Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS

Обобщенный подход для самоподстраивающихся деревьев поиска / Generic self-adjusting tree approach

Обучающийся / Student Сластин Александр Андреевич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан
Сластин Александр Андреевич
16.05.2023

(эл. подпись/ signature)

Сластин Александр Андреевич
(Фамилия И.О./ name and surname)

Руководитель ВКР/ Thesis supervisor

Документ подписан
Аксенов Виталий Евгеньевич
16.05.2023

(эл. подпись/ signature)

Аксенов Виталий Евгеньевич
(Фамилия И.О./ name and surname)

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ / OBJECTIVES FOR A GRADUATION THESIS

Обучающийся / Student Сластин Александр Андреевич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Обобщенный подход для самоподстраивающихся деревьев поиска / Generic self-adjusting tree approach
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Основные вопросы, подлежащие разработке / Key issues to be analyzed
Technical specification: It is required to develop a general approach for creating self-adjusting data structures which support the execution of queries typical for ordered sets: insert, delete, contains and additionally range-queries. The data structures must have an asymptotical complexity of queries that depends on the number of accesses to its elements, i.e., frequently accessed elements should be faster to access and show better performance on skewed workloads and our data structures should have better performance on skewed workloads in compare to their original counterparts. Moreover, it is required to compare the resulting self-adjusting data structures with their original versions, as well as with existing self-adjusting solutions, e.g. splay-tree.

Contents: The thesis should contain the description of an approach for creating self-adjusting data structures, theoretical proofs of asymptotics complexity. It should also contain the experimental comparison of self-adjusting versions with the original ones as well as with existing solutions.

Goal of the project: Develop of a generic approach to create self-adjusting version of ordinary data structures that provide interface of an ordered set.

Tasks of the project: to create generic self-adjusting tree approach and apply it for well-known classic search trees; to calculate the complexity of new data structures; to implement created data structures and to compare classic search trees and existing self-adjusting solutions, with the created ones.

Форма представления материалов ВКР / Format(s) of thesis materials:
software code, presentation, explanatory note

Дата выдачи задания / Assignment issued on: 01.10.2022

Срок представления готовой ВКР / Deadline for final edition of the thesis 24.05.2023

Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области фундаментальных исследований / Subject of fundamental research: да / yes
Тема в области прикладных исследований / Subject of applied research: нет / not

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/ Thesis supervisor

Документ подписан
Аксенов Виталий Евгеньевич
16.05.2023

(эл. подпись)

Аксенов Виталий Евгеньевич

Задание принял к исполнению/ Objectives assumed BY

Документ подписан
Сластин Александр Андреевич
16.05.2023

(эл. подпись)

Сластин Александр Андреевич

Руководитель ОП/ Head of educational program

Документ подписан
Станкевич Андрей Сергеевич
22.05.2023

(эл. подпись)

Станкевич Андрей Сергеевич

Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

АННОТАЦИЯ ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ SUMMARY OF A GRADUATION THESIS

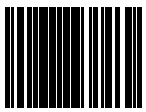
Обучающийся / Student Сластин Александр Андреевич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M34391
Направление подготовки/ Subject area 01.03.02 Прикладная математика и информатика
Образовательная программа / Educational program Информатика и программирование 2019
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Бакалавр
Тема ВКР/ Thesis topic Обобщенный подход для самоподстраивающихся деревьев поиска / Generic self-adjusting tree approach
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ DESCRIPTION OF THE GRADUATION THESIS

Цель исследования / Research goal
Develop a generic approach to create self-adjusting version of ordinary data structures that provide interface of an ordered set.
Задачи, решаемые в ВКР / Research tasks
to create generic self-adjusting tree approach and apply it for well-known classic search trees; to calculate the complexity of new data structures; to implement created data structures and to compare classic search trees and existing self-adjusting solutions, with the created ones.
Краткая характеристика полученных результатов / Short summary of results/findings
I developed a generic approach for creating self-adjusting data structures, applied it to the classic data structures and proved the complexities of the resulting structures. As the result, implemented self-adjusting versions showed better performance comparing with original ones on skewed workloads.

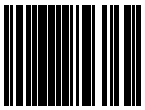
Обучающийся/Student

Документ подписан
Сластин Александр Андреевич



Сластин Александр

	16.05.2023		Андреевич
	(эл. подпись/ signature)		(Фамилия И.О./ name and surname)

Руководитель ВКР/ Thesis supervisor	Документ подписан Аксенов Виталий Евгеньевич 16.05.2023		Аксенов Виталий Евгеньевич
	(эл. подпись/ signature)		(Фамилия И.О./ name and surname)

CONTENTS

INTRODUCTION

Whenever it is necessary to use efficient sequential data structures, the choice usually falls on hash tables or balanced search trees [6] for their optimal worst-case guarantees. These data structures assume that every element has the same probability to be accessed, in other words, the data access distribution is uniform. However, in many real workloads, the frequency of accesses to different elements are not uniform. This fact is well-known, and is modelled in several industrial benchmarks, such as TPC-C [9], or YCSB [4], where the generated access distributions are heavy-tailed, e.g., following a Zipf distribution [13]. In that case, there are sequential self-adjusting data structures holding static-optimality property, which limits their efficiency to the efficiency of the best offline data structure for the given access sequence. However, in practice, e.g., as a database index, the software engineers usually use non-self-adjusting multiway trees such as B-Trees [2] or Interpolation Search Trees [8]. At the same time, they lack the properties that self-adjusting data structures hold, for example, static-optimality. The existing well-known self-adjusting data structures such as CBTree [5], Splay Tree [12], and its multiway variation, K-ary Splay Tree [3] are inferior in efficiency to the mentioned above non-self-adjusting multiway trees even on skewed workloads.

Our goal is to describe a generic approach to build self-adjusting data structures from the non-self-adjusting known structures. The main function that we need to use is the construction of our self-adjusting data structures from the list of pairs: (key, number of accesses). Designed data structures should provide significant performance benefits over an original, non-self-adjusting, tree designs as well as existing self-adjusting solutions on skewed workloads, and should have a static-optimality property for a large class of access distributions. Moreover, our self-adjusting data structures support range queries, allowing us to get a segment of existing keys, and effectively calculate some function on a segment of keys. In this work, using our approach, we create self-adjusting versions of the well-known data structures: Interpolation Search Tree (later referred to as IST) and B-Tree, as well as we propose a new data structure — Self-Adjusting Log Tree.

We defined several targets:

— Describe a generic approach for designing self-adjusting data structures, which exports an interface of an ordered set, apply it for well-known clas-

sic search trees and create a new data structure using it. This task includes a theoretical analysis of time complexities and memory.

— Extend designed data structures to support range queries for getting some segment of keys and for calculating some function on the segment of keys. This target includes a theoretical analysis of time complexities and memory.

— Experimentally compare the existing self-adjusting and classic, non-self-adjusting solutions with the new ones, presented in this work, on different workloads. This target includes an implementation of described data structures and a tool that executes the experiments and visualizes the results. Also, we need to analyze the obtained results.

The thesis is structured as follows:

— In Chapter 1, we give an introduction to the field of self-adjusting data structures. We define an ordered set, describe the static-optimality theorem, briefly describe data structures on top of which, we build our self-adjusting implementations and describe range queries supported by our created self-adjusting data structures. We finish the chapter with the description of related work and analysis of existing solutions.

— In Chapter 2, we describe a generic approach for creating self-adjusting data structures and consider some specific parameterizations of this approach. Then, we analyse time and memory complexities for building such trees and then for the operations' they support, including range queries. Next we prove the static-optimality property for our data structures. Finally, we briefly describe how to transform our self-adjusting trees into concurrent lock-free trees. We finish the second chapter with the comparison of the theoretical results.

— In the last Chapter 3, we present the results of experiments. In these experiments we compare the original search trees and existing self-adjusting solutions with the created self-adjusting data structures on different workloads and briefly describe them.

CHAPTER 1. THE INTRODUCTION INTO SELF-ADJUSTING ORDERED SETS

In this chapter, we provide the general information about the common interface and properties that we desire to obtain for our designed self-adjusting trees. Also, we describe the original versions of the data structures on top of which we create our solutions and consider existing self-adjusting trees, especially their rebuilding heuristic.

1.1. The Ordered Set

The ordered set is a data structure that provides the following operations:

a) `contains(key)` — returns an information if the `key` exists in the data structure or not.
b) `find(key)` — returns the `value` of an element with such a `key`, if it exists in the data structure, or `null`, otherwise.
c) `insert(key, value)` — adds the pair of a `key` and a `value` to the data structure if the `key` is not already present.
d) `delete(key)` — removes a pair with that `key` from the data structure if it exists.

Until we consider range queries, `find` operation is an obvious modifications of `contains` operation (for range queries we also need to propagate modifications from node's ancestors), so at the beginning we will focus only on `contains`, `insert` and `delete` operations.

We say that a `contains(key)` operation is *successful* (returns `true`) if the requested `key` is found in the data structure and was not marked as deleted; otherwise, the operation is *unsuccessful*. An `insert(key)` operation is *successful* (returns `true`) if the requested `key` was not present upon insertion; otherwise, it is *unsuccessful*. A `delete` operation is *successful* (returns `true`) if the requested `key` is found and was not marked as deleted, otherwise, the operation is *unsuccessful*. As suggested, in our implementations, the `delete` operation does not physically delete the object from the lists — instead, it just marks it as deleted. From here and then, without loss of generality, we assume that each object consists of a key-value pair. We thus use the terms *object* and *key* interchangeably.

1.2. The Static-Optimality

One of the important properties of self-adjusting data structures is the static-optimality. The Splay Tree [12] is one of the oldest data structures that satisfies it. For the Splay Tree the following theorem was proven in [12]:

Theorem 1. If every element is accessed at least once, then the total access time is $O\left(m + \sum\limits_{i=1}^{n} q(i) \times \log\left(\frac{m}{q(i)}\right)\right)$, where n is the number of elements in the data structure, $q(i)$ is the number of accesses made to the i-th element and m is the total number of accesses to all elements ($m = \sum\limits_{m=1}^{n} q(i)$).

Note, that if we are given the requests in advance, the best static data structure has exactly this complexity due to the information theory results. More precisely, the total access time for any fixed tree is $\Omega\left(m + \sum\limits_{i=1}^{n} q(i) \times \log\left(\frac{m}{q(i)}\right)\right)$ by a standard theorem from the information theory [1].

We want our data structures to hold this static-optimality property so they are as efficient as a static-optimal tree for the given access sequence. Formally, our task is to create data structures that:

a) provide ordered set operations;
b) hold static-optimality property for a wide class of access distributions;
c) show good worst-case time and memory performance.

1.3. The Splay Tree

Splay Tree [12] is the first self-adjusting data structure whose time complexity satisfies the static-optimality property. It is, as well as its multiway generalization, K-ary Splay Tree[3], use the *splay* restructuring heuristic for rebalancing: each operation moves a target node to to the root by performing a sequence of rotations along the original traversed path (see Figure 1).

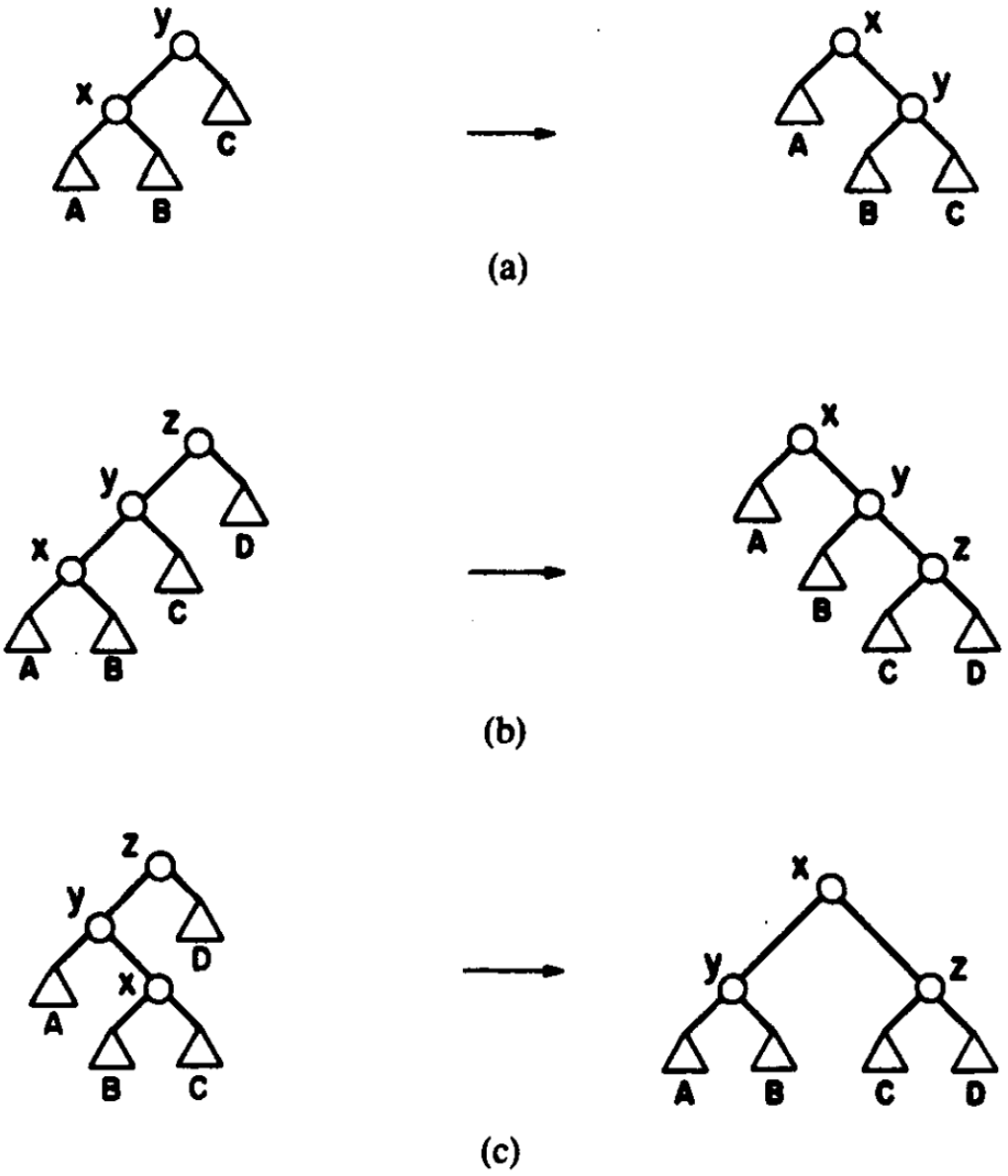


Figure 1 – Rotations of Splay-Tree [12]. The accessed node is x. Each case has a symmetric variant (not shown). (a) Zig: terminating single rotation. (b) Zig-zig: two single rotations. (c) Zig-zag: double rotation.

Now, let us discuss the main issue of the *splay* heuristic. First of all, it requires frequent manipulation of pointers along the traversed path to adjust the tree structure to the request frequencies. Secondary, when it is used in its original form for the binary search trees, it is not as efficient for external memory as B-Trees, because the number of accesses to the nodes is still high. Its generalization for multiway nodes, shown for K-ary Splay Tree, provides only the upper bounds of $O(\log_2 n)$ on the amortized number of node accesses per operation, so its also no better than B-Tree. Finally, the concurrent modification of such rebalancing approach is not

trivial and usually it makes threads to conflict on top of the tree, which makes the data structure less scalable.

Our designed self-adjusting data structures do not use pointers manipulation for rebalancing, therefore, they avoid the disadvantages mentioned above. In our work, we do not focus on analysing concurrent versions of the created trees, however, we describe a simple way to make them concurrent lock-free, using an approach from [10].

1.4. The B-Tree

The B-Tree is a balanced search tree whose nodes have the number of keys from B to $2 \cdot B$ and the number of children from $B + 1$ to $2 \cdot B + 1$, respectively, where B is a preselected constant.

All operations in B-tree search for a node with the specified key. To do that, they traverse down the tree, starting from the root node, and use the binary search in each node: 1) to stop traversal, if the requested key was found, or 2) to find the proper children node to look at.

Restructuring can be done only during update, `insert` or `delete`, operations:

a) If an `insert` ended in the leaf node and the requested key was not found, then we physically add it to the node. If after that the number of keys in the changed node exceeds the limit, i.e., it becomes $2 \cdot B + 1$, then, we move the middle key of the changed node to the parent node, split the changed node into two, link them to the moved key, and, then, repeat the same procedure with the parent node;

b) If the `delete` operation finds a node with the provided key and replaces it with either with the minimum key in its left subtree or the maximum key in its right subtree. After that if one of the node on the path lacks the required number of keys, i.e., becomes $B - 1$, then, we merge that node with its neighbour and proceed to the next node on the path.

This data structure is usually used as a database index due to better work with memory and $O(\log n)$ operations' time complexity.

1.5. The Interpolation Search Tree

The interpolation search tree was proposed in [8]. Its worst-case amortized bounds for all operations are $O(\log^2 n)$, but for a wide class of distributions called

`smooth`, its expected time of all operations is $O(\log \log n)$. Here, in our work, we use the same definition of smoothness, which was given in [8], and prove additional Lemma 22 for it.

1.6. The Range queries

The existing self-adjusting data structures are usually not supplemented with range queries. But even if they start to support them, these queries do not affect the structure of the tree. In our work, the created extended self-adjusting data structures support range queries that affect the overall structure of the tree, thereby preserving the properties of self-adjusting trees.

We require our data structures to support the following range-queries, denoting the key as x_i, and the corresponding value as y_i:

a) `get(a, b)` — returns an array of objects, whose keys belong to the passed range $[a, b]$ and are currently present in the tree: $[x_i < x_{i+1} < \ldots < x_{i+l}] \subseteq [a, b]$;

b) `calculate(a, b)` — returns the result of applying function $\odot$ to all values in the order of keys, which are currently present in the tree: $y_i \odot y_{i+1} \odot \ldots \odot y_{i+l}$ for which $[x_i < x_{i+1} < \ldots < x_{i+l}] \subseteq [a, b]$;

c) `update(a, b, c)` — applies to all values whose keys belong to the range $[a, b]$ the following action: $y_i' = y_i \star c$ if $x_i \in [a, b]$.

In the corresponding chapter, we explain how to change the newly designed self-adjusting data structures, so that they support range queries with different asymptotics bounds, depending on its parameters. Also, we explain some restrictions on $\odot$ and $\star$.

Conclusions on Chapter 1

There are a lot of data structures that implement the ordered set interface. But all of them have some drawbacks: some are not statically-optimal, others are not efficient in practice.

In this Chapter, we presented operations that our data structures should support, gave an overview of several data structures that we are going to use as a parameterizations of generic self-adjusting approach. Moreover, we described the existing self-adjusting solutions and their flaws.

CHAPTER 2. GENERIC SELF-ADJUSTING TREE APPROACH

In this chapter we introduce an approach to create self-adjusting trees and prove its operations' theoretical bounds on the time complexity and memory consumption. Finally, we describe a way to make them concurrent lock-free.

2.1. The Tree Structure

Generic Self-Adjusting Tree (GSAT) is a multiway tree, where the degree of the node depends on the number of requests to the underlying set. More precisely, the degree of the root is $O(\mathrm{D}(m))$, where D is the function used for *delimiting* elements into the corresponding subtree and m is the total number of requests to all the elements in the tree. Additionally, we have the function $\mathrm{S}(T, key)$ which is used to search for the subtree of the tree T that should store key. In *ideal* GSAT, subtrees of a tree have almost equal total number of requests to all their elements, i.e., $\Theta(\frac{m}{\mathrm{D}(m)+1})$, and hence a child of the root of an ideal GSAT has a degree $O(\mathrm{D}(\frac{m}{\mathrm{D}(m)+1}))$.

2.1.1. Formal definitions

Let a and b be integers, $a < b$. A Generic Self-Adjusting Tree (GSAT) with boundaries a and b for a set $X = \{x_1 < x_2 < \ldots < x_n\} \subseteq [a, b]$ of n elements with $ac_1, ac_2, \ldots, ac_n$ accesses made to respective elements, consists of:

a) An integer $m = \sum_{i=1}^{n} ac_i$ — the total number of accesses to the elements of the tree, i.e., the set X.

b) An array $REP[1 \ldots k]$ of representatives $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, where $i_1 < i_2 < \ldots < i_k$ and $REP[j] = x_{i_j}$. Furthermore, k does not exceed $\lceil \mathrm{D}(m) \rceil$.

c) An array $AC[1 \ldots k]$ of the number of accesses made to $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, respectively.

d) The subtrees of the root node are GSATs $T_1, T_2, \ldots, T_{k+1}$ for the subsets $X_1, X_2, \ldots, X_{k+1}$, where $X_1 = \{x_1, \ldots, x_{i_1-1}\}$, $X_j = \{x_{i_{j-1}+1}, \ldots, x_{i_j-1}\}$ for $2 \leqslant j \leqslant k$, $X_{k+1} = \{x_{i_k+1}, \ldots, x_n\}$. Furthermore, T_1 has boundaries a and x_{i_1}, T_j has boundaries x_{i_j-1} and x_{i_j} for $2 \leqslant j \leqslant k$, T_{k+1} has boundaries x_k and b.

e) A function $\mathrm{S}(T, key)$ that returns the index i such that either $REP[i]$ is equal to key or T_i may contain key, or `null` if there is no such T_i.

The array REP contains several keys from the set X. In ideal GSATs, we require these keys to be equally spaced in accordance with the number of accesses to the elements between them.

Definition 2. A GSAT for a set X, $|X| = n$, and a number of accesses $ac_1, ac_2, \ldots, ac_n$ to the keys $x_1, x_2, \ldots, x_n$ respectively, is *ideal* if: (1) for each $j > 0$, i_j is the first element to the right of i_{j-1} such that the number of accesses between $i_{j-1}+1$ and i_j, both inclusive, is at least $\left\lceil \frac{m}{\lceil \mathrm{D}(m) \rceil + 1} \right\rceil$ and (2) if the GSATs $T_1, T_2, \ldots, T_{k+1}$ are also *ideal*.

Let us denote $m(T_i)$ for the number of accesses made to all elements of a subtree T_i. In the ideal GSAT, $m(T_i)$ has no more than $\left\lfloor \frac{m}{\lceil \mathrm{D}(m) \rceil + 1} \right\rfloor$ accesses, since x_i has at least one access, and the root has no more than $\lceil \mathrm{D}(m) \rceil$ elements. The general structure of the GSAT is the following (see Figure 2):

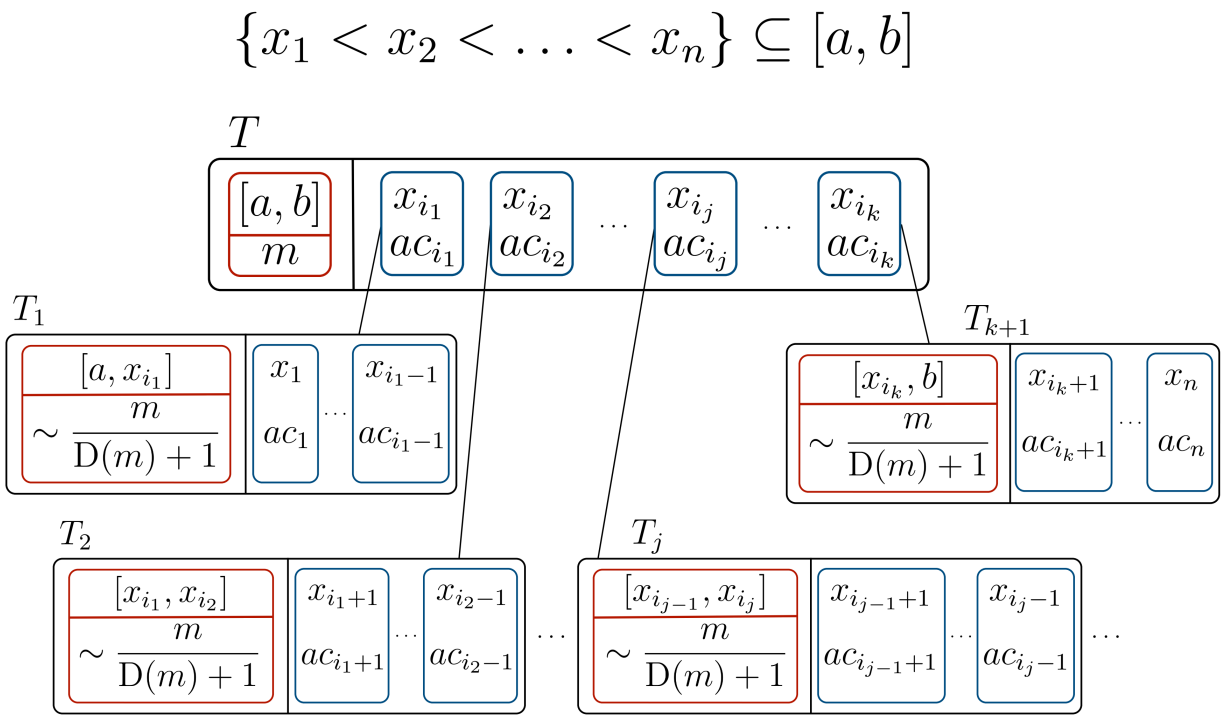


Figure 2 – General GSAT structure. The upper part of the red square corresponds to the segment on which the GSAT was built, and its lower part is equal to the total number of accesses to the elements of the tree.

However, the GSAT root node can have less than $\mathrm{D}(m)$ representatives. We will look at such an example in the next sub-chapter, in which we discuss the construction of a tree.

Not every function can be used as $\mathrm{D}(m)$ to build a GSAT. The function $\mathrm{D}(m)$ must be *sqrt-bounded*:

Definition 3. The function $f(m)$ is *sqrt-bounded* if:

a) $f(m) \geqslant 1$ if $m >= 2$;

b) $f(m) \leqslant \sqrt{m}$ except for the set of a finite measure on which $f(m)$ is bounded.

The first property (a) means that every GSAT node must have at least one element. The second one (b) is used to prove the construction time of ideal GSAT (Lemma 5) and actually there is no reason to have more than $\sqrt{m}$ representatives in the root — the subtrees would have less requests than the root, which looks strange. However, we relax this condition by allowing $\mathrm{D}(x) > \sqrt{x}$ (finite measure property) if:

a) $x \in \bigcup_{i=1}^{t} [a_i, b_i]$;

b) $t < +\infty$;

c) $\forall i = 1 \ldots t : |b_i - a_i| < +\infty$;

d) $\forall i = 1 \ldots t, \exists M_i \in [a_i, b_i], M_i < +\infty : M_i \geqslant x \in [a_i, b_i]$.

Definition 4. A GSAT is *correct* if its $\mathrm{D}(m)$ is sqrt-bounded.

From here, we assume that all considering GSATs are *correct*, or this fact is explicitly proved. So when we write *"An ideal GSAT..."*, we also mean that this GSAT is correct.

Depending on $\mathrm{D}(m)$ and $\mathrm{S}(T, key)$, the time complexities of operations as well as memory consumption may vary. For the further analysis, we consider three variations of these parameters:

a) $\mathrm{D}(m) = \sqrt{m}$, $\mathrm{S}(T, key)$ uses the interpolation search, based on the special array $ID[1 \ldots m^{\alpha}]$ with $\alpha \in [\frac{1}{2}, 1)$, and, then, the exponential search, together with binary search. We call this parameterization as Self-Adjusting Interpolation Tree (SAIT);

b) $\mathrm{D}(m) = \log_2(m)$, $\mathrm{S}(T, key)$ uses the pure binary search. We call this parameterization as Self-Adjusting Log Tree (SALT);

c) $\mathrm{D}(m) = B$, where B is preselected constant, $\mathrm{S}(node, key)$ uses only binary search. We call this parameterization as Self-Adjusting B-Tree (SABT).

SAIT in our scheme is similar to the standard Interpolation Search Tree [8]. However, it is based on the number of accesses rather then on the number of elements.

Obviously, SAIT is a *correct* (Definition 4) GSAT and we prove later that SALT (Lemma 7) and SABT (Lemma 8) are also correct GSATs.

2.1.2. Construction of Ideal Tree

We start with an algorithm on how to build the ideal GSAT given an ordered array of elements, augmented with the array of accesses. At the beginning, we count prefix sums of all accesses made to an ordered array. Then, we generate representatives in the root node on after another using binary search on the collected prefix sums, splitting the number of accesses almost evenly between the corresponding subtrees. The overall pseudo code is shown on listing 1:

Listing 1 – Building an ideal GSAT for n elements for the segment $[a, b]$.

```
1  fun BuildIdealTree(E[] elements, int[] ac, int n, float a,
2                     float b):
3    pac = new int[n + 1]
4    pac[0] = 0
5    for i = 1 .. n:
6      pac[i] = pac[i - 1] + ac[i]
7    return Build(elements, ac, pac, a, b, 0, n)
8
9  fun Build(E[] elements, int[] ac, int[] pac, float a, float b,
10           int lt, int rt):
11   if rt - lt ⩽ 0:
12     return null
13   m = pac[rt] - pac[lt]
14   node = new Node(m, a, b)
15   k = 0
16   for i = 1 .. ⌈D(m)⌉:
17     from, to = lt, rt
18     while to - from > 1:
19       m = ⌊(from+to)/2⌋
20       if pac[m] - pac[lt] < ⌈m/(⌈D(m)⌉+1)⌉:
21         from = m
22       else:
23         to = m
24     node.rep[i] = elements[to]
25     node.ac[i] = ac[to]
26     node.children[i] =
27       Build(elements, ac, pac, a, elements[to].key, lt, to - 1)
28     k = k + 1
29     a = elements[to].key
30     lt = to
31     if lt == rt:
32       break
33   node.children[k + 1] =
34     Build(elements, ac, pac, a, b, lt, rt)
35   return node
```

Each representative is searched using binary search (lines 16–22), initialized (lines 23–24) and its left subtree is build recursively (lines 25–26). Each subtree has less than $\left\lceil \frac{m}{\lceil \mathrm{D}(m) \rceil + 1} \right\rceil$ accesses in total and $m(T_{k+1}) \leqslant m - \lceil \mathrm{D}(m) \rceil \times \left\lceil \frac{m}{\lceil \mathrm{D}(m) \rceil + 1} \right\rceil \leqslant m - \frac{\lceil \mathrm{D}(m) \rceil \times m}{\lceil \mathrm{D}(m) \rceil + 1} = \frac{m}{\lceil \mathrm{D}(m) \rceil + 1} \leqslant \frac{m}{\mathrm{D}(m)+1}$, which means that every subtree would have no more than $\frac{m}{\mathrm{D}(m)+1}$ accesses in total. Also, a node could have less than $\mathrm{D}(m)$ representatives, which can be seen in the Figure 3, but the subtrees have at least the required total number of accesses:

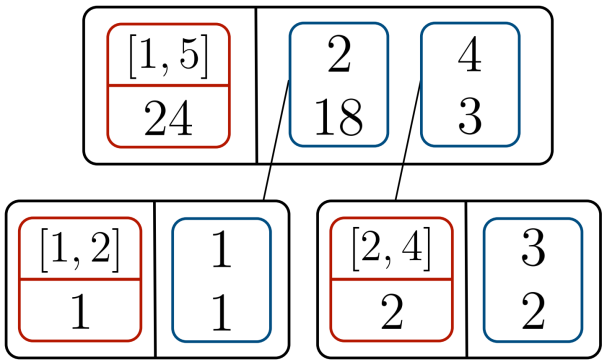


Figure 3 – Non-full GSAT. $\mathrm{D}(m) = \sqrt{m}$, $x = [1, 2, 3, 4] \subseteq [1, 5]$, $ac = [1, 18, 2, 3]$.

Additionally, for SAIT we will build an array ID and it can be done after the line 33, when we have already built all representatives for the current node, so the analysis for building some auxiliary data structure can be done separately from building representatives.

Let us prove the several properties of that algorithm and the ideal GSAT.

Theorem 5. An ideal GSAT for an ordered set and array of accesses, having size n, can be built in time $O(m)$ and requires $O(m)$ memory. Also, it has depth $O(\log m)$.

Proof. Let $Time(m)$ be the time required to build a GSAT, then

$$Time(m) \leqslant c \times \mathrm{D}(m) + \log_2(n) \times \mathrm{D}(m) + (\mathrm{D}(m) + 1) \times Time\left(\frac{m}{\mathrm{D}(m) + 1}\right) :$$

$c \times \mathrm{D}(m)$ can be spent for the node to build an additional data structure, e.g., segment tree for range queries; to build representatives array in the root via binary search we spend no more than $\log_2(n) \times \mathrm{D}(m)$, since the total number of elements is n; there are no more than $(\mathrm{D}(m)+1)$ subtrees T_i for which $m(T_i) \leqslant \frac{m}{\mathrm{D}(m)+1}$. Finally, $n \leqslant m$,

so $Time(m) \leqslant (c + \log_2(m)) \times \mathrm{D}(m) + (\mathrm{D}(m)+1) \times Time\Big(\frac{m}{\mathrm{D}(m)+1}\Big)$. We prove that this function is $O(m)$ in Lemma 6.

Let $Mem(m)$ be the required memory for storing GSAT, then using the same discussion as for the time,

$$Mem(m) = c \times \mathrm{D}(m) + (\mathrm{D}(m)+1) \times Mem\Big(\frac{m}{\mathrm{D}(m)+1}\Big) \leqslant Time(m) = O(m)$$

Lastly, let see $d(m)$ be the depth of the ideal GSAT. Then, $d(m) = 1 + d\Big(\frac{m}{\mathrm{D}(m)+1}\Big)$. By the sqrt-bounded property (a), $\mathrm{D}(m)$ is at least one, so, $\frac{m}{\mathrm{D}(m)+1} \leqslant \frac{m}{2}$, thus, GSAT has depth $O(\log m)$. $\square$

Lemma 6. If $T(m) = c \times \log_2(m) \times f(m) + (f(m)+1) \times T\Big(\frac{m}{f(m)+1}\Big)$ where $f(m)$ is *sqrt-bounded* function, then $T(m) = O(m)$ for $m \in \mathbb{N}$.

Proof. Let us restrict m to be at least 8. Consider $\widetilde{T}(m) = \frac{T(m)}{m}$. Then $m\widetilde{T}(m) = c \times \log_2(m) \times f(m) + (f(m)+1) \times \frac{m}{f(m)+1} \times \widetilde{T}\Big(\frac{m}{f(m)+1}\Big)$, so, $\widetilde{T}(m) = \frac{c \times \log_2(m) \times f(m)}{m} + \widetilde{T}\Big(\frac{m}{f(m)+1}\Big)$. For $m \geqslant 2$, we have $\frac{m}{f(m)+1} \leqslant \frac{m}{2}$ thus $\widetilde{T}(m) \leqslant \frac{c \times \log_2(m) \times f(m)}{m} + \widetilde{T}\Big(\frac{m}{2}\Big)$. Now, we can unwrap the recurrence explicitly:

$$\widetilde{T}(m) \leqslant \sum_{i=0}^{log_2(m)} \frac{c \times \log_2(\frac{m}{2^i}) \times f(\frac{m}{2^i}) \times 2^i}{m} = \frac{c}{m} \times \Big(\sum_{i=0}^{log_2(m)} f\Big(\frac{m}{2^i}\Big) \times \log_2\Big(\frac{m}{2^i}\Big) \times 2^i \Big).$$

Let M^* be the minimum value such that if $f(x) > \sqrt{x}$ then $M^* \geqslant x$. From the finite measure property in sqrt-bounded definition, we know that $M^* < +\infty$. Therefore,

$$\widetilde{T}(m) \leqslant \frac{c}{m} \times \Big(\sum_{i=0}^{log_2(m)} \Big(\sqrt{\frac{m}{2^i}} \times \log_2\Big(\frac{m}{2^i}\Big) \times 2^i \Big) + M^* \times \sum_{i=0}^{log_2(m)} \Big(\log_2\Big(\frac{m}{2^i}\Big) \times 2^i \Big) \Big).$$

Consider functions

$$P(m) = \sum_{i=0}^{log_2(m)} p(i), p(x) = \sqrt{\frac{m}{2^x}} \times \log_2\Big(\frac{m}{2^x}\Big) \times 2^x$$

and

$$Q(m) = \sum_{i=0}^{log_2(m)} q(i), q(x) = \log_2\Big(\frac{m}{2^x}\Big) \times 2^x.$$

If we show that $P(m) = O(m)$, then $Q(m) = O(m)$, because $p(x) \geqslant q(x)$ if $x \geqslant 0$, and we get our target that $\widetilde{T}(m) = O(1)$ since

$$\widetilde{T}(m) \leqslant \frac{c}{m} \times \Big(P(m) + M^* \times Q(m)\Big)$$

$$= \frac{c}{m} \times \Big(O(m) + M^* \times O(m)\Big) = c \times O(1) + c \times M^* \times O(1) = O(1),$$

because M^* is a constant. Therefore, we get $T(m) = m \cdot \widetilde{T}(m) = m \cdot O(1) = O(m)$ and the lemma is proved.

By assuming that $m = 2^r$, we can write $p(x) = (\sqrt{2})^{r+x} \times (r - x)$ since $\sqrt{\frac{m}{2^x}} \times \log_2\left(\frac{m}{2^x}\right) \times 2^x = 2^{\frac{r-x}{2}} \times (r - x) \times 2^x = 2^{\frac{r+x}{2}} \times (r - x)$. We want to find intervals where $p(x)$ is monotonous. For that we take a derivative

$$p'(x) = (\sqrt{2})^{r+x} \times \Big(\frac{\ln(2)}{2}(r - x) - 1\Big)$$

and compare it with 0. As we can see, $p'(x) = 0$ only if $x = r - 2\log_2 e = x_0$. Therefore, the function $p(x)$ has only one point where the monotonicity changes. On the right of x_0, by substituting $x = r$, we can see that the derivative is negative. At the same time, on the left of x_0, by substituting $x = 0$, we get a positive value. So, $p'(x)$ is first positive and then negative meaning that x_0 is the point of maximum.

From that we split x-s by x_0. If $x \geqslant r - 3$ we get $p(x) \leqslant p(x_0)$ since $2\log_2(e) < 3$, hence,

$$P(m) = \sum_{i=0}^{r} p(i) \leqslant \Big(\sum_{i=0}^{r-3} p(i)\Big) + 3 \times f(x_0) \leqslant \Big(\int_0^{r-3} p(x)\,\mathrm{d}x\Big) + 4 \times f(x_0)$$

For the last inequality we additionally add $f(x_0)$ so $\sum\limits_{i=0}^{r-3} p(i) \leqslant (\int\limits_0^{r-3} p(x)\,\mathrm{d}x) + f(x_0)$ and

$$\int p(x)\,\mathrm{d}x = \int \Big((\sqrt{2})^{r+x} \times (r-x)\Big)\,\mathrm{d}x = \frac{(\sqrt{2})^{r+x+2}}{\ln^2(2)} \times (r\ln(2) - x\ln(2) + 2) + C.$$

So, $P(m) \leqslant (\int\limits_0^{r-3} p(x)\,\mathrm{d}x) + 4 \times f(x_0) = \frac{(\sqrt{2})^{r+2}}{\ln^2(2)} \times ((\sqrt{2})^{r-3}(3\ln(2)+2) - r\ln(2) - 2)$ $+4 \times (\frac{2}{e} \times 2^r \log_2(e)) \leqslant \frac{2^r}{\ln^2(2)} \times (3\ln(2)+2) + \frac{8}{e} \times 2^r \log_2(e) = 2^r \times (\frac{3}{\ln(2)} + \frac{2}{\ln^2 2} + \frac{8\log_2(e)}{e})$ $= mc'$, where c' is a constant, thus, $P(m) = O(m)$. $\square$

Now, we are ready to prove that SALT and SABT are correct GSATs (Definition 4) by showing that $\mathrm{D}(m)$ is sqrt-bounded (Definition 3). While proving this fact, it is convenient to consider function

$$g(x) = \sqrt{x} - \mathrm{D}(x)$$

and then look at points where $g(x) < 0$ and x is the minimum point, i.e, $g'(x) = 0$, $g(x-\delta) > 0$, $g(x+\delta) < 0$. If for all such x it is true that $\mathrm{D}(x) < +\infty$, then for M^*, from the proof of Lemma 6, we can take the maximum value among such $\mathrm{D}(x)$, so function $\mathrm{D}(x)$ is sqrt-bounded.

Lemma 7. SALT is a *correct* GSAT.

Proof. For SALT we get $g(x) = \sqrt{x} - \log_2(x)$, $g'(x) = \frac{\sqrt{x}\ln(2)-2}{x\ln 4}$. $g'(x)$ has a single extremum point, $x_0 = \frac{4}{\ln^2 2}$, which is the minimum point. $g(x_0) \approx -0.172143 < 0$ and $\log_2(x_0) < +\infty$. $\square$

Lemma 8. SABT is a *correct* GSAT.

Proof. For SABT we get $g(x) = \sqrt{x} - B$. $g'(x) = \frac{1}{2\sqrt{x}}$. If $x > 0$ we get $g'(x) > 0$, so $g(x)$ is monotonically increasing function. We see that $g(x) \leqslant 0$ if $x \in [0, B^2]$ and $B < +\infty$, so SABT is a correct GSAT. $\square$

2.1.3. Parameterization Analysis

The Theorem 5 works for any GSAT regardless of its parameterization. However, by analyzing each parameterization separately, we can improve asymptotics bounds on the construction time, the memory consumption, and the depth of the tree.

Before proceeding to the proofs, we denote by $d(m)$ the depth of GSAT with m accesses to all its elements.

Theorem 9. An ideal SAIT requires $O(m)$ time to build, $O(m^{\frac{\alpha}{2}} \times n + m^{\alpha})$ memory, and has depth $O(\log\log m)$.

Proof. For SAIT we can build representatives for $O(m)$ in accordance with Theorem 5. However, we need to analyse building time for an ID array separately. Let us denote $P(m)$ time to build an ID array, then we have $P(m) = O(m^{\alpha}) + (\sqrt{m}+1) \times P(\frac{m}{\sqrt{m}+1})$. Let $\widetilde{P}(m) = \frac{P(m)}{m}$, so $m \cdot \widetilde{P}(m) = P(m)$

and $\widetilde{P}(m) = O(m^{\alpha-1}) + \widetilde{P}(\frac{m}{\sqrt{m}+1}) \leqslant O(m^{\alpha-1}) + \widetilde{P}(\sqrt{m})$, thus, if $m = 2^{2^r}$:

$$\widetilde{P}(m) = O\Big(\sum_{i=0}^{r}(m^{\frac{1}{2^i}})^{\alpha-1}\Big) = O\Big(\sum_{i=0}^{r}(2^{2^i})^{\alpha-1}\Big) = O(1),$$

since $\alpha < 1$. Therefore, $P(m) = O(m)$, which gives us $O(m)$ time to build an ideal SAIT.

Now, let us count the memory consumption. The root node of SAIT requires $O(m^{\alpha})$ space to store ID array. Also, the total number of nodes in SAIT does not exceed n, because tree is internal and, so each node has at least one element inside. Subtrees of the root node satisfies $m(T_i) \leqslant \sqrt{m}$. Thus, ID for each of these trees has size no more than $m^{\frac{\alpha}{2}}$, and, therefore, the memory required for all of the nodes, except for the root, does not exceed $O(m^{\frac{\alpha}{2}} \times n)$. After summing everything up, we get that the total memory does not exceed $O(m^{\frac{\alpha}{2}} \times n + m^{\alpha})$.

For $d(m)$ we get $d(m) \leqslant 1 + d(\sqrt{m})$, so $d(m) = O(\log\log m)$. $\square$

Not every GSAT needs $O(m)$ time to build and needs $O(m)$ memory. If each GSAT node stores only $O(k)$, where k is the degree of the current node (thus does not exceed $\lceil \mathrm{D}(m) \rceil$), and such data structure can be built in $O(k)$ time, we immediately get $O(n \log n)$ bound for the construction time, since there are only n elements in the tree, each is selected as a representative exactly once during the binary search for $O(\log n)$. Also, we have $O(n)$ bound for memory, because GSAT is an internal tree. However, the depth should be considered separately to get a more precise bound.

Theorem 10. An ideal SALT requires $O(\min(n \log n, m))$ time to build, uses $O(n)$ memory, and has depth $O(\frac{\log m}{\log\log m})$.

Proof. SALT stores only elements and does not store additional data structure, so, we immediately get the required bounds for the construction time and the memory consumption.

For SALT depth we have:

$$d(m) \leqslant 1 + d\Big(\frac{m}{\log_2 m}\Big) \leqslant 2 + d\Big(\frac{m}{\log_2 m \times (\log_2 m - \log_2\log_2 m)}\Big).$$

Let us find such y that $\log_2 m - \log_2\log_2 m \geqslant y > 0$. Rewriting inequality through the exponentiation $\frac{m}{\log_2 m} \geqslant 2^y$ and assuming $\frac{m}{\log_2 m} \geqslant \sqrt{m} = 2^y$, we have $\log_2 m - \log_2\log_2 m \geqslant y = \frac{1}{2} \times \log_2 m$, thus, by applying the inequality one addi-

tional time,

$$d\Big(\frac{m}{\log_2 m \times (\log_2 m - \log_2 \log_2 m)}\Big) \leqslant d\Big(\frac{m \times 2}{\log_2 m \times \log_2 m}\Big)$$

$$\leqslant 1 + d\Big(\frac{m \times 2}{\log_2^2 m \times (\log m + 1 - 2\log_2 \log_2 m)}\Big).$$

Consider the function from the denominator above:

$$h_1(x) = \log m + x - (x+1)\log_2 \log_2 m,$$

participating in such inequality:

$$2^{h_1(x)} = \frac{m \cdot 2^x}{(\log_2 m)^{(x+1)}} \geqslant \sqrt{m}.$$

Until the inequality holds, we have $h_1(x) \geqslant \frac{1}{2} \times \log_2 m$ and

$$d\Big(\frac{m \times 2}{\log_2^2 m \times (\log m + 1 - 2\log_2 \log_2 m)}\Big) = d\Big(\frac{m \times 2}{\log_2^2 m \times h_1(1)}\Big) \leqslant d\Big(\frac{m \times 2^2}{(\log_2 m)^3}\Big)$$

$$= d\Big(2^{h_1(2)}\Big) \leqslant 1 + d\Big(\frac{m \times 2^2}{(\log_2 m)^3 \times (\log_2 m - 3 + 4\log_2 \log_2 m)}\Big) \leqslant 1 + d\Big(\frac{m \times 2^3}{(\log_2 m)^4}\Big)$$

$$= 1 + d\Big(2^{h_1(3)}\Big).$$

Thus, the depth can be estimated as:

$$d(m) \leqslant 1 + t + d\Big(2^{h_1(t)}\Big),$$

where t is the first value with

$$2^{h_1(t)} = \frac{m2^t}{(\log_2 m)^{t+1}} \leqslant \sqrt{m} \implies \log_2 m + t \leqslant \frac{1}{2} \times \log_2 m + (t+1) \times \log_2 \log_2 m$$

$$\implies t \geqslant \frac{\frac{1}{2} \times \log_2 m - \log_2 \log_2 m}{\log_2 \log_2 m - 1} \geqslant \frac{\frac{1}{2} \times \log_2 m}{\log_2 \log_2 m - 1}.$$

So,

$$d(m) \leqslant \Big(1 + \frac{\frac{1}{2} \times \log_2 m}{\log_2 \log_2 m - 1}\Big) + d(\sqrt{m}).$$

Next, we repeat the similar analysis but now for $d(\sqrt{m})$ instead of $d(m)$:

$$d(\sqrt{m}) \leqslant 1 + d\Big(\frac{2 \times \sqrt{m}}{\log_2 m}\Big) \leqslant 2 + d\Big(\frac{2 \times \sqrt{m}}{\log_2 m \times (1 + \frac{1}{2} \times \log_2 m - \log_2 \log_2 m)}\Big).$$

Let y be the value such that $(1+\frac{1}{2}\times\log_2 m-\log_2\log_2 m) \geqslant y > 0 \implies \frac{2\sqrt{m}}{log_2 m} \geqslant 2^y$. If $\frac{2\sqrt{m}}{log_2 m} \geqslant m^{\frac{1}{4}} = 2^y$ holds, we have $y = \frac{1}{4} \times \log_2 m$, so

$$d\Big(\frac{2 \times \sqrt{m}}{\log_2 m \times (1 + \frac{1}{2} \times \log_2 m - \log_2 \log_2 m)}\Big) \leqslant d\Big(\frac{2 \times \sqrt{m} \times 4}{(\log_2 m)^2}\Big)$$

$$\leqslant 1 + d\Big(\frac{2 \times \sqrt{m} \times 4}{(\log_2 m)^2 \times (1 + \frac{1}{2} \times \log_2 m + 2 - 2 \times \log_2 \log_2 m)}\Big).$$

Consider the function from the denominator above

$$h_2(x) = 1 + \frac{1}{2} \times \log_2 m + 2x - (x+1) \log_2 \log_2 m.$$

We have $h_2(x) \geqslant \frac{1}{4} \times \log_2 m$ until the inequality holds:

$$2^{h_2(x)} = \frac{2\sqrt{m} \times 2^{2x}}{(\log_2 m)^{x+1}} \geqslant m^{\frac{1}{4}},$$

therefore

$$d\Big(\frac{2 \times \sqrt{m}}{(\log_2 m)^2 \times (1 + \frac{1}{2} \times \log_2 m + 2 - 2 \times \log_2 \log_2 m)}\Big) = d\Big(\frac{2 \times \sqrt{m} \times 4}{(\log_2 m)^2 \times h_2(1)}\Big)$$

$$\leqslant d\Big(\frac{2 \times \sqrt{m} \times 4^2}{(\log_2 m)^3}\Big) = d\Big(2^{h_2(2)}\Big)$$

$$\leqslant 1 + d\Big(\frac{2 \times \sqrt{m} \times 4^2}{(\log_2 m)^3 \times (1 + \frac{1}{2} \times \log_2 m + 4 - 3 \times \log_2 \log_2 m)}\Big)$$

$$= 1 + d\Big(\frac{2 \times \sqrt{m} \times 4^2}{(\log_2 m)^3 \times h_2(2)}\Big) \leqslant 1 + d\Big(\frac{2 \times \sqrt{m} \times 4^3}{(\log_2 m)^4}\Big) = 1 + d\Big(2^{h_2(3)}\Big).$$

So,

$$d(\sqrt{m}) \leqslant 1 + t + d\Big(2^{h_2(t)}\Big),$$

where t is the first value that

$$\frac{2\sqrt{m}\times 2^{2t}}{(\log_2 m)^{t+1}} \leqslant m^{\frac{1}{4}} \implies t \geqslant \frac{1+\frac{1}{4}\times\log_2 m-\log_2\log_2 m}{\log_2\log_2 m-2} \geqslant \frac{1+\frac{1}{4}\times\log_2 m}{\log_2\log_2 m-2},$$

hence getting new estimate for the depth:

$$d(m) \leqslant \Big(1+\frac{\frac{1}{2}\times\log_2 m}{\log_2\log_2 m-1}\Big)+\Big(1+\frac{1+\frac{1}{4}\times\log_2 m}{\log_2\log_2 m-2}\Big)+d(m^{\frac{1}{4}}).$$

With this approach, we performed two iterations moving from $d(m)$ to $d(m^{\frac{1}{4}})$. Now, let us consider k-th iteration:

$$d(m^{\frac{1}{2^k}}) \leqslant 1+d\Big(\frac{2^k\times m^{\frac{1}{2^k}}}{\log_2 m}\Big) \leqslant 2+d\Big(\frac{2^k\times m^{\frac{1}{2^k}}}{\log_2 m\times(k+\frac{1}{2^k}\times\log_2 m-\log_2\log_2 m))}\Big).$$

Consider the function

$$h_{2^k}(x) = k+\frac{1}{2^k}\times\log m+(k+1)x-(x+1)\log_2\log_2 m,$$

until the inequality holds

$$2^{h_{2^k}(x)} = \frac{2^k\times m^{\frac{1}{2^k}}\times 2^{(k+1)x}}{(\log_2 m)^{x+1}} \geqslant m^{\frac{1}{2^{k+1}}} \implies h_{2^k}(x) \geqslant \frac{1}{2^{k+1}}\times\log_2 m,$$

so,

$$d(m^{\frac{1}{2^k}}) \leqslant 1+t+d\Big(\frac{2^k\times m^{\frac{1}{2^k}}\times 2^{t(k+1)}}{(\log_2 m)^{t+1}}\Big),$$

where t is the first value that

$$\frac{2^k\times m^{\frac{1}{2^k}}\times 2^{t(k+1)}}{(\log_2 m)^{t+1}} \leqslant m^{\frac{1}{2^{k+1}}}$$

$$\implies t \geqslant \frac{k+\frac{1}{2^{k+1}}\times\log_2 m-\log_2\log_2 m}{\log_2\log_2 m-(k+1)} \geqslant \frac{k+\frac{1}{2^{k+1}}\times\log_2 m}{\log_2\log_2 m-(k+1)},$$

which allows us to write down the recurrence explicitly, assuming that $m=2^{2^r}$:

$$d(m) \leqslant \sum_{i=0}^{\log_2\log_2 m-1}\Big(1+\frac{\frac{1}{2^{i+1}}\times\log_2 m}{\log_2\log_2 m-(i+1)}\Big) = \sum_{i=0}^{r-1}\Big(1+\frac{2^{r-1-i}}{r-1-i}\Big) \leqslant r+\sum_{i=0}^{r-1}\frac{2^i}{i}.$$

Then, using inequality $\frac{2^i}{i} \leqslant \frac{1}{2} \times \frac{2^{i+1}}{i+1}$ and supposing that $r \geqslant 2$ we get:

$$\sum_{i=0}^{r-1} \frac{2^i}{i} \leqslant \frac{2^{r-1}}{r-1} \times \sum_{i=0}^{r-1} \frac{1}{2^i} \leqslant \frac{2^r}{r-1} \leqslant \frac{2^{r+1}}{r},$$

so the depth can be estimated as:

$$d(m) \leqslant \log_2 \log_2 m + 2\Big(\frac{\log_2 m}{\log_2 \log_2 m}\Big) = O\Big(\frac{\log_2 m}{\log_2 \log_2 m}\Big). \quad \square$$

Theorem 11. An ideal SABT has $O(\min(n \log n, m))$ the construction time, has $O(n)$ memory, and has depth $O(\log_B(m))$.

Proof. For same reasons as for SALT (Theorem 11) we get $O(\min(n \log n, m))$ time to build and $O(n)$ memory. For $d(m)$ we get $d(m) \leqslant 1 + d(\frac{m}{B})$, so $d(m) = O(\log_B(m))$. $\square$

This concludes the analysis of the GSAT construction and we move on to operations.

2.2. Operations

2.2.1. Update operations

`Insert`, `delete` and `contains` operations are implemented in the same way as in [8], using an approach based on the counters to amortize the tree rebuilding. Except, now we use im and m instead of $isize$ and $size$, since GSAT builds subtrees based on the number of requests, not on the size of the set.

Now, let us discuss these operations in more detail. We associate with each node a counter which is initially equal to zero. Going down the tree in the search of the requested key, we increment counters of each node on the traversed path, until the key was found or there is no such key. Then, it finds the node on the traversed path with the minimum depth that *overflows*, i.e., whose counter exceeds $\frac{im}{4}$. The algorithm rebuilds the whole subtree of the overflowed node. im stands for the number of total requests to the tree after the last build, so, im changes only during the rebuilding.

Insertion links the key to the parent node, if the tree has not already contained it.

Deletions are performed by marking an element as deleted. The element is not physically removed — it is removed when some subtree containing it is rebuilt.

Let's consider that the subtree T' has the elements $X' = \{x_{i_1}, x_{i_2}, \ldots, x_{i_s}\}$ and there is a subset of marked for deletion elements $Marked \subseteq X'$.

If $T'' = \text{Rebuild}(T')$, then T'' contains the elements $X' \setminus Marked$ and the total number of requests is $\text{Requests}(T'') = \sum\limits_{x_i \in X'} ac_i - \sum\limits_{y_j \in Marked} ac_j$.

The pseudo code for `contains` operation (listing 2), `insert` operation (listing 3) and `delete` operation (listing 4) is listed below.

Listing 2 – Contains operation

```
1  fun Contains(Node T, E key):
2    T_R = null
3    result = false
4    while T ≠ null:
5      T.c = T.c + 1
6      if T.c > T.im/4 and T_R == null:
7          T_R = T
8      i = S(T, key)
9      if T.rep[i] == key:
10       if not T.marked[i]:
11         result = true
12       T.ac[i] = T.ac[i] + 1
13       break
14     T = T.children[i]
15   if T_R ≠ null:
16     T_R = RebuildTree(T_R)
17   return result
```

Rebuilding replaces the tree T by an ideal GSAT for the set of unmarked keys stored in T. After it, we have $u.c = 0$ and $u.im = u.m$ for each node in T. The pseudo code for this operation is listed below (listing 5).

Now, let us consider, how the GSAT structure has changed after introducing these operations.

Lemma 12. The worst-case depth of GSAT for the set of n elements, after executing `insert`, `delete`, `contains` operations, is $O(\log m)$.

Proof. Consider a node v and its parent w. Then for $m(v)$ (the current number of accesses to the subtree with the root v) we have: $m(v) \leqslant \frac{im(w)}{4} + \frac{im(w)}{\text{D}(m)+1}$, since no more than $\frac{im(w)}{4}$ accesses could occurred in T_v, otherwise, T_w would have been rebuilt. Also $\frac{im(w)}{\text{D}(m)+1} \leqslant \frac{im(w)}{2}$, so $m(v) \leqslant \frac{im(w)}{4} + \frac{im(w)}{2} = \frac{3}{4} \times im(w)$, thus, the depth of a GSAT is $O(\log im(root))$. Since $m = m(root) \geqslant im(root) - C(root) \geqslant \frac{im(root)}{2}$, we conclude that the depth is $O(\log m)$. $\square$

Listing 3 – Insert operation

```
1  fun Insert(Node T, E key):
2    T_R = null
3    P, j = null, null
4    while T ≠ null:
5      T.c = T.c + 1
6      if T.c > T.im/4 and T_R == null:
7          T_R = T
8      i = S(T, key)
9      if T.rep[i] == key:
10       if T.marked[i]:
11         T.marked[i] = false
12       T.ac[i] = T.ac[i] + 1
13       break
14     P, j = T, i
15     T = T.children[i]
16   if T == null:
17     P[j] = new Node(key, ac=1)
18   if T_R ≠ null:
19     T_R = RebuildTree(T_R)
```

Listing 4 – Delete operation

```
1  fun Delete(Node T, E key):
2    T_R = null
3    while T ≠ null:
4      T.c = T.c + 1
5      if T.c > T.im/4 and T_R == null:
6          T_R = T
7      i = S(T, key)
8      if T.rep[i] == key:
9        T.marked[i] = true
10       T.ac[i] = T.ac[i] + 1
11       break
12     T = T.children[i]
13   if T_R ≠ null:
14     T_R = RebuildTree(T_R)
```

Listing 5 – Rebuild operation

```
1  fun RebuildTree(Node T):
2    T' = BuildIdealTree(Unmarked(T))
3    for v ∈ T':
4      v.c = 0
5      v.im = v.m
6    return T'
```

Lemma 13. The amortized cost of `insert`, `delete` or `contains` operations (not counting the time for the preceding search) in GSAT is $O(\log m)$, that is, the total cost of the first m insertions, deletions and contains in GSAT is $O(m \times \log m)$.

Proof. We use the following accounting scheme to amortize the rebuild over all operations: every operation puts one token on each node on the path to the target element. By Lemma 12 each operation adds no more than $O(\log m)$ tokens. Let us denote $C(v)$ as the number of tokens in node v (the counter value). Rebuilding T_v in GSAT costs $O(m(T_v))$. From properties of counter and rebuilding we get $m(T_v) \leqslant \frac{5}{4} \times im(T_v)$ and $\frac{im(T_v)}{4} \leqslant C(v)$, thus, we have enough tokens to repay for the rebuilding of T_v. □

Now let us consider, how the memory bounds for GSAT parameterizations change.

Theorem 14. After `insert`, `delete` and `contains` operations, the SAIT for the set of n elements consumes $O(m^\alpha \times n)$ space.

Proof. If after any operation above no subtrees were rebuilt, then we spend $O(1)$ in case of an `insert` operation, since new node consumes only a constant memory. Otherwise, some subtree T' was rebuilt. After rebuilding T', memory has changed only for T' and its subtrees. If T' had n' elements, after rebuilding it can not spend more than $O(m^\alpha \times n')$, since for each element we can not spend more memory than for ID array, whose maximum size is $O(m^\alpha)$, thus, we get the required bound.

Theorem 15. After `insert`, `delete` and `contains` operations, the SALT and SABT for the set of n elements consumes $O(n)$ space.

Proof. Both of the trees are internal, thus, if after an operation no tree were rebuilt, we spend $O(1)$ in case of `insert` operation, and if some subtree T' with size n' was rebuilt, it consumes $O(n')$ space. □

As we can see, the worst-case analysis does not give good bounds for the depth of GSAT, however, it is worth considering this issue in the expected sense, which we will do in the next sub-chapter.

2.2.2. Expected Time Analysis

We start this section with the expected time analysis to improve worst-case bounds, if keys are requested in the accordance with some distribution.

Let μ be the probability density function on reals. Let F_n be a random file of size n which is generated by drawing independently n reals according to density μ.

Definition 16. A $\mu[a,b]$-random GSAT is a GSAT with boundaries $[a,b]$ with total number of accesses m to all its elements, generated by the following actions provided that μ is a density function with finite support $[a,b]$:

a) Take a random file $F_{n'}$ and build an ideal GSAT from its content, treating the number of occurrences of the element in the file as the number of accesses to it.

b) Perform a sequence of i μ-random insertions, d random deletions and c μ-random contains $Op_1, \ldots, Op_{i+c+d}$ so the number of accesses to non-deleted elements is m. An insertion is μ-random if it inserts a random real drawn according to density μ into the tree. A deletion is random if it deletes a random element from the tree and each element in the tree has the same probability to deleted. A contains operation is μ-random if it finds a random real drawn in accordance with density μ in the tree.

An ideal GSAT has a great asymptotic bounds for its depth and by analysing a random GSAT we want to say that with high probability the depth of some key remains the same as it was after the last rebuild and this fact is true up to a constant until the next rebuild. The following lemmas in this sub-chapter together show this fact.

Lemma 17. Let T be a $\mu[a,b]$-random GSAT and let T' be a subtree of T. Then there are reals c, d such that T' is a $\mu'[c,d]$-random GSAT such that $a \leqslant c < d \leqslant b$ and for $x \in [c,d]$:

$$\mu'(x) = \frac{\mu(x)}{\int_c^d \mu(x)}.$$

This lemma can be proven in the same way as in [8], Lemma 4.

Lemma 18. Let T be a $\mu[a,b]$-random GSAT with number of accesses to all its elements equal to m, and let T' be the direct subtree of T. Then, $m(T')$ is $O(\frac{m}{\mathrm{D}(m)+1})$ with probability at least $1 - O(\frac{\mathrm{D}(m)+1}{m})$.

Proof. Let m_0 be the initial size of T after its last rebuild. Then, $m \leqslant \frac{5}{4} \times m_0$ and $k \leqslant \frac{1}{4} \times m_0$ accesses were made into T since it was rebuilt for the last time. When T was rebuilt for the last time no more than $\frac{m}{\mathrm{D}(m)+1}$ accesses had elements that were stored in $m(T')$. Since then some additional accesses X were made in T'.

From here on let us treat these accesses as a separate elements which differ by some small epsilon. For example, if key x had 5 accesses, we will make from

it five different elements, which then fit in the segment $[x - \frac{1}{2}, x + \frac{1}{2}]$ so it will not intersect with other keys.

From this point of view, no more than $\frac{m}{\mathrm{D}(m)+1}$ elements were stored in $m(T')$ and X additional items were stored in T', so we are in the same configuration as in [8], Lemma 5.

Repeating the reasoning from there, the probability that more than $\frac{5}{4} \times \frac{m}{\mathrm{D}(m)+1}$ accesses were made into the tree T' is $\leqslant \frac{1}{2} \times \frac{\mathrm{D}(m)+1}{m}$, so $m(T') = \frac{m}{\mathrm{D}(m)+1}$ with probability $1 - O(\frac{\mathrm{D}(m)+1}{m})$. $\square$

Lemma 18 illustrates the self-organizing feature of GSATs and now we are ready to prove expected total cost of executed operations.

Lemma 19. Let μ be a density with finite support $[a, b]$. Then the expected total cost of processing a sequence of m μ-random insertions, random deletions and μ-random contains to an initially empty GSAT is $O(m \cdot d)$, where d is the depth of the ideal GSAT. Thus, the expected amortized cost of insertion, deletion or contains is $O(d)$.

Proof. Let $f(m)$ be the expected number of tokens put down by the m-th operation. Then

$$f(m) \leqslant 1 + f\Big(O\Big(\frac{m}{\mathrm{D}(m)+1}\Big)\Big) + O(\log m) \times O\Big(\frac{\mathrm{D}(m)+1}{m}\Big).$$

This can be seen as follows: if the operation goes into a subtree of size $O(\frac{m}{\mathrm{D}(m)+1})$, then we put down $1 + f(O(\frac{m}{\mathrm{D}(m)+1}))$ tokens. If it does not, then we put down at most $O(\log m)$ tokens by Lemma 12. The probability of the latter event is $O(\frac{\mathrm{D}(m)+1}{m})$ by Lemma 18, therefore, since $\mathrm{D}(m)$ is sqrt-bounded:

$$O(\log m) \times O\Big(\frac{\mathrm{D}(m)+1}{m}\Big) \leqslant O(\log m) \times O\Big(\frac{2}{\sqrt{m}}\Big) = O\Big(\frac{2 \cdot \log m}{\sqrt{m}}\Big) = O(1).$$

Thus, $f(m) = O(d)$. $\square$

Lemma 20. The expected amortized cost of `insert`, `delete` or `contains` operations (not counting the time for the preceding search) for element x with $ac(x)$ accesses in GSAT is $O(\log(\frac{m}{ac(x)}))$, that is, the total expected cost of the first m insertions, deletions and contains in GSAT is $O\Big(m + \sum\limits_{i=1}^{n} ac(x_i) \times \log\Big(\frac{m}{ac(x_i)}\Big)\Big)$.

Proof. Let us consider element x with ac accesses, having depth d. Then if k is the first value such that $ac \geqslant \frac{m}{2^k} \implies ac \leqslant \frac{m}{2^{k-1}} \implies k \leqslant \log_2(\frac{m}{ac}) + 1$. Also

$d \leqslant k$ since the number of requests from ancestor to descendant in GSAT decreases at least twice, thus, x has expected depth $O(\log(\frac{m}{ac}))$. $\square$

The Lemma 20 means that if we can find next subtree for key in $O(1)$ for each node, GSAT satisfies static-optimality property (1) in expected way.

Theorem 21. SABT holds static-optimality property in the expected way.

Proof. For SABT each node has degree B, where B is a constant, thus, next subtree for each node can be found in $O(1)$. $\square$

However, this fact is not true for SAIT and SALT, because currently for both of them $Time(\mathrm{S}(T, key)) = O(\log n)$. Nevertheless, this bound can be improved for SAIT with the help of ID array and by imposing restrictions on the probability density function.

2.2.3. Search

For every GSAT the function $\mathrm{S}(T, key)$ can be implemented in that way (see listing 6), so $Time(\mathrm{S}(T, key)) = O(\log n)$, because GSAT is an internal tree.

Listing 6 – $\mathrm{S}(T, key)$ basic implementation

```
1  fun S(Node T, E key):
2    a = 0
3    b = T.k + 1
4    while b − a > 1:
5      c = ⌊(a+b)/2⌋
6      if T.rep[c] < key:
7        a = m
8      else:
9        b = m
10   return b
```

However, let us make the separate analysis for SAIT, similar to [8], to see that by spending more memory for GSAT we can speed up $\mathrm{S}(T, key)$.

A density μ is `smooth` for a parameter α, $\frac{1}{2} \leqslant \alpha < 1$, if there are constants a, b and d such that $\mu(x) = 0$ for $x < a$ and $x > b$ and such that for all c_1, c_2, c_3, $a \leqslant c_1 < c_2 < c_3 \leqslant b$, and all integers n and m with $m = \lceil n^{\alpha} \rceil$,

$$\int_{c_2 - \frac{c_3 - c_1}{m}}^{c_2} \mu[c_1, c_3](x)\, dx \leqslant d \times n^{-\frac{1}{2}},$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$ and $\mu[c_1, c_3](x) = \frac{\mu(x)}{p}$ for $c_1 \leqslant x \leqslant c_3$ where $p = \int_{c_1}^{c_3} \mu(x) dx$. We use the same definition of smoothness as in [8]. Now let

us prove that distributions like `zipf`, `90/10`, `70/30` and `uniform` are smooth for any α. The following lemma proves it.

Lemma 22. Let μ be a probability density function for $[a, b]$ and which values on $[a, b]$ lies in $[x, y]$, where $x > 0$. Then, μ is smooth for any parameter $\alpha \in [\frac{1}{2}; 1)$.

Proof. At first, we fix a parameter α, n and $m = \lceil n^\alpha \rceil$. Then, consider any triple of reals c_1, c_2, c_3 such that $a \leqslant c_1 < c_2 < c_3 \leqslant b$. After denoting $c_3 - c_1$ as len, and $\frac{c_3 - c_1}{m}$ as $mlen$, consider the integral from the definition of smoothness:

$$\int_{c_2 - \frac{c_3 - c_1}{m}}^{c_2} \mu[c_1, c_3](x)\, dx = \int_{c_2 - mlen}^{c_2} \mu[c_1, c_3](x)\, dx \leqslant \frac{y \times mlen}{y \times mlen + x \times (len - mlen)}$$

$$= \frac{y \times len}{y \times len + x \times len \times (m-1)} = \frac{y}{y + x \times (m-1)} = \frac{y}{y - x + x \times m}$$

$$\leqslant \frac{y}{y - x + x \times \sqrt{n}} \leqslant \frac{d}{\sqrt{n}} \implies \frac{y}{x + \frac{y-x}{\sqrt{n}}} \leqslant d.$$

So we can take $d = 1$ and the lemma is proved. □

An array ID, used for SAIT, initialized and used in the same way as in [8].

Lemma 23. Let μ be a smooth density a parameter α and let T be a $\mu[a, b]$-random SAIT with parameter α. Then the expected search time in the root array is $O(1)$.

Proof. Similar to the reasoning in [8], changing the view from accesses to different elements, if we have spent $l + 1$ iterations while searching in ID array, at least $t = l \times \sqrt{m_0}$ accesses lie in the interval of $ID\left[y - \frac{b-a}{m}, y\right]$, where m_0 is the number of accesses after the last rebuild. The probability of the last event is at most $\left(\frac{3 \times d}{l}\right)^{l \cdot \sqrt{m_0}}$ by claim from [8], and, hence, the expected number of iterations over the array is

$$\sum_{l \geqslant 1} \min\left(1, \left(\frac{3 \times d}{l}\right)^{l \cdot \sqrt{m_0}}\right) = O(1). \quad \square$$

Now, we are ready to proof the bound on the expected search time in SAIT.

Theorem 24. Let μ be a smooth density for a parameter α. Then the expected search time of an element with the number of accesses ac in a $\mu[a, b]$-random SAIT for a parameter α is $O(\log \frac{\log m}{\log ac})$.

Proof. Let $T(m)$ be the expected search time in a $\mu[a,b]$-random SAIT with m accesses to all its elements. Then

$$T(m) = O(1) + T\Big(O\Big(\sqrt{m}\Big)\Big) + O(m^{-\frac{1}{2}}) \times O(\sqrt{m}).$$

The search time in the root array is $O(1)$ by Lemma 23. The next subtree in which we search is μ-random by Lemma 17. We have total number of accesses $O(\sqrt{m})$ with probability at least $1 - O(m^{-\frac{1}{2}})$ by Lemma 18, otherwise the search time is bounded by $O(\sqrt{m})$.

When the total number of accesses to a subtree is less or equal to ac, the search stops. So, $T(m) = O(\log \frac{\log m}{\log ac})$. □

Theorem 25. SAIT holds static-optimality in expected way.

Proof. This fact is true by the Theorem 24. □

2.2.4. Range queries

Now we extend GSAT by making it support range queries.

We start with `get(a, b)` query. Let us denote $B(T)$ to be the segment on top of which GSAT, T, was built for the last time. Until we have $[a,b] \subset B(T_i)$ we just go down to T_i, increasing node's counter. Then, being on the depth d, consider subtrees $T_i, T_{i+1}, \ldots, T_{i+j}$ with maximum j such that $\bigcup_{k=i}^{j} B(T_k) \subseteq [a,b]$. To gather all elements from $[a,b]$ we will visit all such T_i and we would also visit $REP[i], REP[i+1], \ldots REP[i+j] \subseteq [a,b]$. Initial range $[a,b]$ can be split into two parts only once, and it has already happened for the depth d. If we have $B(T_{i-1}) \cap [a,b] \neq \emptyset$, we need to do `get` query for the tree T_{i-1} with the segment $[a, REP[i-1]]$, which obviously will not split. Similarly, if $B(T_{i+j+1}) \cap [a,b] \neq \emptyset$, we have to do `get` query for the tree T_{j+1} with the segment $[REP[j], b]$, which obviously will not split again. So the amortized time for `get(a, b)` query is $O(Time(\mathrm{S}(T, key)) \cdot \max_{x \in [a,b]} depth(x) + |b-a|)$. During such request, because we split no more than once, there can not be more than two rebuilds, since no more than 2 different subtrees will be traversed. So such range query will not affect other operations asymptotic bounds and the total number of accesses after such query will be $m' \leqslant m + |b - a + 1|$. To support `get(a, b)` query we do not need to store any additional data, we just need to traverse the tree in the right order.

Now we consider queries `calculate(a, b)` (compute $\odot$ for the values whose keys from $[a,b]$) and `update(a, b, c)` (apply to each value $(\star c)$ if its

key from $[a, b]$). We extend the definition of GSAT by supplementing each node with 2 additional data structures, PF (propagate function) and PA (propagate accesses). Both of them would have size $2 \cdot k + 1$, where $(k+1)$ is the degree of the considered node and these data structures have to support following operations: (1) change segment's values by applying some function to all values from the segment; (2) to compute some value on the segment. To make this operations reasonable, let us add some properties to $\odot$ and $\star$:

a) $(a \odot b) \odot c = a \odot (b \odot c)$ — calculate-associative;

b) $(a \star b) \star c = a \star (b \star c)$ — update-associative;

c) $(a \star x) \odot (b \star x) = (a \odot b) \star x$ – distributive.

By this operations' requirements we see that Segment Tree [11], using lazy propagation technique [7], can be used, so we can build it for $O(k)$ and each operation will have time $O(\log k)$. Now we consider indices: $PF[1]$ corresponds to T_1, $PF[2]$ corresponds to $REP[1]$, $PF[3]$ corresponds to $REP[2]$, so, in general, $PF[i]$ is equal to the value that has to be propagated to $T_{\frac{i+1}{2}}$ if i is odd, or is equal to the value that has to be propagated to $REP[\frac{i}{2}]$ if i is even. The same rules used for PM, but for this array it has to propagate the number of accesses for some segment.

Now we ready to consider query `update(a, b, c)` in action: the start is the same as for `get(a, b)`: until we have $[a, b] \subset B(T_i)$ we just go down to T_i, increasing node's counter and also propagating value and accesses to subtree T_i. Then on the depth d, we have $\bigcup_{k=i}^{j} B(T_k) \subseteq [a, b]$, so we make request $(\star c)$ to $PF[2 \cdot i - 1, 2 \cdot i, \ldots, 2 \cdot j]$ and request $(+1)$ for the same range for PM and then again split no more than once at the current node on the depth d, and finish searching if we processed all keys from the request query.

For `calculate(a, b)` everything is similar as to `update(a, b, c)`: we also need not to forget to increase counter and propagate value to the next visited node, by we do not change PF, instead, we calculate function on its some segment. Also we have to preserve associative property there, so order of applying results from subtree is important.

Now we can say that such range queries complicate all the previous operations, because we additionally compute accesses for propagation which is done in $O(\log n)$, so such queries can be done in $O(\log n \cdot \max_{x \in [a,b]} depth(x))$.

Storing PF and PM helps us to speed up operations, but if we have GSAT with $\mathrm{D}(m) = O(1)$, we can use the same approach as for `get(a, b)` by explicitly pushing and computing corresponding function value and the number of accesses.

2.3. Concurrent lock-free extension

To make GSAT concurrent, we use approach from[10]. The main difference is that instead of storing sizes, we store the number of accesses and their changes. Thus, in the function *markAndCount* [10] we count the number of accesses made to the subtree and then choose representatives on the first level. Next, we use collaborative rebuild, changing initial builder to our GSAT builder, which uses binary searches for splitting subtrees. This extension will support `insert`, `delete` and `contains` operations, but not range queries.

Conclusions on Chapter 2

In this chapter, we described a generic approach for creating self-adjusting data structures, looked through different parameterizations and saw how they reflect time complexity and memory consumption. Moreover, we have proved static-optimality in expected way for SABT and separately for SAIST, assuming that probablity density is smooth. We described how to make GSAT support range queries and saw, what asymptotic bounds such operations have as well as how they affect others operations. Finally, we briefly discussed how to transform sequential GSAT into the concurrent one.

CHAPTER 3. EXPERIMENTS AND RESULTS

In this chapter, we compare different GSAT parameterizations with their classic versions and splay-tree on different workloads.

These workloads were launched on a system with the following characteristics: Intel Xeon Gold 6240R CPU @ 2.40GHz, RAM 256Gb.

3.1. Graphs structure

OX axis shows the size of the set of keys and OY axis shows the number of operations per second on one process — the higher the better. Each benchmark was launched for 20 seconds. Each point on the graph was obtained by the following procedure: we launched the same workload with same parameters for five times and then plotted the average value among them to prevent any fluctuations.

Each picture consists of two graphs, where on the left side we have *only-find* queries, while on the right side we have *mixed* queries.

SA2T on the graphs below means parametrization of SABT, where $B = 2$.

3.2. x/y workloads

We start with the experiments on `x/y` workloads. There are two key sets S_r and S_u which have sizes $y \cdot |S|$, where S is the whole set of keys and the keys in these sets are chosen uniformly at random. In that workload, x% of contains operations choose an argument from S_r while the rest contains operations choose an argument from $S \setminus S_r$. The same happens with update operations and set S_u.

In other words, the more difference $x - y$ becomes the more workload skewed.

We consider three workloads of different skewness: `100/100`, i.e., uniform; `70/30`; and `90/10`. The amount of update operations is also tunable. We have two settings: 0% and 60%, i.e., 30% of inserts and 30% of removes. The operations are chosen uniformly with respect to the proportions.

At first, we look on the uniform workload (Figure 4):

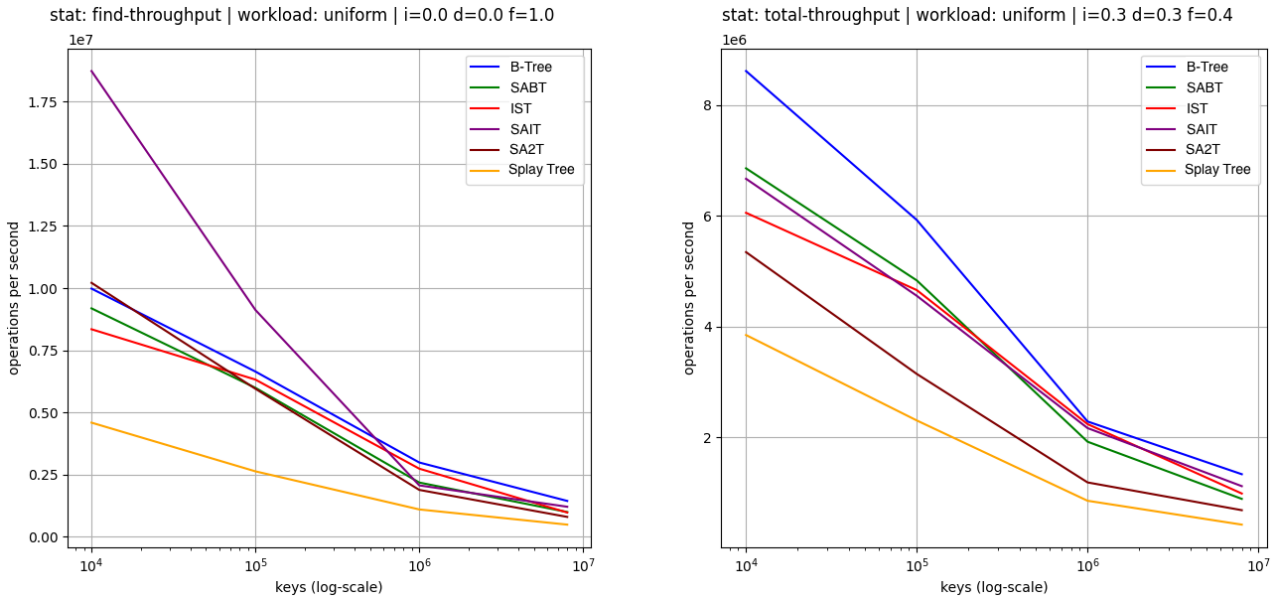


Figure 4 – Uniform workload

Then, we consider `70/30` workload (Figure 5):

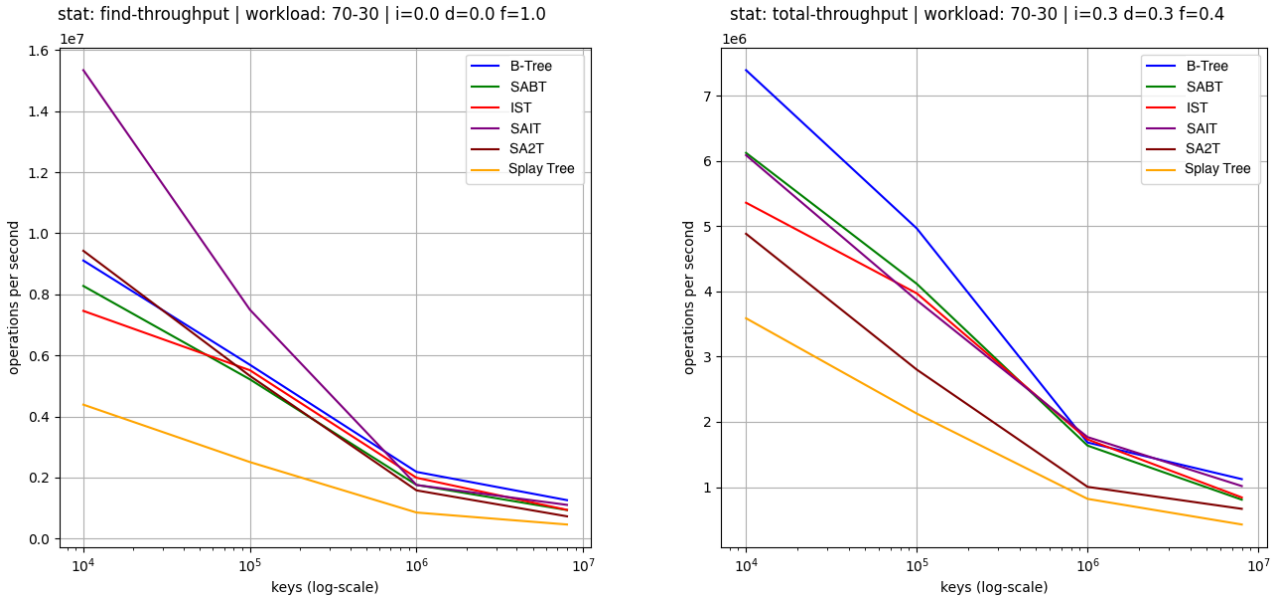


Figure 5 – 70/30 workload

Finally, we look onto `90/10` workload (Figure 6):

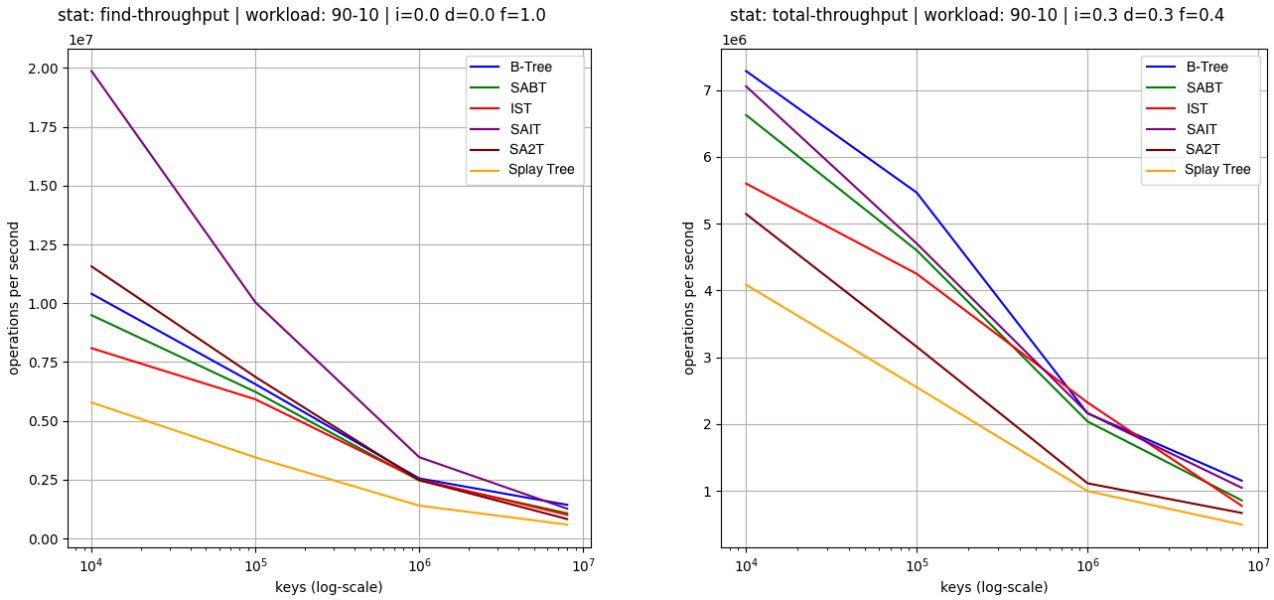


Figure 6 – 90/10 workload

As we can see, the more skewed the access distribution becomes, the better GSATs works — they are superior on only-find queries and slightly worse on mixed queries to classic tree versions. Moreover, GSATs surpass splay-tree on every workload. SAIT is superior to other trees, because on average, it takes ~ 1 iterations to determine the next subtree, thanks to the ID array.

3.3. zipf workload

Let us consider last heavy-tailed distribution workload — Zipf, parameterized with 1 and see, how implemented data structures behave on it (Figure 7):

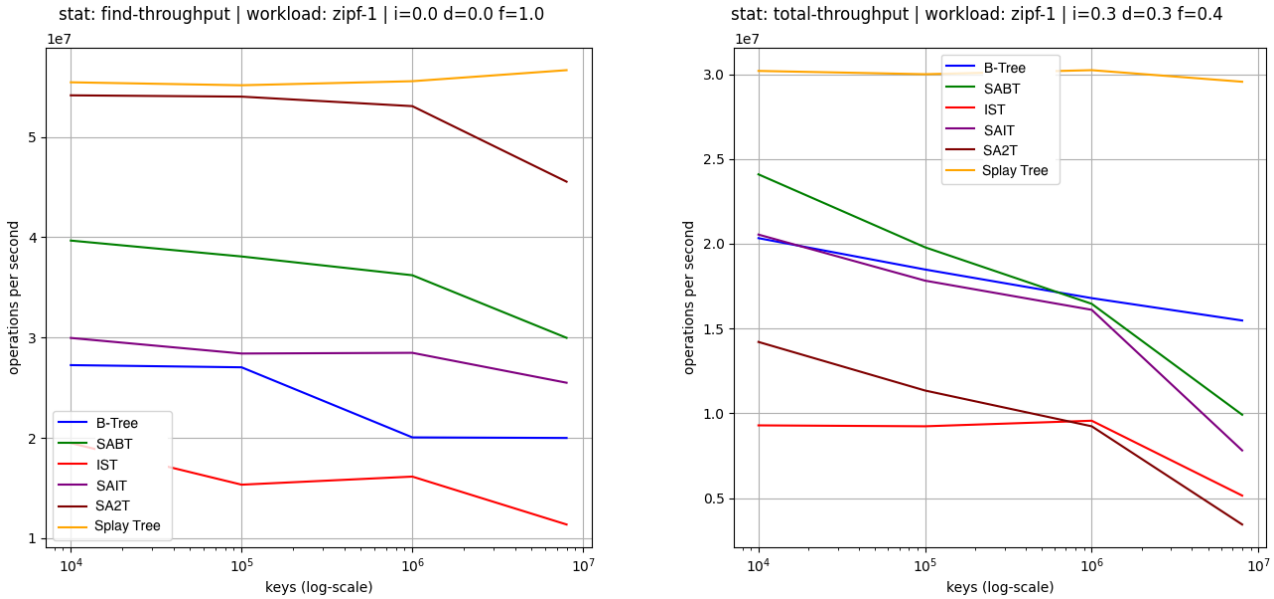


Figure 7 – Zipf-1 workload

Here splay-tree equivalent to SA2T on only-find queries, however, splay-tree outperforms GSAT for mixed queries — it is reasonable, because splay-tree store near the root only a small amount of requesting keys while GSATs spend time on rebuilding subtrees to balance them.

Conclusions on Chapter 3

In this chapter, we provided results of the experiments that indicate that GSATs implementations, especially SAIT, show overall better results on the general and read-only highly skewed workloads than the classic tree versions or Splay Tree. Moreover, it is not significantly worse on mixed workloads then their original tree versions. The obtained practical results coincide with the expectations after the received theoretical bounds.

CONCLUSION

The Generic Self-Adjusting Tree Approach can be applied to different data structures to make them self-adjusting by extracting their main characteristics into the functions D(m) and S($node, key$). The general analysis for GSATs is done practically in spite of D(m), however, to get a more precise bounds it is better to consider each parameterization separately. So we have proved the static-optimality property in expected way for the SABT and SAIT.

The GSAT approach is not limited to only basic set operations, it can be extended to support range queries, which, unlike other self-adjusting data structures, considers the number of accesses for an element to restructure the tree in accordance with the data distribution.

GSATs have shown themselves decent in practice on different workloads in comparison with original, non-self-adjusting, trees as well as with other self-adjusting trees — the more skewed workload, the more GSAT overpower non-self-adjusting tree, however, it requires more time in comparison with Splay Tree to adapt to data, which can be seen on the zipf-1 workload.

REFERENCES

1 *Abramson N.* Information theory and coding. — New York: McGraw-Hill, 1983.

2 *Bayer R.*, *McCreight E.* Organization and maintenance of large ordered indices. — 1970.

3 *Sherk M.* Self-adjusting k-ary search trees // J. Algorithms. — 1995. — P. 24–44.

4 Benchmarking cloud serving systems with YCSB / B. F. Cooper [et al.] // Proceedings of the 1st ACM symposium on Cloud computing. — 2010. — P. 143–154.

5 CBTree: A Practical Concurrent Self-adjusting Search Tree / Y. Afek [et al.] // Proceedings of the 26th International Conference on Distributed Computing. — Salvador, Brazil : Springer-Verlag, 2012. — P. 1–15. — (DISC'12). — URL: http://dx.doi.org/10.1007/978-3-642-33651-5_1.

6 *Knuth D. E.* The art of computer programming. Vol. 3. — Pearson Education, 1997.

7 Lazy Propagation in Segment Tree [Электронный ресурс]. — 2023. — URL: https://www.geeksforgeeks.org/lazy-propagation-in-segment-tree/.

8 *Mehlhorn K.*, *Tsakalidis A.* Dynamic interpolation search. — 1991. — URL: https://people.mpi-inf.mpg.de/~mehlhorn/ftp/DynamicInterpolationSearch.pdf.

9 *Poess M.*, *Floyd C.* New TPC benchmarks for decision support and web commerce // ACM Sigmod Record. — 2000. — Vol. 29, no. 4. — P. 64–71.

10 *Prokopec A.*, *Brown T.*, *Alistarh D.* Analysis and Evaluation of Non-Blocking Interpolation Search Trees // Proceedings of Principles and Practice of Parallel Programming 2020. — 2020. — (PPoPP'20).

11 Segment tree [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/wiki/Segment_tree.

12 *Tarjan R.*, *Sleator D.* Self-Adjusting Binary Search Trees. — 1985. — URL: https://www.cs.princeton.edu/courses/archive/fall07/cos521/handouts/self-adjusting.pdf.

13 Zipf's law [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/wiki/Zipf%5C%27s_law.