lies strictly inside $g_{j-1}(x_t)$. If GGREEDY adds p at time t then x_t lies in $g_j(p)$.*

We will now show that GGREEDY outputs an arborally satisfied set. Let us assume that we are processing x_t. We can visualize GGREEDY as running GREEDY first on the level 1 group. This execution is the same as GREEDY. Then, we add points at the boundary of $g_1(x_t)$. This is the first step when we deviate from the GREEDY algorithm. Once we have added the boundary points, we again run GREEDY on level 2 groups, and the process continues. One may feel that touching the boundary keys may create some unsatisfied rectangles. But in the following lemma, we show that this is not the case.

▶ **Lemma 33.** *GGREEDY outputs an arborally satisfied set on any input representing X.*

The proof can be found in the full version of the paper.

In the next section, we generalize the *access lemma*, which have been proved for the Splay tree and GREEDY [33, 20]. Let us define a notation before we move to the next section.

▶ **Definition 34.** *Let $\hat{\mathcal{T}}_j(t)$ denote the amortized number of groups touched by GGREEDY in the j-th iteration of Algorithm 4 at time t.*

6 The Group Access Lemma

In this section, we introduce the *group access lemma*, which is a generalization of the *access lemma* and show that the GGREEDY algorithm satisfies it.

Consider the group access tree $\mathcal{T}$. When a BST algorithm $\mathcal{A}$ accesses x_t at time t, after the access, it adjusts itself to $\mathcal{A}'$. Let $g_1 = g_j(x_t)$ be the group that contains x_t. Let Φ_t^j be a potential function that depends on the state of the algorithm with respect to the groups at level j in $\mathcal{T}$. Define $\Phi_t = \sum_j \Phi_t^j$. Define the group access lemma as follows:

▶ **Definition 35** (Group Access Lemma). *A BST algorithm $\mathcal{A}$ satisfies the group access lemma if the amortized cost to access x_t at time t in level j is:*

$$\hat{\mathcal{T}}_j(t) \leq O\left(\log \frac{W^j}{w_{t-1}(g_1)}\right) + \Phi_{t-1}^j - \Phi_t^j$$

The amortized cost of the algorithm $\mathcal{A}$ at time t can be denoted as:

$$\hat{\mathcal{T}}(\mathcal{A}, x_t) = \sum_j \hat{\mathcal{T}}_j(t)$$

and the amortized cost of the algorithm $\mathcal{A}$ on the access sequence X can be defined as:

$$\hat{\mathcal{T}}(\mathcal{A}, X) = \sum_t \hat{\mathcal{T}}(\mathcal{A}, x_t).$$

With the definition of the group access lemma in hand, we will now show that GGREEDY satisfies the group access lemma.

Please refer to the full version of the paper for the proof that GGREEDY satisfies the group access lemma and the simulation theorem.

References

1 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.

2 Benjamin Aram Berendsohn, László Kozma, and Michal Opler. Optimization with pattern-avoiding input. *CoRR*, abs/2310.04236, 2023. arXiv:2310.04236.

3 Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An o (log log n)-competitive binary search tree with optimal worst-case access times. In *Scandinavian Workshop on Algorithm Theory*, pages 38–49. Springer, 2010.

4 Prosenjit Bose, Karim Douïeb, Vida Dujmović, and John Howat. Layered working-set trees. *Algorithmica*, 63(1-2):476–489, 2012.

5 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Greedy is an almost optimal deque. In *Workshop on Algorithms and Data Structures*, pages 152–165. Springer, 2015.

6 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Pattern-avoiding access in binary search trees. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 410–423. IEEE, 2015.

7 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Self-adjusting binary search trees: What makes them tick? In *Algorithms-ESA 2015*, pages 300–312. Springer, 2015.

8 Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. Multi-finger binary search trees. *arXiv preprint*, 2018. `arXiv:1809.01759`.

9 Parinya Chalermsook, Manoj Gupta, Wanchote Jiamjitrak, Nidia Obscura Acosta, Akash Pareek, and Sorrachai Yingchareonthawornchai. Improved pattern-avoidance bounds for greedy bsts via matrix decomposition. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 509–534. SIAM, 2023.

10 Parinya Chalermsook, Manoj Gupta, Wanchote Jiamjitrak, Nidia Obscura Acosta, Akash Pareek, and Sorrachai Yingchareonthawornchai. Improved pattern-avoidance bounds for greedy bsts via matrix decomposition. In *SODA*, pages 509–534. SIAM, 2023.

11 Parinya Chalermsook and Wanchote Po Jiamjitrak. New binary search tree bounds via geometric inversions. In *28th Annual European Symposium on Algorithms (ESA 2020)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

12 Parinya Chalermsook, Seth Pettie, and Sorrachai Yingchareonthawornchai. Sorting pattern-avoiding permutations via 0-1 matrices forbidding product patterns. In *SODA*, pages 133–149. SIAM, 2024.

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

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

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

16 Erik D Demaine, John Iacono, Stefan Langerman, and Özgür Özkan. Combining binary search trees. In *International Colloquium on Automata, Languages, and Programming*, pages 388–399. Springer, 2013.

17 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*, pages 194–205. Springer, 2009.

18 Jonathan Carlyle Derryberry. *Adaptive binary search trees.* PhD thesis, Carnegie Mellon University, 2009.

19 Amr Elmasry, Arash Farzan, and John Iacono. On the hierarchy of distribution-sensitive properties for data structures. *Acta informatica*, 50(4):289–295, 2013.

20 Kyle Fox. Upper bounds for maximally greedy binary search trees. In *Workshop on Algorithms and Data Structures*, pages 411–422, 2011.

21 George F Georgakopoulos. Chain-splay trees, or, how to achieve and prove loglogn-competitiveness by splaying. *Information Processing Letters*, 106(1):37–43, 2008.

22 Navin Goyal and Manoj Gupta. On dynamic optimality for binary search trees. *arXiv preprint*, 2011. `arXiv:1102.4523`.

23 Navin Goyal and Manoj Gupta. Better analysis of binary search tree on decomposable sequences. *Theoretical Computer Science*, 776:19–42, 2019.

24 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*, pages 516–522. Society for Industrial and Applied Mathematics, 2001.

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

26 John Iacono and Stefan Langerman. Weighted dynamic finger in binary search trees. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 672–691. SIAM, 2016.

27 Elias Koutsoupias and Christos H Papadimitriou. On the k-server conjecture. *Journal of the ACM (JACM)*, 42(5):971–983, 1995.

28 László Kozma and Thatchaphol Saranurak. Smooth heaps and a dual view of self-adjusting data structures. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 801–814, 2018.

29 James R Lee. Fusible hsts and the randomized k-server conjecture. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 438–449. IEEE, 2018.

30 J Ian Munro. On the competitiveness of linear search. In *European Symposium on Algorithms*, pages 338–345. Springer, 2000.

31 Daniel Sleator. Achieving the unified bound in the best model. Talk, 2011.

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

33 Robert Endre Tarjan. Sequential access in splay trees takes linear time. *Combinatorica*, 5(4):367–378, 1985.

34 Chengwen Chris Wang. *Multi-Splay Trees*. PhD thesis, Carnegie Mellon University, USA, 2006. AAI3253576.

35 Chengwen Chris Wang, Jonathan Derryberry, and Daniel Dominic Sleator. O (log log n)-competitive dynamic binary search trees. In *SODA*, volume 6, pages 374–383, 2006.

36 Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM journal on Computing*, 18(1):56–67, 1989.