mathematics

MDPI

Article

Noisy Tree Data Structures and Quantum Applications

Kamil Khadiev [1,*], Nikita Savelyev [2], Mansur Ziatdinov [1] and Denis Melnikov [1]

[1] Institute of Computational Mathematics and Information Technologies, Kazan Federal University, 420008 Kazan, Russia; 1mansur.ziyatdinov@kpfu.ru (M.Z.); deimelnikov@stud.kpfu.ru (D.M.)
[2] Intel Inc., D-81671 Munich, Germany
* Correspondence: kamil.hadiev@kpfu.ru

Abstract: We suggest a new technique for developing noisy tree data structures. We call it a "walking tree". As applications of the technique we present a noisy Self-Balanced Binary Search Tree (we use a Red–Black tree as an implementation) and a noisy segment tree. The asymptotic complexity of the main operations for the tree data structures does not change compared to the case without noise. We apply the data structures in quantum algorithms for several problems on strings like the string-sorting problem and auto-complete problem. For both problems, we obtain quantum speed-up. Moreover, for the string-sorting problem, we show a quantum lower bound.

Keywords: noisy computation; self-balanced search tree; segment tree; quantum computation; quantum algorithms; string processing; sorting

MSC: 68P05; 68W20; 68Q12

1. Introduction

Tree data structures are well-known and used in different algorithms. At the same time, when we construct algorithms with random behavior like randomized and quantum algorithms, we should consider error probability. We suggest a general method for updating a tree data structure in the noisy case. We call it Walking tree. For a tree of height h, we consider an operation for processing all nodes from a root to a target node. Suppose the running time of the operation is $O(h \cdot T)$, where T is running time required to process a node. Then, if navigation by the tree can have an error, our technique allows us to carry it out with $O(\log(1/\varepsilon) + h \cdot T)$ running time, where ε is the error probability for the whole operation. Note that the standard way to handle probabilistic navigation is the success probability boosting (repetition of the noisy action) with $O(h \cdot \log(1/\varepsilon) + h \cdot T)$ complexity.

Our technique is based on results for the noisy Binary search algorithm [1]. The authors of that paper present an idea based on the random walk algorithm for a balanced binary tree that can be constructed for the binary search algorithm. We generalize the idea for a tree with any structure that allows us to apply the method to a wide class of tree data structures. Different algorithms for noisy search, especially a noisy tree, and graph processing and search were considered in [2–8]. We apply our technique to two tree data structures. The first one is the Red–Black tree [9], which is an implementation of a self-balanced binary search tree [9]. If the key comparing procedure has a bounded error, then our noisy self-balanced binary search tree allows us to conduct add, remove, and search operations in $O(\log(N/\varepsilon))$ running time, where ε is the error probability for a whole operation and N is the number of nodes in the tree. In the case of $\varepsilon = 1/Poly(N)$, we have $O(\log N)$ running time and the noisy key comparing procedure does not affect running time (asymptotically). At the same time, if we use the success probability boosting technique, then the running time is $O((\log N)^2)$. The second one is the Segment tree [10,11]. If the indexes comparing procedure has a bounded error, then our noisy segment tree allows us to conduct update and request operations in $O(\log(N/\varepsilon))$ running time, where ε is the error probability for

check for updates

Citation: Khadiev, K.; Savelyev, N.; Ziatdinov, M.; Melnikov, D. Noisy Tree Data Structures and Quantum Applications. *Mathematics* **2023**, *11*, 4707. https://doi.org/10.3390/math11224707

Academic Editor: Jonathan Blackledge

Received: 25 September 2023
Revised: 6 November 2023
Accepted: 15 November 2023
Published: 20 November 2023

a whole operation and N is the number of leaves. In the case of $\varepsilon = 1/Poly(N)$, we have $O(\log N)$ running time. So, we obtain a similar advantage. We use these data structures in the context of quantum computation [12] which is one of the hot topics in the last decades. There are many problems where we can obtain a quantum speed-up. Some of them can be found here [13,14], including graph problems [15–21] and string processing problems [22–28]. Quantum algorithms have randomized behavior, so it is important to use noisy data structures for this model. We use the quantum query model [29] as the main computational model for the quantum algorithms. We apply the walking tree method for the following problems.

The first one is the string-sorting problem. We want to sort n strings of a length l in lexicographical order. However, quantum algorithms cannot sort arbitrary comparable objects faster than $\Omega(n \log n)$ [30,31]. At the same time, some results improve the hidden constant [32,33]. Other researchers investigated the space-bounded case [34]. The situation with sorting strings is a little bit different. We know that the classical Radix sort algorithm has $O(nl)$ running time [9] for a finite-size alphabet. That is faster than sorting algorithms for arbitrary comparable objects. Here, the lower bound for classical (randomized or deterministic) algorithms is $\Omega(nl)$. In the quantum case, faster algorithms with $O(n(\log n) \cdot \sqrt{l}) = \tilde{O}(n\sqrt{l})$ running time are known [35,36]. Here, $\tilde{O}$ does not consider log factors. In this paper, we suggest a simpler implementation based on a noisy red–black tree.

The second one is the Auto-Complete Problem. We have two kinds of queries: adding a string s to the dictionary $\mathcal{S}$ and querying the most frequent completion of a string t from the dictionary. We call s a completion of t if t is a prefix of s. Assume that L is the total sum of all lengths of strings from all queries. We solve the problem using quantum string comparing algorithm [35–39] and noisy Red–Black tree. The running time of the quantum algorithm is $O(\sqrt{nL} \log n)$. The lower bound for quantum running time is $\Omega(\sqrt{L})$. At the same time, the best classical algorithm based on trie (prefix tree) [40–43] has $O(L)$ running time. That is also the classical (deterministic or randomized) lower bound $\Omega(L)$. So, we obtain quantum speed-up if most of the strings have $\Omega((\log n)^2)$ length.

2. Preliminaries

In the paper, we use the following notation.

- $\log N$ means $\log_2 N$ that is a logarithm with base 2.
- $Poly(N)$ means a polynomial dependent on N. Formally, $Poly(N) = \sum_{i=0}^{\alpha} a_i N^i$ for some constants α and $a_0, \dots a_\alpha$.
- $O(f(N))$ means big-O notation, upper bound. Formally, $g(N) = O(f(N))$ if $\lim_{N\to\infty} g(N)/f(N) \leq \mathcal{C}$ for some $\mathcal{C} = const$.
- $\tilde{O}(f(N))$ means big-O notation with ignoring log factor. Formally, $g(N) = \tilde{O}(f(N))$ if $\lim_{N\to\infty} g(N)/f(N) \leq \mathcal{L}$ for some $\mathcal{L} = Poly(\log N)$.
- $\Omega(f(N))$ means big-Omega notation, lower bound. Formally, $g(N) = \Omega(f(N))$ if $\lim_{N\to\infty} g(N)/f(N) > \mathcal{C}$ for some $\mathcal{C} = const$.

In the paper, for two strings s and t, the notation $s < t$ means s precedes t in the lexicographical order. Let $|s|$ be the length of a string s.

2.1. Graph Theory

Let us consider a rooted tree G. Let $\mathcal{V}(G)$ be the set of nodes (vertices), and $\mathcal{E}(G)$ be the set of edges. Let one fixed node be the root of the tree. Assume, a procedure GetTheRoot(G) returns it. A path P is a sequence of nodes $(v_1, \dots, v_k)$ that are connected by edges, i.e., $(v_i, v_{i+1}) \in E$ for $i \in \{1, \dots, k-1\}$. Note, that there are no duplicates among $v_1, \dots, v_k$. Here, k is the length of the path. We use $v \in P$ notation if there is j such that $v_j = v$. The notation is reasonable because there are no duplicates in a path. Because G is a tree, the path between any two nodes u and v is unique. The distance $dist(v, u)$ between two nodes v and u is the length of the path between them. The height of v is the distance between it and the root that is $dist(root, v)$. Let $h(G) = \max_{v\in\mathcal{V}(G)} dist(root, v)$ be the tree's height.

For a node v, $\textsc{Parent}(v)$ is a parent node, where $dist(root, \textsc{Parent}(v)) + 1 = dist(root, v)$ and $(\textsc{Parent}(v), v) \in \mathcal{E}(G)$; a set of children is $\textsc{Children}(v) = \{u : \textsc{Parent}(u) = v\}$.

2.2. Quantum Query Model

In Section 6 we suggest quantum algorithms as applications for our data structures. We have only one quantum subroutine, and the rest part of the algorithm is classical. One of the most popular computation models for quantum algorithms is the query model. We use the standard form of the quantum query model. Let $f : D \to \{0,1\}, D \subseteq \{0,1\}^M$ be an M variable function. We wish to compute on an input $x \in D$. We are given oracle access to the input x, i.e., it is implemented by a specific unitary transformation that is usually defined as $|i\rangle|z\rangle|w\rangle \to |i\rangle|z + x_i \pmod 2\rangle|w\rangle$ where the $|i\rangle$ register indicates the index of the variable we are querying, $|z\rangle$ is the output register, and $|w\rangle$ is some auxiliary work-space. The operation is implemented by the CNOT gate. An algorithm in the query model consists of alternating applications of arbitrary unitaries independent of the input and the query unitary, and measurement in the end. The smallest number of queries for an algorithm that outputs $f(x)$ with probability $\geq \frac{2}{3}$ on all x is called the quantum query complexity of the function f, and $Q(f)$ notation is used for it. We use the term running time instead of query complexity to remove confusion with "query" in the definition of problems in Section 6. In the paper, we use modifications of Grover's search algorithm [44,45] as quantum subroutines. For these subroutines, the time complexity is more than the query complexity for additional log factor [46,47]. Note that in the general case, we can consider a function f with non-Boolean arguments. It can be simulated by a Boolean-argument-function case using a binary representation of arguments.

The modification of Grover's search algorithm [48] in the case of a known number of solutions can be used in our problems. We refer the readers to [12,29] for more details on quantum computing.

3. Main Technique: A Walking Tree

In this section, we present a rooted tree that we call a walking tree. It is a utility data structure for noisy computation for the main data structure. Here we use it for the following data structures: (i) Binary Search Tree. We assume that elements comparing procedures can have errors. (ii) Segment Tree. We assume that the indexes (borders of segments) comparing procedure can have errors.

Note that the walking tree is a general technique, and it can be used for other tree data structures. Let us present the general idea of the tree. The technique is motivated by [1]. Assume that G is a rooted tree. We are interested in moving from the root to a specific (*target*) node operation. Assume that we have the following procedures:

- $\textsc{GetTheRoot}(G)$ returns the root node of the tree G.
- $\textsc{SelectChild}(v)$ returns the child of the node $v \in \mathcal{V}(G)$ that should be reached from the node v. We assume that there is only one child that should be reached from a node.
- $\textsc{IsItTheTarget}(v)$ returns *True* if the node $v \in \mathcal{V}(G)$ is the last node that should be visited in the operation; and returns *False* otherwise.
- $\textsc{ProcessANode}(v)$ processes the node $v \in \mathcal{V}(G)$ in the required way.
- $\textsc{IsANodeCorrect}(v)$ returns *True* if the node $v \in \mathcal{V}(G)$ should be visited during the the operation; and returns *False* if the node is visited because of an error.

Assume that the operation has the following form (Algorithm 1).

Algorithm 1 An operation on the tree G

$v \leftarrow$ GETTHEROOT(G)
PROCESSANODE(v)
while ISITTHETARGET$(v) = False$ **do**
  $v \leftarrow$ SELECTCHILD(v)
  PROCESSANODE(v)
end while

Let us consider the operation such that "navigation" procedures (that are SELECTCHILD, ISANODECORRECT, and ISITTHETARGET) can return an answer with an error p, where $p < 0.5 - \delta$, where $0 < \delta \leq 0.5$. We assume that error events are independent. We need to isolate p from 0.5. We cannot be sure that the algorithm works if $p \to 0.5$. That is why we use $p < 0.5 - \delta$ statement. Our goal is to conduct the operation with an error ε. Note that in the general case, ε can be non-constant and depend on the number of tree nodes. Let $h = h(G)$ be the height of the tree. The standard technique is boosting success probability. On each step, we repeat SELECTCHILD procedure $O(\log(h/\varepsilon))$ times and choose the most frequent answer. In that case, the error probability of the operation is at most ε, and the running time of the operation is $O(h \log(h/\varepsilon) + h \cdot T)$, where T is complexity of PROCESSANODE procedure. Our goal is to have $O(h + \log(h/\varepsilon) + h \cdot T) = O(\log(h/\varepsilon) + h \cdot T)$ running time.

Let us construct a rooted tree W by G such that the set of nodes of W has a one-to-one correspondence to the nodes of G and the same with sets of edges. We call W a *walking tree*. Let $\lambda_W : \mathcal{V}(W) \leftrightarrow \mathcal{V}(G)$ and $\lambda_G : \mathcal{V}(G) \leftrightarrow \mathcal{V}(W)$ be bijections between these two sets. For simplicity, we define procedures for W similar to the procedures for G. Suppose $u \in \mathcal{V}(W)$, then GETTHEROOT(W) = λ_G(GETTHEROOT(G)); SELECTCHILD(u) = λ_G(SELECTCHILD$(\lambda_W(u))$); ISITTHETARGET(u) = ISITTHETARGET$(\lambda_W(u))$; ISANODECORRECT(U) = ISANODECORRECT $(\lambda_W(u))$. Note that the navigation procedures are noisy (have an error). We reduce the error probability to 0.1 by constant number of repetitions (using the boosting success probability technique). Additionally, we associate a counter $c(u)$ with a node $u \in \mathcal{V}(W)$ that is a non-negative integer number. Initially, values of counters are 0 for all nodes, i.e., $c(u) = 0$ for each $u \in \mathcal{V}(W)$.

We invoke a random walk by the walking tree W. The walk starts from the root node GETTHEROOT(W). Let us discuss processing $u \in \mathcal{V}(W)$. Firstly, we check the counter's value $c(u)$. If $c(u) = 0$, then we carry out steps from 1.1 to 1.3.

Step 1.1. We check the correctness of current node using ISANODECORRECT(u) procedure. If the result is *True*, then we go to Step 1.2. If the result is *False*, then we are here because of an error, and we go up by changing $u \leftarrow$ PARENT(u). If the node u is the root, then we stay in u.

Step 1.2. We check whether the current node is target using ISITTHETARGET(u) procedure. If it is *True*, then we increase the counter $c(u) \leftarrow 1$. If it is *False*, then we go to Step 1.3.

Step 1.3. We go to the children $u \leftarrow$ SELECTCHILD(u).

If $c(u) > 0$, then we carry out Step 2.1. We can say that the counter $c(u)$ is a measure of confidence that u is the target node. If $c(u) = 0$, then we should continue walking. If $c(u) > 0$, then we think that u is the target node. If a bigger value of $c(u)$ means we are more confident that it is the target node.

Step 2.1. If ISITTHETARGET$(u) = True$, then we increase the counter $c(u) \leftarrow c(u) + 1$. Otherwise, we decrease the counter $c(u) \leftarrow c(u) - 1$. So, we become more or less confident in the fact that the node u is the target.

The walking process stops in s steps, where $s = O(h + \log(1/\varepsilon))$. The stopping node u is the target one. After that, we carry out the operation with the original tree G. We store path in $Path = (v^1, \ldots, v^k)$, such that $v^k = \lambda_W(u)$, $v^i =$ PARENT(v^{i+1}), and v_1 is the root node of G. Then, we process them using PROCESSANODE(v^i) for i from 1 to k. Let a procedure ONESTEP(u) be one step of the walking process on the walking tree W. It accepts the current node u and returns the new node. The code representation of the procedure is in Algorithm 2.

Algorithm 2 One step of the walking process, OneStep(u). The input is $u \in \mathcal{V}(W)$ and the result is the node for the next step of the walking

```
if c(u) = 0 then
    if IsANodeCorrect(u) = False then                    ▷ Step 1.1
        if u ≠ GetTheRoot(W) then
            u ← Parent(u)
        end if
    else
        if IsItTheTarget(u) = True then                  ▷ Step 1.2
            c(u) ← 1
        else
            u ← SelectChild(u)                           ▷ Step 1.3
        end if
    end if
else
    if IsItTheTarget(u) = True then                      ▷ Step 2.1
        c(u) ← c(u) + 1
    else
        c(u) ← c(u) − 1
    end if
end if
```

The whole algorithm is presented in Algorithm 3.

Algorithm 3 The walking algorithm for $s = O(h + \log(1/\varepsilon))$ steps

```
u ← GetTheRoot(W)
for j ∈ {1, ... s} do
    u ← OneStep(u)
end for
v ← λ_W(u), Path = (v), k = 1
while v ≠ GetTheRoot(G) do
    v ← Parent(v)
    Path = v ∘ Path   ▷ Here ∘ means the concatenation of two sequences. The line adds
the node to the beginning of the path sequence
    k ← k + 1                          ▷ The length of the path sequence
end while
for i ∈ {1, ... k} do
    ProcessANode(v^i)
end for
```

Let us discuss the algorithm and its properties. On each node, we have two options, we go in the direction of the target node or the opposite direction.

Assume that $c(u) = 0$ of the current node u. If we are in a wrong branch, then the only correct direction is the parent node. If we are in the correct branch, then the only correct direction is the correct child node. All other directions are wrong. Assume that $c(u) > 0$. If we are in the target node, then the only correct direction is increasing the counter, and the wrong direction is decreasing the counter. Otherwise, the only correct direction is decreasing the counter.

Choosing the direction is based on the results of at most three invocations of navigation procedures (SelectChild, IsANodeCorrect, and IsItTheTarget). Remember that we reach 0.1 error probability using a constant number of repetitions. Due to the independence of error events, the total error probability of choosing a direction is at most 0.3. So, the probability of moving in correct direction is at least $2/3$ and for a wrong direction it is at most $1/3$. Let us show that if $s = O(h + \log(1/\varepsilon))$, then the error probability for

an operation on G is ε. Note that ε can be non-constant. In Corollarys 1 and 2, we have $\varepsilon = 1/Poly(|\mathcal{V}(G)|)$.

Theorem 1. *Given error probability $\varepsilon \in (0, 0.5)$, Algorithm 3 completes the same action as Algorithm 1 with a running time of $O(h + \log(1/\varepsilon))$.*

Proof. Let us consider the walking tree. We emulate the counter by replacing it with a nodes chain of length $s+1$. Formally, for a node $u \in \mathcal{V}(W)$ we add $s+1$ nodes $d^u_1, \ldots, d^u_{s+1}$ such that $\textsc{Parent}(d^u_1) = u$, $\textsc{Parent}(d^u_i) = d^u_{i-1}$ for $i \in \{2, \ldots, s+1\}$. The only child of d^u_i is d^u_{i+1} for $i \in \{1, \ldots, s\}$, and d^u_{s+1} does not have children.

In that case, the increasing of $c(u)$ can be emulated by moving from $d^u_{c(u)}$ to $d^u_{c(u)+1}$. The decreasing can be emulated by moving from $d^u_{c(u)}$ to $d^u_{c(u)-1}$. We can assume that d^u_0 is the node u itself.

Let u_{target} be the target node, i.e., $\textsc{IsItTheTarget}(u_{target}) = True$. Let us consider the distance L between the target node $d^{u_{target}}_{s+1}$ and the current node in the modified tree. The distance L is a random variable. Each step of the walk increases or decreases the distance L by 1. So, we can present $L = dist(root, d^{u_{target}}_{s+1}) - Y$, where $root$ is the root node of W, $Y = (Y_1 + \cdots + Y_s)$, and $Y_i \in \{-1, +1\}$ are independent random variables that represent i-th step and show increasing or decreasing the distance. Let $Y_i = +1$ if we move in the correct direction, and $Y_i = -1$ if we move in the wrong direction. Note that the probability of moving to the correct direction ($Y_i = +1$) is at least $2/3$ and the probability of moving to the wrong direction ($Y_i = -1$) is at most $1/3$. From now on, without loss of generality, we assume that $\mathbf{Pr}[Y_i = +1] = 2/3$ and $\mathbf{Pr}[Y_i = -1] = 1/3$.

If $L \leq s$, then we are in the $d^{u_{target}}_i$ node in the modified tree and in the u_{target} node in the original walking tree W. Note that $dist(root, d^{u_{target}}_{s+1}) \leq h + s$, where $h = h(W)$ by the definition of the height of a tree. Therefore, $L \leq s$ means $Y = Y_1 + \cdots + Y_s \geq h$. So, the probability of success of the operation is the probability of the $Y \geq h$ event, i.e., $\mathbf{Pr}_{success} = \mathbf{Pr}[Y \geq h]$.

Let $X_i = (Y_i + 1)/2$ for $i \in \{1, \ldots, s\}$. We treat $X_1, \ldots, X_s$ as independent binary random variables. Let $X = \sum_{i=1}^{s} X_i$. For such X and for any $0 < \delta \leq 1$, the following form of Chernoff bound [49] holds

$$\mathbf{Pr}[X < (1-\delta)\mathbf{E}[X]] \leq \exp(-\mathbf{E}[X]\delta^2/2). \tag{1}$$

Since $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[Y_i = +1] = 2/3$, we have $\mathbf{E}[X] = 2s/3$ and the inequality (1) becomes

$$\mathbf{Pr}[X < 2s \cdot (1-\delta)/3] \leq \exp(-\delta^2 s/3).$$

Substituting Y for X we, obtain

$$\mathbf{Pr}[Y < s \cdot (4(1-\delta)/3 - 1)] \leq \exp(-\delta^2 s/3).$$

From now on without loss of generality, we assume that $s = \lceil c \cdot (h + \log(1/\varepsilon)) \rceil$ for some $c > 0$. Let $\delta = \frac{1}{6}$ and $c \geq 108$.

In the following steps, we relax the inequality by obtaining less tight bounds for the target probability.

Firstly, we obtain a new lower bound

$$s \cdot (4(1-\delta)/3 - 1) = \lceil c \cdot (h + \log(1/\varepsilon)) \rceil / 9 \geq c \cdot (h + \log(1/\varepsilon))/9 \geq$$
$$\geq c \cdot h/9 \geq 108 \cdot h/9 > h,$$

and hence

$$\mathbf{Pr}[Y < h] \leq \mathbf{Pr}[Y < s \cdot (4(1-\delta)/3 - 1)]$$

Secondly, we obtain a new upper bound

$$\exp(-\delta^2 s/3) = \exp(-\lceil c \cdot (h + \log(1/\varepsilon))\rceil/108) \le \exp(-c/108 \cdot (h + \log(1/\varepsilon)) < \\ < \exp(-c/108 \cdot \log(\varepsilon^{-1})) \le \exp(-\log(\varepsilon^{-1})) = \varepsilon.$$

Combining the two obtained bounds we have

$$\mathbf{Pr}[Y < h] \le \mathbf{Pr}[Y < (\frac{4}{3}(1-\delta) - 1)s] \le \exp(-\delta^2 s/3) < \varepsilon,$$

and hence

$$\mathbf{Pr}[Y < h] < \varepsilon.$$

Considering the probability of the opposite event we finally obtain

$$\mathbf{Pr}_{success} = \mathbf{Pr}[Y \ge h] > 1 - \varepsilon.$$

□

In the next section, we show several applications of the technique.

4. Noisy Tree Data Structures

4.1. Noisy Binary Search Tree

Let us consider a Self-Balanced Search Tree [9]. It is a binary rooted tree G. Let $n = |\mathcal{V}(G)|$. We associate a comparable element $\alpha(v)$ with a node $v \in \mathcal{V}(G)$. (i) For a node $v \in \mathcal{V}(G)$, we have $\alpha(v') < \alpha(v)$ where v' is from the left sub-tree of v; and $\alpha(v'') > \alpha(v)$ where v'' is from the right sub-tree of v. (ii) The height of the tree $h(G) = O(\log n)$.

As an implementation of *Self-Balanced* Search Tree, we use a Red–Black Tree [9,50]. It allows us to add and remove a node with a specific value with $O(\log n)$ running time. Assume that the comparing procedure of two elements has an error p. Each operation (remove and add an element) has three steps: searching, carrying out the action (removing or adding), and re-balancing. Re-balancing does not invoke comparing operations, that is why it does not have an error. So, the only "noisy" procedure (which can have an error) is searching. Let us discuss it.

Let us associate $\beta(v)$ and $\gamma(v)$ with a node $v \in \mathcal{V}(G)$. That are left and right bounds for $\alpha(v)$ with respect to the ancestor nodes. Formally, $\beta(v) < \min\{\alpha(v') : v \text{ is an ancestor of } v'\}$, $\gamma(v) > \max\{\alpha(v') : v \text{ is an ancestor of } v'\}$. We can compute them as follows. If v is the root, then $\beta(v) = -\infty$, and $\gamma(v) = +\infty$. Here, $-\infty$ and $+\infty$ are constants that are a priori less and more than any $\alpha(v')$ for $v' \in \mathcal{V}(G)$. Let v be a non-root node. If v is the left child of $\textsc{Parent}(v)$, then $\beta(v) = \beta(\textsc{Parent}(v))$, $\gamma(v) = \alpha(\textsc{Parent}(v))$. If v is the right one, then $\beta(v) = \alpha(\textsc{Parent}(v))$, $\gamma(v) = \gamma(\textsc{Parent}(v))$. Assume that a comparing function for elements $\textsc{Compare}(a, b)$ returns -1 if $a < b$; $+1$ if $a > b$; 0 if $a = b$. An error probability for the function is $p < 0.5 - \delta$ for some $0 < \delta < 0.5$. Let us present each of the required procedures for searching an object x operation. $\textsc{GetTheRoot}(G)$ is for the root node of G. $\textsc{SelectChild}(v)$ returns the left child of v if $\textsc{Compare}(\alpha(v), x) = -1$; and returns the right child if $\textsc{Compare}(\alpha(v), x) = +1$. $\textsc{IsItTheTarget}(v)$ returns *True* iff $\mathit{Compare}(\alpha(v), x) = 0$. $\textsc{ProcessANode}(\text{V})$ does nothing. $\textsc{IsANodeCorrect}(\text{V})$ returns *True* iff $\beta(v) < x < \gamma(v)$, formally, $\textsc{Compare}(\beta(v), x) = -1$ and $\textsc{Compare}(x, \gamma(v)) = -1$. The presented operations satisfy all requirements. Let us present the complexity result that directly follows from Theorem 1.

Theorem 2. *Suppose the comparing function for elements of Red–Black Tree has an error $p < 0.5 - \delta$ for some $0 < \delta < 0.5$. Then, using the walking tree, we can carry out searching, adding and removing operations with $O(\log(n/\varepsilon))$ running time and an error probability ε.*

Remind that it is important for the proof to isolate p from 0.5. We cannot be sure that the algorithm works if $p \to 0.5$. If $\varepsilon = 1/Poly(n)$, then the "noisy" setting does not affect asymptotic complexity.

Corollary 1. *Suppose the comparing function for elements of the Red–Black Tree has an error $p < 0.5 - \delta$ for some $0 < \delta < 0.5$. Then, using the walking tree, we can carry out searching, adding, and removing operations with $O(\log n)$ running time and an error probability $1/Poly(n)$.*

4.2. Noisy Segment Tree

We consider a standard segment tree data structure [10,11] for an array $b = (b_1, \ldots, b_n)$ for some integer $n > 0$. The segment tree is a full binary tree such that each node corresponds to a segment of the array b. If a node v corresponds to a segment $(b_{left}, \ldots, b_{right})$, then we store a value $\alpha(v)$ that represents some information about the segment. Let us consider a function f such that $\alpha(v) = f(b_{left}, \ldots, b_{right})$. A segment of a node is the union of segments that correspond to its two children. Typically, the children correspond to segments $(b_{left}, \ldots, b_{mid})$ and $(b_{mid+1}, \ldots, b_{right})$, for $mid = \lfloor (left + right)/2 \rfloor$. We consider $\alpha(v)$ such that it can be computed by values of two children $\alpha(v_l)$ and $\alpha(v_r)$, where v_l and v_r are left and right children of v. Leaves correspond to single elements of the array b. As an example, we can consider integer values b_i and sum $\alpha(v) = b_{left} + \cdots + b_{right}$ as the value in a node v and a corresponding segment $(b_{left}, \ldots, b_{right})$. The data structure allows us to invoke the following requests in $O(\log n)$ running time.

- Update. Parameters are an index i and an element x ($1 \le i \le n$). The procedure assigns $b_i \leftarrow x$. For this goal, we assign x for the leaf w that corresponds to the b_i and update ancestors of w.
- Request. Parameters are two indexes i and j ($1 \le i \le j \le n$), the procedure computes $f(b_i, \ldots, b_j)$.

The main part of both operations is the following. For the given root node and an index i, we should find the leaf node corresponding to b_i. The main step is the following. If we are in a node v with associated segment $(b_{left}, \ldots, b_{right})$, then we compare i with a middle element $mid = \lfloor (left + right)/2 \rfloor$ and choose the left or the right child. Assume that we have a comparing function for indexes $\textsc{Compare}(a, b)$ that returns -1 if $a < b$; $+1$ if $a > b$; 0 if $a = b$. The comparing function returns the answer with an error $p < 0.5 - \delta$ for some $0 < \delta < 0.5$.

Let us present each of the required procedures for searching the leaf with index i in a segment tree G. $\textsc{GetTheRoot}(G)$ returns the root node of the segment tree G. For $mid = \lfloor (left + right)/2 \rfloor$, and the segment $(b_{left}, \ldots, b_{right})$ associated with a node v, the function $\textsc{SelectChild}(v)$ returns the left child of v if $Compare(mid, i) \le 0$; and returns the right child otherwise. $\textsc{IsItTheTarget}(v)$ returns *True* if $left = i = right$, formally, $Compare(left, i) = 0$ and $Compare(right, i) = 0$; and returns *False* otherwise. Here the segment $(b_{left}, \ldots, b_{right})$ is associated with v. $\textsc{ProcessANode}(v)$ recomputes $\alpha(v) = f(b_{left}, \ldots, b_{right})$ according to the values of α in the left and the right children. $\textsc{IsANodeCorrect}(v)$ returns *True* if $left \le i \le right$, formally, $\textsc{Compare}(left, i) \le 0$ and $\textsc{Compare}(x, right) \le 0$; and returns *False* otherwise. Here, the segment $(b_{left}, \ldots, b_{right})$ is associated with v. The presented operations satisfy all requirements. Let us present the complexity result that directly follows from Theorem 1.

Theorem 3. *Suppose, the comparing function for indexes of a segment tree is noisy and has an error $p < 0.5 - \delta$ for some $0 < \delta < 0.5$. Then, using the walking tree, we can conduct update and request operations with $O(\log(n/\varepsilon))$ running time and an error probability ε.*

If we take $\varepsilon = 1/Poly(n)$, then the "noisy" setting does not affect asymptotic complexity.

Corollary 2. *Suppose, the comparing function for indexes of a segment tree is noisy and has an error $p < 0.5 - \delta$ for some $0 < \delta < 0.5$. Then, using the walking tree, we can carry out update and request operations with $O(\log n)$ running time and an error probability $1/Poly(n)$*

We can apply the same ideas for the segment tree with range updates [11] (examples of applications can be found in [51,52]) or other modifications of the data structure. If we have access to the full segment tree, including leaves, then we can complete operations without the walking tree. We can use the noisy binary search algorithm [1] for searching the leaf that corresponds to the index i and then process all the ancestors of the leaf. At the same time, if we have only access to the root node or compressed case, then a noisy segment tree is more useful.

Theorem 3 and Corollary 2 are the direct application of the main technique and results from Theorem 1 to a specific data structure (segment tree). This is one of two applications that are presented in this paper.

Analysis, Discussion, Modifications for Noisy Segment Tree

There are different additional operations with a segment tree. One such example is the segment tree with range updates. In this modification, we can update the values b_i for a range $i \in \{\ell, \ldots, r\}$ by a value in one request. The reader can find more information in [11] and examples of applications in [51,52]. The main operation with a noisy comparing procedure is the same. So, we can still use the same idea for such modifications of the segment tree.

Remark 1. *If the segment tree is constructed for an array $b_1, \ldots, b_n$, then we can extend it to $b_1, \ldots, b_n, \ldots, b_k$, where $k = 2^{\lceil \log_2 n \rceil}$ that is closest to n power of 2 and $b_{n+1}, \ldots, b_k$ are neutral element for the function f. If we have a node v and two borders $left$ and $right$ of the segment associated with v, then we always can compute the segments for the left and the right children that are $b_{left}, \ldots, b_{mid}$ and $b_{mid+1}, \ldots, b_{right}$ for $mid = (left + right)/2$. Additionally, we can compute the segment for the parent that is $b_{left}, \ldots, b_{pright}$, where $pright = 2 \cdot right - left$ if the node is the left child of its parent. If the node is the right child of its parent, then the parent's segment is $b_{pleft}, \ldots, b_{right}$, where $pleft = 2 \cdot left - right$. Therefore, we should not store the borders of a segment in a node, and we can compute them during the walk on the segment tree. Additionally, we should not construct the walking tree. We can keep it in mind and walk by the segment tree itself using only three variables: the $left$ and right borders of the current segment and a counter if required.*

If we have access to the full segment tree, including leaves, then we can conduct operations without the walking tree. We can use the noisy binary search algorithm [1] for searching the leaf that corresponds to the index i, and then process all the ancestors of the leaf. There are at least two useful scenarios for a noisy segment tree.

1. We have access only to the root and have no direct access to leaves.

2. The second one is the compressed segment tree. If initially, all elements of the array b are empty or neutral for the function f, then we can compress a subtree with one node with a label of a segment with empty elements. On each step, we do not store any subtree if it has only neutral elements. In that case, we store only a root of this tree and mark it as a subtree with neutral elements. It is reasonable if n is very big and storing the whole tree is very expensive. In that case, we can replace the noisy binary tree with the noisy self-balanced search tree from the previous section. The search tree stores the updated elements in leaves, and we can search the required index in this data structure. At the same time, the noisy segment tree uses much less memory with respect to Remark 1. That is why a noisy segment tree is more effective in this case too.

5. Quantum Sort Algorithm for Strings

As one of the interesting applications, we suggest applications from quantum computing [12,29]. Let us discuss the string-sorting problem as one of the applications.

Problem: There are n strings $s^0, \ldots, s^{n-1}$ of size l for some positive integers n and l. The problem is to find a permutation $(j_0, \ldots, j_{n-1})$ such that $s^{j_i} < s^{j_{i+1}}$, or $s^{j_i} = s^{j_{i+1}}$ and $j_i < j_{i+1}$ for each $i \in \{0, \ldots, n-2\}$. The Quantum sorting algorithm for strings was presented in [35,36]. The running time of the algorithm is $O(n\sqrt{l}\log n)$. We can present the algorithm with the same complexity but in a simpler way. Assume that we have a noisy self-balanced binary search tree with strings as keys and a quantum string comparing procedure.

There is a quantum algorithm for comparing two strings quadratically faster than any classical counterparts. The algorithm is based on modifications [39,53] of Grover's search algorithm [44,45]. The result is the following.

Lemma 1 ([36]). *There is a quantum algorithm that compares two strings s and t of lengths $|s|$ and $|t|$ in the lexicographical order with $O(\sqrt{\min(|s|, |t|)})$ running time and error probability* 0.1.

We assume that the comparing procedure compares strings in lexicographical order, and if they are equal, then it compares indexes. In fact, we store indexes of the strings in nodes. We assume that if a node stores a key index i, then any node from the left subtree has a key index j such that $s^j < s^i$ or ($s^j = s^i$ and $j < i$); and any node from the right subtree has a key index j' such that $s^{j'} > s^i$ or ($s^j = s^i$ and $j' > i$). Initially, the tree is empty. Let $\textsc{Add}(i)$ be a function that adds a string s^i to the tree. Let GetMin be a function that returns the index of the minimal string s from the tree according to the comparing procedure. After returning the index, the function removes it from the tree. The algorithm is two for-loops. The first one is adding all strings one by one using $\textsc{Add}(i)$ for $i \in \{0, \ldots, n-1\}$. The second one is obtaining the index of minimal strings GetMin for $i \in \{0, \ldots, n-1\}$. The code representation of the algorithm is in Appendix A. The second For-loop can be replaced by in-order traversal (dfs) of the tree for constructing the list. This approach has a smaller hidden constant in big-O. The full idea is presented in Appendix A for completeness.

Theorem 4. *The quantum running time for sorting n string of length l is $O(n\sqrt{l}\log n)$ and $\Omega(n + \sqrt{nl})$.*

The upper bound is complexity of the presented algorithm and algorithm from [36]. The proof of the lower bound is presented below.

For simplicity, we assume that strings are binary, i.e., $s^i \in \{0,1\}^l$, for $i \in \{0, \ldots, n-1\}$. Let us formally define the sorting function.

For positive integers n, l, let $\text{RADIX}_{n,l} : \{0,1\}^{nl} \to \mathbb{S}_n$ be a function that obtains n binary strings of length l as input and returns a permutation of n integers that is a result of sorting input strings. Here, $\mathbb{S}_n$ is a set of all permutations of integers from 0 to $n-1$. For $s^0, \ldots, s^{n-1} \in \{0,1\}^l$, we have $\text{RADIX}_{n,l}(s^0, \ldots, s^{n-1}) = (j_0, \ldots, j_{n-1})$, where $(j_0, \ldots, j_{n-1}) \in \mathbb{S}_n$ is a permutation such that $\sigma_{j_i} < \sigma_{j_{i+1}}$ or ($\sigma_{j_i} = \sigma_{j_{i+1}}$ and $j_i < j_{i+1}$), for $i \in \{0, \ldots, n-2\}$.

Note that in the case of $l = 1$, the function $\text{RADIX}_{n,1}$ can be used to compute the majority function. We use $\text{RADIX}_{n,1}$ to sort strings and the $n/2$-th string is a value of the majority function. Therefore, we expect that complexity of $\text{RADIX}_{n,1}$ should be $\Omega(n)$ [54]. In the case of $n = 2$, the function $\text{RADIX}_{2,l}$ is similar to the OR function, so we expect that it requires $\Omega(\sqrt{l})$ queries [54].

Note that in the case of $l = 1$, the function $\text{RADIX}_{n,1}$ can be used to compute the majority function. We use $\text{RADIX}_{n,1}$ to sort strings and the $n/2$-th string is a value of the majority function. Therefore, we expect that complexity of $\text{RADIX}_{n,1}$ should be $\Omega(n)$. In

the case of $n = 2$, the function $\mathrm{RADIX}_{2,l}$ is similar to the OR function, so we expect that it requires $\Omega(\sqrt{l})$ queries.

Formally, we prove the following.

Lemma 2. *For positive integers n, l, let $\mathrm{MAJ}_n : \{0,1\}^n \to \{0,1\}$ be a majority function, and $\mathrm{SEARCH1}_l : \{0,1\}^l \to \{1,\ldots,l\}$ be a function that returns the minimal index of one in the input.*

$$\mathrm{Q}(\mathrm{RADIX}_{n,1}) \geq \mathrm{Q}(\mathrm{MAJ}_n) \tag{2}$$

$$\mathrm{Q}(\mathrm{RADIX}_{2,l}) \geq \mathrm{Q}(\mathrm{OR}_l) \tag{3}$$

Proof. Consider the input $x \in \{0,1\}^n$. Suppose that $\mathrm{RADIX}_{n,1}(x) = \sigma = (\sigma_1, \ldots, \sigma_n)$.

Then the proof of (2) follows from the fact that

$$\mathrm{MAJ}_n(x) = x_{\sigma_{\lceil n/2 \rceil}}.$$

Take an input $y \in \{0,1\}^l$. Let $y' \in \{0,1\}^{2l}$ be a pair of words $\vec{0} = 0 \ldots 0 \in \{0,1\}^l$ and y. Now we see that

$$\mathrm{OR}_l(y) = \begin{cases} 1, & \text{if } \mathrm{RADIX}_{2,l}(y') = (10), \\ 0, & \text{if } \mathrm{RADIX}_{2,l}(y') = (01). \end{cases}$$

and this completes the proof. □

Note that [36] proves the lower bound of the form $\mathrm{Q}(\mathrm{RADIX}_{n,l}) \geq \Omega(\sqrt{nl})$. Combining their result with the Lemma 2, we get the following corollary.

Corollary 3. *The complexity of $\mathrm{RADIX}_{n,l}$ is $\mathrm{Q}(\mathrm{RADIX}_{n,l}) \geq \Omega(n + \sqrt{nl})$.*

6. Auto-Complete Problem

In this section, we present the Auto-Complete Problem and a quantum algorithm that is another application of the Noisy Binary Search Tree.

Problem: Assume that we use some constant-size alphabet, for example, binary, ASCII, or Unicode. We work with a sequence of strings $\mathcal{S} = (s^{i_1}, \ldots, s^{i_{|\mathcal{S}|}})$, where $|\mathcal{S}|$ is the length of the sequence; and i_j are increasing indexes of strings. Here, the index i_j is the index of the query for adding this string to $\mathcal{S}$. Initially, the sequence $\mathcal{S}$ is empty. Then, we have n queries of two types. The first type is adding a string s to the sequence $\mathcal{S}$. Let $\#(u) = |\{j : u = s^j, j \in \{i_1, \ldots, i_{|\mathcal{S}|}\}\}|$ be a number of occurrence (or "frequency") of a string u among $\mathcal{S}$. The second type is querying the most frequent completion from $\mathcal{S}$ of a string t. Let us define it formally. If t is a prefix of s^j, then we say that s^j is a completion of t. Let $D(t) = \{s^j : j \in \{i_1, \ldots, i_{|\mathcal{S}|}\}, s^j \text{ is a completion of } t\}$ be the set of completions for t, and let $md(t) = max\{\#(s^r) : r \in \{i_1, \ldots, i_{|\mathcal{S}|}\}, s^r \in D(t)\}$ be the maximal "frequency" of strings from $D(t)$. The problem is to find the index $mi(t) = min\{r : r \in \{i_1, \ldots, i_{|\mathcal{S}|}\}, s^r \in D(t), \#(s^r) = md(t)\}$.

We use a Self-Balanced Search tree for our solution. A node v of the tree stores 4-tuple (i, c, j, jc), where i is an index of a string s^i that is "stored" in the node, and $c = \#(s^i)$. The tree is a Search tree by this strings s^i similar to storing strings in Section 5. For comparing strings, we use a quantum procedure from Lemma 1. Therefore, our tree is noisy. The index j is the index of the most "frequent" string in the sub-tree whose root is v, and $jc = \#(s^j)$. Formally, for any node v' from this sub-tree if (i', c', j', jc') is associated with v', then $c' < jc$ or $(c' = jc$ and $i' \geq j)$.

Initially, the tree is empty. Let us discuss processing the first type of query. We want to add a string s to $\mathcal{S}$. We search a node v with associated (i, c, j, jc) such that $s^i = s$. If we can find it, then we increase $c \leftarrow c + 1$. It means j parameter of the node v or its ancestors can be updated. There are at most $O(\log n)$ ancestors because the height of the tree is $O(\log n)$.

So, for each ancestor of v that associated with (i', c', j', jc'), if $jc' < c$ or ($jc' = c$ and $j' > i$), then we update $j' \leftarrow i$ and $jc \leftarrow c$.

If we cannot find the string s in $\mathcal{S}$, then we add a new node to the tree with associated 4-tuple $(r, 1, r, 1)$, where r is the index of the query. Note that if we re-balance nodes in the Red–Black tree, then we easily can recompute j and jc elements of nodes.

Assume that we have a SEARCH(s) procedure that returns a node v with associated (i, c, j, jc) where $s = s^i$. If there is no such a node, then the procedure returns *NULL*. A procedure ADDASTRING(r) adding a node $(r, 1, r, 1)$ to the tree. A procedure GETTHEROOT returns the root of the search tree. The processing of the first type of query is presented in Algorithm 4.

Algorithm 4 Processing a query of the first type with an argument s and a query number r

```
v ← SEARCH(s)
if v ≠ NULL then
    (i, c, j, jc) is associated with v
    c ← c + 1
    if jc < c or (jc = c and j ≥ i) then
        jc ← c
        j ← i
    end if
    while v ≠ GETTHEROOT do
        v ← PARENT(v)
        (i', c', j', jc') is associated with v
        if jc' < c or (jc' = c and j' ≥ i) then
            jc' ← c
            j' ← i
        end if
    end while
else
    ADDASTRING(r)
end if
```

Let us discuss processing the second type of query. All strings s^i that can be a completion for t belong to the set $C(t) = \{s^r : r \in \{i_1, \ldots, i_{|\mathcal{S}|}\}, s^r \geq t \text{ and } s^r < t'\}$. Here, we can obtain t' from the string t by replacing the last symbol with the next symbol from the alphabet. Formally, if $t = (t_1, \ldots, t_{|t|-1}, t_{|t|})$, then $t' = (t_1, \ldots, t_{|t|-1}, t'_{|t|})$, where the symbol $t'_{|t|}$ succeeds $t_{|t|}$ in the alphabet. We can say that $C(t) = D(t)$. The query processing consists of three steps.

Step 1. We search for a node v_{Rt} such that t should be in the left sub-tree of v and t' should be in the right sub-tree of v. Formally, $\beta(v_{Rt}) \leq t \leq \alpha(v_{Rt})$, and $\alpha(v_{Rt}) < t' \leq \gamma(v_{Rt})$. For implementing this idea the procedure ISITTHETARGET(v) checks the following condition:

$$\big(\text{COMPARE}(\beta(v), t) \leq 0 \text{ and } \text{COMPARE}(t, \alpha(v)) \leq 0\big)$$

and

$$\big(\text{COMPARE}(\alpha(v), t') < 0 \text{ and } \text{COMPARE}(t', \gamma(v)) \leq 0\big).$$

The procedure SELECTCHILD(v) returns the right child if $t > \alpha(v)$, i.e.,

$$\text{COMPARE}(t, \alpha(v)) > 0,$$

and returns the left child otherwise.

If we come to the null node, then there are no completions of t. If we find v_{Rt}, then we carry out the next two steps. In those two steps, we compute the index of the required string j_{ans} and $jc_{ans} = \#(s^{j_{ans}})$.

Step 2. Let us look to the left sub-tree with t. Let us find a node v_L that contains an index i_L of the minimal string $s^{i_L} \geq t$. For implementing this idea, the procedure IsItTheTarget(v) checks whether the current string is t. Formally, it checks Compare$(\alpha(v), t) = 0$. Additionally, the procedure saves the node $v_L \leftarrow v$ if $\alpha(v) \geq t$, i.e., Compare$(t, \alpha(v)) \geq 0$. The procedure SelectChild(v) works as for searching t. It returns the right child if $t > \alpha(v)$, i.e., Compare$(t, \alpha(v)) > 0$, and returns the left child otherwise. In the end, the target node is stored in v_L.

Then, we go up from this node. Let us consider a node v. If it is the right child of Parent(v), then it is bigger than the string from Parent(v) and the left child's sub-tree, so we do nothing. If it is the left child of Parent(v), then it is less than the string from Parent(v) and all strings from the right child's sub-tree, so we update j_{ans} and jc_{ans} by values from the parent node and the right child. Formally, if (i_p, c_p, j_p, jc_p) for the Parent(v) node and (i_r, c_r, j_r, jc_r) for the right child node, then we complete the following actions. If $c_p > jc_{ans}$ or ($c_p = jc_{ans}$ and $i_p < j_{ans}$), then $j_{ans} \leftarrow i_p$ and $jc_{ans} \leftarrow c_p$. If $jc_r > jc_{ans}$ or ($jc_r = jc_{ans}$ and $j_r < j_{ans}$), then $j_{ans} \leftarrow j_r$ and $jc_{ans} \leftarrow jc_r$. This idea is presented in Algorithm 5.

Algorithm 5 Obtaining the answer of Step 2 by v_L

```
v ← v_L
while v ≠ Parent(v_Rt) do
    parent ← Parent(v)
    if v = LeftChild(parent) then
        (i_p, c_p, j_p, jc_p) is associated with parent
        if c_p > jc_ans or (c_p = jc_ans and i_p < j_ans) then
            j_ans ← i_p
            jc_ans ← c_p
        end if
        (i_r, c_r, j_r, jc_r) is associated with RightChild(parent)
        if RightChild(parent) ≠ NULL then
            if (c_r > jc_ans or (c_r = jc_ans and i_r < j_ans)) then
                j_ans ← i_r
                jc_ans ← c_r
            end if
        end if
    end if
    v ← parent
end while
```

Step 3. Let us look to the right sub-tree with t'. Let us find a node v_R that contains an index i_R of the maximal string $s^{i_R} < t'$. Then, we go up from this node and carry out the symmetric actions as in Step 2.

Each of these three steps requires $O(\log n)$ running time because each of them observes nodes of a single branch. The complexity of the quantum algorithm is presented in Theorem 5. Before presenting the theorem, let us present a lemma that helps us to prove quantum and classical lower bounds.

Lemma 3. *Auto-complete problem at least as hard as unstructured search* 1 *among* $O(L)$ *bits.*

Proof. Assume that the alphabet is binary. For any other case, we just consider two letters from the alphabet. Assume that all strings from queries of the first type have length k.

Let $m = \lfloor (n-1)/2 \rfloor$. Let us consider the following case. We have m queries of the first type of the next form: $s^i = (0, 0, s^i_3, \ldots, s^i_k)$, Here, $s^i_j = 0$ for all $i \in \{1, \ldots, m\}$ and $j \in \{3, \ldots, k\}$ except one bit. We have two cases.

The first case is the following. There is only one pair (r, w) such that $r \in \{1, \ldots, m\}$, $w \in \{3, \ldots, k\}$, and $s^r_w = 1$. In the second case, there is no such a pair.

Next m queries of the first type of the next form $(0, 1, 0, \ldots, 0)$.

If $n-1$ is odd, then we add a query of the first type of the next form $(1, \ldots, 1)$.

The last query of the second type of the form $t = (0)$.

If we have the first case, then $\#(0, 1, 0, \ldots, 0) = m$, $\#(0, 0, 0, \ldots, 0) = m - 1$ and the answer is $(0, 1, 0, \ldots, 0)$. If we have the second case, then $\#(0, 1, 0, \ldots, 0) = m$, $\#(0, 0, 0, \ldots, 0) = m$ and $(0, 0, 0, \ldots, 0)$ has a smaller index. Therefore, the answer is $(0, 0, 0, \ldots, 0)$.

Hence, the answer for the input is at least as hard as distinguishing between these two cases. At the same time, it requires searching 1 among $O((k-3)m) = O(L)$ bits. □

Based on the presented lemma and the discussed algorithm, we can present quantum lower and upper bounds for the problem in the next theorem.

Theorem 5. *The quantum algorithm with a noisy Self-Balanced Search tree for the Auto-Complete Problem has $O(\sqrt{nL}\log n)$ running time and error probability $1/3$, where L is the sum of lengths of all queries. Additionally, the lower bound for quantum running time is $\Omega(\sqrt{L})$.*

Proof. Let us start with the upper bound. Let us consider the processing of the first type of query. Here we add a string s. Firstly, we find the target node with $O(\sqrt{|s|}\log n)$ running time and $1/n^2$ error probability according to Corollary 1. Then, we consider at most $O(\log n)$ ancestors for updating with $O(\log n)$ running time and no error. So, processing the first type of query works with $O(\sqrt{|s|}\log n)$ running time and $1/n^2$ error probability.

Let us consider the processing of the second type of query. Here we search for a completion for a string t. Searching nodes v_{Rt}, v_L and v_R works with $O(\sqrt{|t|}\log n)$ running time and $1/n^2$ error probability according to Corollary 1. Then, we consider at most $O(\log n)$ ancestors for updating with $O(\log n)$ running time and no error. So, processing the second type of query works with $O(\sqrt{|t|}\log n)$ running time and $3/n^2$ error probability.

Let $m_1, \ldots, m_n$ be the lengths of strings from queries. So, the total complexity is

$$O(\sqrt{m_1}\log n + \cdots + \sqrt{m_n}\log n) = O((\sqrt{m_1} + \cdots + \sqrt{m_n})\log n) =$$

$$O(\sqrt{n(m_1 + \cdots + m_n)}\log n = O(\sqrt{nL}\log n)$$

The last two equalities are due to Cauchy–Bunyakovsky–Schwarz inequality and $L = m_1 + \cdots + m_n$.

The error probability of processing a query is at most $3/n^2$. Therefore, the error probability of processing n queries is at most 0.1 due to all error events are independent.

Let us discuss the lower bound. It is known [55] that the quantum running time for the unstructured search among $O(L)$ variables is $\Omega(\sqrt{L})$. So, due to Lemma 3 we obtain the required lower bound. □

Let us consider the classical (deterministic or randomized) case. If we use the same Self-balanced search tree, then the running time is $O(L\log n)$. At the same time, if we use the Trie (prefix tree) data structure [40], then the complexity is $O(L)$.

We store all strings of $\mathcal{S}$ in the trie. For each terminal node, we store the "frequency" of the corresponding string. For each node v (even non-terminal) that corresponds to a string u as a path from the root to v, additionally to the regular information we store the index of the required completion for t and its frequency. When we process the first type of query, we update the frequency in the terminal node and update additional information in all ancestor nodes because they store all possible prefixes of the string. For processing the query of the second type, we just find the required node and take the answer in the additional information of the node.

We can show that it is also the lower bound for the classical case.

Lemma 4. *The classical running time for the Auto-Complete Problem is $\Theta(L)$, where L is the sum of the lengths of all queries.*

Proof. Let us start with the upper bound. Similarly to the proof of Theorem 5, we can show that the running time of processing a query of both types is $O(|s|)$ or $O(|t|)$ depends on the type of a query. Let $m_1, \ldots, m_n$ be the lengths of strings from queries. So, the total complexity is $O(m_1 + \cdots + m_n) = O(L)$.

Let us discuss the lower bound. It is known [55] that the classical running time for the unstructured search among $O(L)$ variables is $\Omega(L)$. So, due to Lemma 3 we obtain the required lower bound. □

If $O(n)$ strings have a length that is at least $\Omega((\log_2 n)^2)$, then we obtain a quantum speed-up.

7. Conclusions

We suggest the Walking tree technique for noisy tree data structures. We apply the technique to the Red–Black tree and the Segment tree. We show that the complexity of the main operations is asymptotically equal to the complexity of standard (not noisy) versions of the data structures. We use a noisy Red–Black tree for two problems: The string-sorting Problem and The auto-complete problem. The considered algorithms are quantum because they use the quantum string comparing algorithm as a subroutine. This subroutine demonstrates quadratic speed-up, but it has a non-zero error probability. For the string-sorting problem, we show lower and upper bounds that are the same up to a log factor. Note that lower bounds for FST_m, SG_n, and $SGN_{n,l}$ functions may be of independent interest. For the auto-complete problem, we obtain quantum speed-up for the problems if $(\log n)^2 = o(l)$ where n is the number of queries and l is the size of an input string of a query. Future work can include applying the Walking tree technique to other tree data structures and obtaining the noisy version of them with good complexity of main operations. Also, it is interesting to find more applications for noisy data structures. We assume that quantum algorithms should be one of the fruitful fields for such applications. It is interesting to meet quantum lower and upper bounds for the considered problems.

Author Contributions: The main idea of the technique, K.K.; technical proofs for the main technique, N.S.; lower bounds, M.Z.; applications, K.K. and D.M.; constructions and concepts, K.K. and M.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This paper has been supported by the Kazan Federal University Strategic Academic Leadership Program (PRIORITY-2030).

Data Availability Statement: There is no data for the research.

Acknowledgments: We thank Aliya Khadieva for useful discussions.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Appendix A. Quantum Sorting Algorithm—The Second Approach

Assume that we want to construct a result list. We use *list* variable as a result.

We use the recursive procedure GetListByTree(v) for in-order traversal (dfs) of the searching tree. Here, v is the processing node. Assume that we have GetTheLeftChild(v) for obtaining the left child of v, GetTheRightChild(v) for obtaining the right child of v; GetIndexOfString(v) for obtaining the index of the string $\alpha(v)$ that is stored in v. The procedure is presented in Algorithm A1.

Algorithm A1 The recursive procedure GETLISTBYTREE(v) for in-order traversal (dfs) of the searching tree

if $v \neq NULL$ **then**
  GETLISTBYTREE(GETTHELEFTCHILD(v))
  $list \leftarrow list \circ$ GETINDEXOFSTRING(v)
  GETLISTBYTREE(GETTHERIGHTCHILD(v))
end if

The total sorting algorithm is Algorithm A2.

Algorithm A2 Quantum string-sorting algorithm

for $i \in \{0, \ldots, n-1\}$ **do**
  ADD(i)
end for
$list \leftarrow []$ ▷ Initially, the list is empty
GETLISTBYTREE(GETTHEROOT())
for $i \in \{0, \ldots, n-1\}$ **do**
  $j_i \leftarrow list[i]$
end for

References

1. Feige, U.; Raghavan, P.; Peleg, D.; Upfal, E. Computing with noisy information. *SIAM J. Comput.* **1994**, *23*, 1001–1018. [CrossRef]
2. Pelc, A. Searching with known error probability. *Theor. Comput. Sci.* **1989**, *63*, 185–202. [CrossRef]
3. Karp, R.M.; Kleinberg, R. Noisy binary search and its applications. In Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Philadelphia, PA, USA, 7–9 January 2007; pp. 881–890.
4. Emamjomeh-Zadeh, E.; Kempe, D.; Singhal, V. Deterministic and probabilistic binary search in graphs. In Proceedings of the Forty-Eighth Annual ACM Symposium on Theory of Computing, Cambridge, MA, USA, 19–21 June 2016; pp. 519–532.
5. Dereniowski, D.; Łukasiewicz, A.; Uznański, P. An efficient noisy binary search in graphs via median approximation. In Proceedings of the 32nd International Workshop on Combinatorial Algorithms, Ottawa, ON, Canada, 5–7 July 2021; pp. 265–281.
6. Deligkas, A.; Mertzios, G.B.; Spirakis, P.G. Binary search in graphs revisited. *Algorithmica* **2019**, *81*, 1757–1780. [CrossRef]
7. Boczkowski, L.; Korman, A.; Rodeh, Y. Searching on trees with noisy memory. *arXiv* **2016**, arXiv:1611.01403.
8. Dereniowski, D.; Kosowski, A.; Uznanski, P.; Zou, M. Approximation Strategies for Generalized Binary Search in Weighted Trees. In Proceedings of the 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017), Warsaw, Poland, 10–14 July 2017.
9. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*; McGraw-Hill: New York, NY, USA, 2001.
10. Mark, D.B.; Otfried, C.; Marc, V.K.; Mark, O. *Computational Geometry Algorithms and Applications*; Spinger: Berlin/Heidelberg, Germany, 2008.
11. Laaksonen, A. *Guide to Competitive Programming*; Springer: Berlin/Heidelberg, Germany, 2017.
12. Nielsen, M.A.; Chuang, I.L. *Quantum Computation and Quantum Information*; Cambridge University Press: Cambridge, UK, 2010.
13. de Wolf, R. *Quantum Computing and Communication Complexity*; University of Amsterdam: Amsterdam, The Netherlands, 2001.
14. Jordan, S. Quantum Algorithms Zoo. 2021. Available online: http://quantumalgorithmzoo.org/ (accessed on 20 September 2023).
15. Dürr, C.; Heiligman, M.; Høyer, P.; Mhalla, M. Quantum query complexity of some graph problems. *SIAM J. Comput.* **2006**, *35*, 1310–1328. [CrossRef]
16. Khadiev, K.; Safina, L. Quantum Algorithm for Dynamic Programming Approach for DAGs. Applications for Zhegalkin Polynomial Evaluation and Some Problems on DAGs. In Proceedings of the International Conference on Unconventional Computation and Natural Computation, Tokyo, Japan, 3–7 June 2019; Volume 4362, pp. 150–163.
17. Khadiev, K.; Kravchenko, D.; Serov, D. On the Quantum and Classical Complexity of Solving Subtraction Games. In Proceedings of the 14th International Computer Science Symposium in Russia, Novosibirsk, Russia, 1–5 July 2019; Volume 11532, pp. 228–236.
18. Khadiev, K.; Safina, L. Quantum Algorithm for Dynamic Programming Approach for DAGs and Applications. *Lobachevskii J. Math.* **2023**, *44*, 699–712. [CrossRef]
19. Lin, C.Y.Y.; Lin, H.H. Upper Bounds on Quantum Query Complexity Inspired by the Elitzur-Vaidman Bomb Tester. In Proceedings of the 30th Conference on Computational Complexity (CCC 2015), Portland, OR, USA, 17–19 June 2015.
20. Lin, C.Y.Y.; Lin, H.H. Upper Bounds on Quantum Query Complexity Inspired by the Elitzur–Vaidman Bomb Tester. *Theory Comput.* **2016**, *12*, 537–566. [CrossRef]
21. Beigi, S.; Taghavi, L. Quantum speedup based on classical decision trees. *Quantum* **2020**, *4*, 241. [CrossRef]
22. Ramesh, H.; Vinay, V. String matching in $O(\sqrt{n} + \sqrt{m})$ quantum time. *J. Discret. Algorithms* **2003**, *1*, 103–110. [CrossRef]
23. Montanaro, A. Quantum pattern matching fast on average. *Algorithmica* **2017**, *77*, 16–39. [CrossRef]

24. Le Gall, F.; Seddighin, S. Quantum Meets Fine-Grained Complexity: Sublinear Time Quantum Algorithms for String Problems. In Proceedings of the 13th Innovations in Theoretical Computer Science Conference (ITCS 2022), Berkeley, CA, USA, 31 January–3 February 2022.
25. Le Gall, F.; Seddighin, S. Quantum meets fine-grained complexity: Sublinear time quantum algorithms for string problems. *Algorithmica* **2023**, *85*, 1251–1286. [CrossRef]
26. Akmal, S.; Jin, C. Near-optimal quantum algorithms for string problems. In Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Alexandria, VA, USA, 9–12 January 2022; pp. 2791–2832.
27. Charalampopoulos, P.; Pissis, S.P.; Radoszewski, J. Longest Palindromic Substring in Sublinear Time. In Proceedings of the 33rd Annual Symposium on Combinatorial Pattern Matching (CPM), Prague, Czech Republic, 27–29 June 2022.
28. Ablayev, F.; Ablayev, M.; Salikhova, N. Hybrid classical-quantum text search based on hashing. *arXiv* **2023**, arXiv:2311.01213.
29. Ambainis, A. Understanding Quantum Algorithms via Query Complexity. *Proc. Int. Conf. Math.* **2018**, *4*, 3283–3304.
30. Høyer, P.; Neerbek, J.; Shi, Y. Quantum complexities of ordered searching, sorting, and element distinctness. In Proceedings of the International Colloquium on Automata, Languages, and Programming, Crete, Greece, 8–12 July 2001; pp. 346–357.
31. Høyer, P.; Neerbek, J.; Shi, Y. Quantum complexities of ordered searching, sorting, and element distinctness. *Algorithmica* **2002**, *34*, 429–448.
32. Odeh, A.; Elleithy, K.; Almasri, M.; Alajlan, A. Sorting N elements using quantum entanglement sets. In Proceedings of the Third International Conference on Innovative Computing Technology, London, UK, 29–31 August 2013; pp. 213–216.
33. Odeh, A.; Abdelfattah, E. Quantum sort algorithm based on entanglement qubits {00, 11}. In Proceedings of the 2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT), Farmingdale, NY, USA, 29 April 2016; pp. 1–5.
34. Klauck, H. Quantum time-space tradeoffs for sorting. In Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing, San Diego, CA, USA, 9–11 June 2003; pp. 69–76.
35. Khadiev, K.; Ilikaev, A. Quantum Algorithms for the Most Frequently String Search, Intersection of Two String Sequences and Sorting of Strings Problems. In Proceedings of the International Conference on Theory and Practice of Natural Computing, Kingston, ON, Canada, 9–11 December 2018; pp. 234–245.
36. Khadiev, K.; Ilikaev, A.; Vihrovs, J. Quantum Algorithms for Some Strings Problems Based on Quantum String Comparator. *Mathematics* **2022**, *10*, 377. [CrossRef]
37. Babu, H.M.H.; Jamal, L.; Dibbo, S.V.; Biswas, A.K. Area and delay efficient design of a quantum bit string comparator. In Proceedings of the 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Bochum, Germany, 3–5 July 2017; pp. 51–56.
38. Aborot, J.A. An Oracle Design for Grover's Quantum Search Algorithm for Solving the Exact String Matching Problem. Theory and Practice of Computation. In Proceedings of the Workshop on Computation: Theory and Practice WCTP2017, Osaka, Japan, 12–13 September 2019; pp. 36–48.
39. Kapralov, R.; Khadiev, K.; Mokut, J.; Shen, Y.; Yagafarov, M. Fast Classical and Quantum Algorithms for Online k-server Problem on Trees. *CEUR Workshop Proc.* **2022**, *3072*, 287–301.
40. Knuth, D. *The Art of Computer Programming*; Sorting and Searching; Pearson Education: London, UK, 1973; Volume 3.
41. De La Briandais, R. File searching using variable length keys. In Proceedings of the Western Joint Computer Conference, San Francisco, CA, USA, 3–5 March 1959; pp. 295–298.
42. Black, P.E. *Dictionary of Algorithms and Data Structures*; Technical Report; National Institute of Standards and Technology: Gaithersburg, MD, USA, 1998.
43. Brass, P. *Advanced Data Structures*; Cambridge University Press: Cambridge, UK, 2008; Volume 193.
44. Grover, L.K. A fast quantum mechanical algorithm for database search. In Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing, Philadelphia, PA, USA, 22–24 May 1996; pp. 212–219.
45. Boyer, M.; Brassard, G.; Høyer, P.; Tapp, A. Tight bounds on quantum searching. *Fortschritte Phys.* **1998**, *46*, 493–505. [CrossRef]
46. Arunachalam, S.; de Wolf, R. Optimizing the Number of Gates in Quantum Search. *Quantum Inf. Comput.* **2017**, *17*, 251–261. [CrossRef]
47. Grover, L.K. Trade-offs in the quantum search algorithm. *Phys. Rev. A* **2002**, *66*, 052314. [CrossRef]
48. Long, G.L. Grover algorithm with zero theoretical failure rate. *Phys. Rev. A* **2001**, *64*, 022307. [CrossRef]
49. Motwani, R.; Raghavan, P. *Randomized Algorithms*; Chapman & Hall/CRC: Boca Raton, FL, USA, 2010.
50. Guibas, L.J.; Sedgewick, R. A dichromatic framework for balanced trees. In Proceedings of the 19th Annual Symposium on Foundations of Computer Science, Washington, DC, USA, 16–18 October 1978; pp. 8–21.
51. Khadiev, K.; Remidovskii, V. Classical and quantum algorithms for constructing text from dictionary problem. *Nat. Comput.* **2021**, *20*, 713–724. [CrossRef]
52. Khadiev, K.; Remidovskii, V. Classical and Quantum Algorithms for Assembling a Text from a Dictionary. *Nonlinear Phenom. Complex Syst.* **2021**, *24*, 207–221. [CrossRef]
53. Kothari, R. An optimal quantum algorithm for the oracle identification problem. In Proceedings of the 31st International Symposium on Theoretical Aspects of Computer Science, Lyon, France, 5–8 March 2014; p. 482.

54. Beals, R.; Buhrman, H.; Cleve, R.; Mosca, M.; de Wolf, R. Quantum Lower Bounds by Polynomials. In Proceedings of the 39th Annual Symposium on Foundations of Computer Science, Palo Alto, CA, USA, 8–11 November 1998; pp. 352–361.
55. Bennett, C.H.; Bernstein, E.; Brassard, G.; Vazirani, U. Strengths and weaknesses of quantum computing. *SIAM J. Comput.* **1997**, *26*, 1510–1523. [CrossRef]