A Fast Algorithm for Online k-servers Problem on Trees

Kamil Khadiev and Maxim Yagafarov
Kazan Federal University, Kazan, Russia
kamilhadi@gmail.com

Abstract. We consider online algorithms for the k-servers problem on trees. There is a k-competitive algorithm for this problem, and it is the best competitive ratio. M. Chrobak and L. Larmore suggested it. At the same time, the existing implementation has $O(n)$ time complexity, where $n$ is a number of nodes in a tree. We suggest a new time-efficient implementation of the algorithm. It has $O(n)$ time complexity for preprocessing and $O(k(\log n)^2)$ for processing a query.

1 Introduction

One of the applications for online algorithms is optimization problems [15]. The peculiarity is the following. An algorithm reads an input piece by piece and returns an answer piece by piece immediately, even if an answer can depend on future pieces of the input. The algorithm should return an answer for minimizing an objective function (the cost of an output). The most standard method to define the effectiveness is the competitive ratio [18,13].

Typically, online algorithms have unlimited computational power. At the same time, many different papers consider online algorithms with different restrictions. Some of them are restrictions on memory of an online algorithm [3,10,6,14,1,2,12], another ones are restrictions on time complexity [9,17].

In this paper, we focus on efficient online algorithms in terms of time complexity. We consider the k-servers problem on trees [7]. There is an k-competitive algorithm for this problem, and it is the best competitive ratio. At the same time, the existing implementation has $O(n)$ time complexity, where $n$ is a number of nodes in a tree. There is a time-efficient algorithm for general graphs [17] that uses min-cost-max-flow algorithms, but it is too slow in the case of a tree.

We suggest a new time-efficient implementation of the algorithm from [7]. It has $O(n)$ time complexity for preprocessing and $O(k(\log n)^2)$ for processing a query. It is based on data structures and techniques like a segment tree [16], heavy-light decomposition (heavy path decomposition) [19,11] for a tree and fast algorithms for computing Lowest Common Ancestor (LCA) [5,4].

The structure of the paper is following. Section 2 contains preliminaries. Tools are described in Section 3. The main algorithm is situated in Section 4.
2 Preliminaries

2.1 Online algorithms

An online minimization problem consists of a set $I$ of inputs and a cost function. Each input $I = (x_1, \ldots, x_n)$ is a sequence of requests, where $n$ is a length of the input $|I| = n$. Furthermore, a set of feasible outputs (or solutions) $O(I)$ is associated with each $I$; an output is a sequence of answers $O = (y_1, \ldots, y_n)$. The cost function assigns a positive real value $\text{cost}(I, O)$ to $I \in I$ and $O \in O(I)$. An optimal solution for $I \in I$ is $O_{\text{opt}}(I) = \arg\min_{O \in O(I)} \text{cost}(I, O)$. Let us define an online algorithm for this problem. A deterministic online algorithm $A$ computes the output sequence $A(I) = (y_1, \ldots, y_n)$ such that $y_i$ is computed by $x_1, \ldots, x_i$. We say that $A$ is $c$-competitive if there exists a constant $\alpha \geq 0$ such that, for every $n$ and for any input $I$ of size $n$, we have: $\text{cost}(I, A(I)) \leq c \cdot \text{cost}(I, O_{\text{opt}}(I)) + \alpha$, where $c$ is the minimal number that satisfies the inequality. Also we call $c$ the competitive ratio of $A$.

2.2 Graph Theory

Let us consider a root tree $G = (V, E)$, where $V$ is a set of nodes (vertexes), and $E$ is a set of edges. Let $n = |V|$ be a number of nodes or the size of the tree. Let the root of the tree be 1-st node.

The path $P$ is a sequence of nodes $(v_1, \ldots, v_h)$ that are connected by edges, i.e. $(v_i, v_{i+1}) \in E$ for all $i \in \{1, \ldots, h - 1\}$. Note, that there are no duplicates among $v_1, \ldots, v_h$. Here $h$ is a length of the path. The distance $\text{dist}(v, u)$ between two nodes $v$ and $u$ is a length of the path between them.

For each node $v$ we can define a parent node $\text{Parent}(v)$, it is such that $\text{dist}(1, \text{Parent}(v)) + 1 = \text{dist}(1, v)$. Additionally, we can define the set of children $\text{Children}(v) = \{u : \text{Parent}(u) = v\}$.

2.3 $k$-servers Problem

We have a root tree $G = (V, E)$. We are also given $k$ servers that can move among nodes of $G$. At each time slot, a request $q \in V$ appears, and we have to “serve” this request, that is, choose one of our servers and move it to $q$. Other servers are also allowed to move. Our measure of cost is the distance by which we move our servers. In other words, if before the request the positions of servers are $v_1, \ldots, v_k$ and after the request they are $v'_1, \ldots, v'_k$, then $q \in \{v'_1, \ldots, v'_k\}$ and the cost of the move is $\sum_{i=1}^{k} \text{dist}(v_i, v'_i)$.

The problem is to design a strategy that minimizes the cost of servicing a sequence of requests given online.

3 Tools

In the paper, we use two main tools. The first one is a segment tree with range updates. The good book for the data structure is [16]. The second one is heavy-
light decomposition (heavy path decomposition) [19] for a tree. Let us describe the main properties of both of them.

3.1 Segment Tree with Range Updates for Coloring Problem

Let us describe the coloring problem and solution using a segment tree data structure. The problem is used as a tool for the main algorithm.

**Coloring problem.** Assume that we have a sequence of \( d \) elements \( 1, \ldots, d \).

We associate a color \( c_i \) with element \( i \), where \( 1 \leq c_i \leq Z \) for some positive integer \( Z \). We should satisfy several queries. Each query can be one of two types:

- **Update.** For three integers \( l, r, c \) \((1 \leq l \leq r \leq d)\), we should color all elements of segment \([l, r]\) by \( c \), i.e. \( c_i = c \) for \( l \leq i \leq r \).
- **Request.** For an integer \( x \) \((1 \leq x \leq d)\), we should return \( c_x \).
- **Request Closest Colored.** For two integers \( l, r \) \((1 \leq l \leq r \leq d)\), we should return the minimal and maximal indexes of colored elements.

Firstly, let us describe the segment tree data structure. It is the full binary tree of height \( h \) such that \( 2^{h-1} < d \leq 2^h \). The data structure works with the sequence of elements of length \( 2^h \), but we are care only about the first \( d \) elements.

Each node of the tree is associated with some segment \([a, b]\) such that \( 1 \leq a \leq b \leq 2^h \). Each leaf is associated with elements of the sequence or we can say that it is associated with a segment of size 1. \( i \)-th node of the last level is associated with a segment \([i, i]\). Let us consider an inner node \( v \) and its two children \( u \) and \( w \). Then, \( u \) is associated with a segment \([a, q]\), \( w \) is associated with a segment \([q + 1, b]\), and \( v \) is associated with a segment \([a, b]\) for some \( 1 \leq a \leq q < b \leq 2^h \).

Note that because of the structure of the tree, we have \( q = (a + b)/2 \).

Each node \( v \) of the segment tree is labeled by color \( C(v) \), where \( 0 \leq C(v) \leq Z \). Assume that \( v \) is associated with \([a, b]\) segment. If \( C(v) = 0 \), then it means that \([a, b]\) segment is not colored at all or it has not a single color. If \( 1 \leq C(v) \leq Z \), then it means that the segment has a single color \( C(v) \), i.e. \( c_a = C(v), \ldots, c_b = C(v) \).

Additionally, we add two labels \( \text{Max}(v) \) and \( \text{Min}(v) \). \( a \leq \text{Max}(v) \leq b \) is the maximal index of a colored element of the segment. \( a \leq \text{Min}(v) \leq b \) is the minimal index of a colored element of the segment. Initially, \( \text{Max}(v) \leftarrow -1 \), \( \text{Min}(v) \leftarrow 2^h + 1 \).

For a vertex \( v \) and associated segment \([a, b]\), we use the following notation.

- \( \text{Left}(v) \) is a left border of the segment. \( \text{Left}(v) = a \)
- \( \text{Right}(v) \) is a right border of the segment. \( \text{Right}(v) = b \)
- \( \text{LeftChild}(v) \) is a left child of \( v \).
- \( \text{RightChild}(v) \) is a right child of \( v \).

Let us describe the processing of three types of query and constructing procedure.
Algorithm 1 CONSTRUCTST\((a, b)\). A procedure for constructing a segment tree for a segment \([a, b]\)

\[\begin{align*}
v &\leftarrow \text{a new node} \\
\text{LEFT}(v) &\leftarrow a \\
\text{RIGHT}(v) &\leftarrow b \\
C(v) &\leftarrow 0, \quad \text{Max}(v) &\leftarrow -1, \quad \text{Min}(v) &\leftarrow 2^h + 1 \\
\text{if } a \neq b \text{ then} &\quad \triangleright \text{not a leaf} \\
\text{LEFTChild}(v) &\leftarrow \text{CONSTRUCTST}(a, (a + b)/2) \\
\text{RIGHTChild}(v) &\leftarrow \text{CONSTRUCTST}((a + b)/2 + 1, b)
\end{align*}\]

Constructing procedure We can construct a segment tree using a simple recursive procedure. Let \(\text{CONSTRUCTST}(a, b)\) be a procedure that returns the root of a segment tree for a segment \([a, b]\). We present it in Algorithm 1.

Let us discuss the property of Algorithm 1.

**Lemma 1.** Time complexity of Algorithm 1 is \(O(d)\).

**Proof.** We construct each node in \(O(1)\). A number of nodes on each next level is twice bigger comparing to the previous one. The number of nodes on a level \(i\) is \(2^i\). The number of levels is \(h = \lceil \log_2 d \rceil\). So, the total time complexity is \(O(\sum_{i=0}^{h} 2^i) = O(2^h) = O(d)\).

**Request Query.** Assume that we want to get \(c_x\) for some \(1 \leq x \leq 2^h\). We start with the root node. Assume that we observe a node \(v\). If \(C(v) = 0\), then we go to the child that is associated with a segment \([a, b]\), where \(a \leq x \leq b\). We continue this process until we meet \(v\) such that \(C(v) \geq 1\) or \(v\) is a leaf. If \(C(v) \geq 1\), then the result is \(C(v)\). If \(C(v) = 0\) and \(v\) is a leaf, then \(c_x\) is not assigned yet. Let us describe this procedure in Algorithm 2.

Let us discuss the property of the algorithm.

**Lemma 2.** Algorithm 2 works correct with \(O(\log d)\) time complexity.

**Proof.** If the segment tree stores correct colors for segments, then the correctness of the algorithm follows from the description. The algorithm returns a color only if \(x\) belongs to a segment that has a single color.

On each step we change a vertex to a vertex on the next level. The tree is a full binary tree, therefore, it has \(h\) levels. Hence, the time complexity is \(O(h) = O(\log d)\) because \(2^{h-1} \leq d \leq 2^h\).

**Update Query.** Assume that we want to color a segment \([l, r]\) in \(c\) color, where \(1 \leq c \leq Z\), \(1 \leq l \leq r \leq 2^h\).

Let us describe two specific cases that are coloring a prefix and coloring a suffix.

Let us have a segment tree with a root node root. A segment \([q, t]\) is associated with root node.
Algorithm 2 ColorRequest\((x, \text{root})\). A request for a color \(c_x\) from a segment tree with \(\text{root}\) node as a root. If the color is not assigned, then the procedure returns 0.

\[
v \leftarrow \text{root} \\quad \text{while} \ v \text{ is not a leaf and } C(v) = 0 \text{ do} \\
\quad u \leftarrow \text{LeftChild}(v) \\
\quad w \leftarrow \text{RightChild}(v) \\
\quad \text{if } x \leq \text{Right}(u) \text{ then} \\
\quad \quad v \leftarrow u \\
\quad \text{end if} \\
\quad \text{if } x > \text{Right}(u) \text{ then} \\
\quad \quad v \leftarrow w \\
\quad \text{end if} \\
\text{end while} \\
\text{return } C(v).
\]

Firstly, assume that \([l, r]\) is a prefix of \([q, t]\), i.e. \(q = l\) and \(q \leq r \leq t\).

Assume that we observe a node \(v\) and an associated segment \([a, b]\). If \(v\) is a leaf, then we assign \(C(v) \leftarrow c\) and stop. Otherwise, we continue. We use a variable \(c'\) for an existing color. Initially \(c' \leftarrow 0\). If on some step \(C(v) \geq 1\) and \(c' = 0\), then we assign \(c' \leftarrow C(v)\). If \(c' \geq 1\) or \(C(v) = 0\), then we do not change \(c'\) because we already have a color for the segment from an ancestor.

Let \(u\) be the left child of \(v\), and let \(w\) be the right child of \(v\). Firstly, we update \(\max(v) \leftarrow \max(\max(v), r), \min(v) \leftarrow \text{Left}(v)\) because \([l, r]\) is a prefix. Secondly, we do the following action.

- If \(r \in \left[\frac{a + b}{2}, b\right]\), then we go to the left child \(u\). Additionally, if \(c' \geq 1\), then we color \(C(u) \leftarrow c\) because a segment of \(w\) has no intersection with \([l, r]\) and keeps its color \(c'\).
- If \(r \in \left[\left(\frac{a + b}{2} + 1\right), b\right]\), then we go to the right child \(w\). Additionally, we color \(C(u) \leftarrow c\) and update \(\min(u) \leftarrow \text{Left}(u), \max(u) \leftarrow \text{Right}(u)\) because \([\left(\frac{a + b}{2} + 1\right), b]\) of \(u\) is a subsegment of \([l, r]\).

Let us describe this procedure in Algorithm 2. It is presented in Appendix A.

Secondly, assume that \([l, r]\) is a suffix of \([q, t]\), i.e. \(t = r\) and \(q \leq l \leq t\). The function is similar to the previous one. The difference is the following. Let \(u\) be the left child of \(v\), and let \(w\) be the right child of \(v\). Firstly, we update \(\min(v) \leftarrow \min(\min(v), l), \max(v) \leftarrow \text{Right}(v)\) because \([l, r]\) is a suffix. Secondly, we do the following action.

- If \(l \in \left[\left(\frac{a + b}{2} + 1\right), b\right]\), then we go to the right child \(w\). Additionally, if \(c' \geq 1\), then we color \(C(u) \leftarrow c\) because a segment of \(u\) has no intersection with \([l, r]\) and we keep its color \(c'\).
- If \(l \in \left[\frac{a + b}{2}, b\right]\), then we go to the left child \(u\). Additionally, we color \(C(w) \leftarrow c\) and update \(\min(w) \leftarrow \text{Left}(w), \max(w) \leftarrow \text{Right}(w)\) because \([\left(\frac{a + b}{2} + 1\right), b]\) of \(w\) is a subsegment of \([l, r]\).
Let us describe this procedure in Algorithm 10. It is presented in Appendix A.

Finally, let us consider a general case for \([l, r]\), i.e. \(t = r\) and \(q \leq l \leq r \leq t\). Assume that we observe a node \(v\) and an associated segment \([a, b]\). If \(v\) is a leaf, then we assign \(C(v) \leftarrow c\) and stop. Otherwise, we continue. We use a variable \(c'\) for an existing color. Initially \(c' \leftarrow 0\). If on some step \(C(v) \geq 1\) and \(c' = 0\), then we assign \(c' \leftarrow C(v)\). If \(c' \geq 1\) or \(C(v) = 0\), then we do not change \(c'\). We update \(Min(v) \leftarrow \min(Min(v), l), Max(v) \leftarrow \max(Max(v), r)\).

- If \((a + b)/2 + 1 \leq l \leq r \leq b\), then we go to the right child \(w\). Additionally, if \(c' \geq 1\), then we color \(C(u) \leftarrow c'\) because a segment of \(u\) has no intersection with \([l, r]\) and we keep its color \(c'\).
- If \(a \leq l \leq (a + b)/2\), then we go to the left child \(u\). Additionally, if \(c' \geq 1\), then we color \(C(w) \leftarrow c'\) because a segment of \(w\) has no intersection with \([l, r]\) and we keep its color \(c'\).
- If \(a \leq l \leq (a + b)/2 \leq r \leq b\), then we split our segment to \([l, (a + b)/2]\) and \([(a + b)/2 + 1, r]\). The segment \([l, (a + b)/2]\) is a suffix of the segment tree with the root \(u\). For coloring it, we invoke \(\text{ColorUpdateSuffix}((l, (a + b)/2, c, c', u)\). The segment \([(a + b)/2 + 1, r]\) is a prefix of the segment tree with the root \(w\). For coloring it, we invoke \(\text{ColorUpdatePrefix}((a + b)/2 + 1, r, c, c', u)\).

Let us describe this procedure in Algorithm 10. It is presented in Appendix A.

Let us discuss properties of the algorithm.

**Lemma 3.** Algorithm 10 works correct with \(O(\log d)\) time complexity.

**Proof.** If the segment tree stores correct colors for segments, then the correctness of the algorithm follows from the description. The algorithm colors a required segment and keeps the color of the rest part.

Algorithm 8 and Algorithm 9 on each step change a vertex to a vertex on the next level. The tree is full binary tree, therefore, the tree has \(h\) levels. Hence, the time complexity of these two algorithms is \(O(h) = O(\log d)\) because \(2^{h-1} \leq d \leq 2^h\). Algorithm 10 on each step changes a vertex to a vertex on the next level then stops and invokes Algorithm 8 and Algorithm 9. Its time complexity also \(O(h) = O(\log d)\). We can say that procedures runs consistently. Therefore, the total time complexity is also \(O(\log d)\).

**Request the Closest Colored Element Query** Assume that we want to get the minimal index of a colored element from a segment \([l, r]\), where \(1 \leq l \leq r \leq 2^h\). Let \([q, t]\) be a segment of the root of the segment tree. Let us describe two specific cases that are requesting from a prefix of \([q, t]\) and requesting from a suffix of \([q, t]\).

Firstly, assume that \([l, r]\) is a prefix of \([q, t]\), i.e. \(q = l\) and \(q \leq r \leq t\). Assume that we observe a node \(v\) and an associated segment \([a, b]\). Let \(u\) be the left child of \(v\), and let \(w\) be the right child of \(v\). We do the following action.
– If \( r \leq (a + b)/2 \), then we go to the left child \( u \).
– If \( r > (a + b)/2 \) and \( \text{Min}(u) \neq 2^h + 1 \) (i.e., there are colored elements in the left child \( u \)), then the result is \( \text{Min}(u) \) and we stop the process.
– If \( r > (a + b)/2 \) and \( \text{Min}(u) = 2^h + 1 \) (i.e., there is no colored element in the left child \( u \)), then we go to the right child \( w \).

If there are no colored elements in \( v \), then the algorithm returns \( \text{NULL} \). Let us describe this procedure \( \text{GetClosestColorRightPrefix}(l, r, \text{root}) \) in Algorithm 11. It is presented in Appendix A.

Secondly, assume that \([l, r]\) is a suffix of \([q, t]\), i.e., \( t = r \) and \( q \leq l \leq t \). Assume that we observe a node \( v \) and an associated segment \([a, b]\). Let \( u \) be the left child of \( v \), and let \( w \) be the right child of \( v \). We do the following action.

– If \( l \geq (a + b)/2 + 1 \), then we go to the right child \( w \).
– If \( l \leq (a + b)/2 \) and \( \text{Min}(u) \neq 2^h + 1 \) (i.e., there are colored elements in the left child \( u \)), then we go to the left child \( u \).
– If \( l \leq (a + b)/2 \) and \( \text{Min}(u) = 2^h + 1 \) (i.e., there is no colored element in the left child \( u \)), then the result is \( \text{Min}(w) \) and we stop the process.

If there are no colored elements in \( v \), then it returns \( \text{NULL} \). Let us describe this procedure \( \text{GetClosestColorRightSuffix}(l, r, \text{root}) \) in Algorithm 12. It is presented in Appendix A.

Finally, let us consider the general case, i.e., \( q \leq l \leq r \leq t \). Assume that we observe a node \( v \) and an associated segment \([a, b]\). Let \( u \) be the left child of \( v \), and let \( w \) be the right child of \( v \). We do the following action.

– If \( (a + b)/2 + 1 \leq l \leq r \leq b \), then we go to the right child \( w \).
– If \( a \leq l \leq r \leq (a + b)/2 \), then we go to the left child \( u \).
– If \( a \leq l \leq (a + b)/2 \leq r \leq b \), then we split our segment to \([l, (a + b)/2]\) and \([(a+b)/2+1, r]\). The segment \([l, (a+b)/2]\) is a suffix of the segment tree with the root \( u \). We invoke \( \text{GetClosestColorRightSuffix}(l, (a+b)/2, u) \). If the result is not \( \text{NULL} \), then we return the result of the procedure. If the result is \( \text{NULL} \), then we invoke \( \text{GetClosestColorRightPrefix}((a + b)/2 + 1, r, u) \) and we return the result of the procedure.

If there are no colored elements in \( v \), then the algorithm returns \( \text{NULL} \).

We call this function \( \text{GetClosestColorRight}(l, r, \text{root}) \). We can define the function that returns the maximal index of a colored element symmetrically. We call it \( \text{GetClosestColorLeft}(l, r, \text{root}) \). Let us discuss the properties of the procedures.

**Lemma 4.** \( \text{GetClosestColorLeft}(l, r, \text{root}) \) and \( \text{GetClosestColorRight}(l, r, \text{root}) \) work correct with \( O(\log d) \) time complexity.

**Proof.** The proof is similar to the proof of Lemma 3.
3.2 Heavy-Light Decomposition

Heavy-light decomposition is a decomposition of a tree of size $n$ to a set of paths $\mathcal{P}$. The technique is presented in [11,19]. It has the following properties:

- Each vertex $v$ of the tree belongs to exactly one path from $\mathcal{P}$, i.e. all paths have no intersections and they cover all nodes of the tree.
- For any vertex $v$ a path from $v$ to the root of the tree contains vertexes of at most $\log_2 n$ paths from $\mathcal{P}$.
- Let us consider a vertex $v$ and a path $P \in \mathcal{P}$ such that $v \in P$. Then, $beg(v)$ is a vertex that belongs to $P$ and has the minimal height.
- For a node $v$ of the tree, let $P(v)$ be a path from $\mathcal{P}$ that contains $v$.
- For a node $v$ of the tree, let $index_P(v)$ be an index of an element of a path $P$. For an index $i$ of an element of a path $P$, let $vertex_P(i)$ be a vertex of the tree.
- We can construct the $\mathcal{P}$ set with $O(n)$ time complexity.

3.3 Lowest Common Ancestor

We use the Lowest Common Ancestor (LCA) problem in our algorithm.

**Lowest Common Ancestor (LCA) problem** For two nodes $u$ and $v$ of a tree, the Lowest Common Ancestor is a node $w$ such that $w$ is ancestor of $u$ and $v$ and $w$ is closest for $u$ and $v$ nodes.

There are several algorithms for solving this problem. Some of them [54] have the following properties:

**Lemma 5 ([54]).** There is an algorithm for LCA problem with the following properties:

- Time complexity of preprocessing is $O(n)$
- Time complexity of computing LCA for two vertexes is $O(1)$.

4 The Fast Online Algorithm for $k$-servers Problem on Trees

Let us describe an $k$-competitive algorithm for $k$-servers problem from [7].

**Chrobak-Larmore’s $k$-competitive algorithm for $k$-servers problem from [7].** Let us have a query on a vertex $q$ and servers are in vertexes $v_1, \ldots, v_k$. Let a server $i$ be active if there is no other servers on the path from $v_i$ to $q$. In each phase, we move each active servers to one step towards the vertex $q$. After each phase, the set of active servers can be changed. We repeat phases (moves of servers) until one of the servers reaches the query vertex $q$.

The naive implementation of the algorithm has time complexity $O(n)$ for each query. It can be the following. Firstly, we run the Depth-first search algorithm with time labels [8]. Using it, we can put labels to each node that allows us to check for any two vertexes $u$ and $v$, whether $u$ is an ancestor of $v$ in $O(1)$. After
that, we can move each active server to query step by step. Together all active servers cannot visit more than $O(n)$ vertexes.

Here we present an effective implementation of Chrobak-Larmore’s algorithm. The algorithm contains two parts that are preprocessing and query processing. The preprocessing part is done once and has $O(n)$ time complexity (Theorem 1). The query processing part is done for each query and has $O(k(\log n)^2)$ time complexity (Theorem 2).

### 4.1 Preprocessing

We do the following steps for preprocessing:

- We construct a Heavy-light decomposition $\mathcal{P}$ for the tree. The properties of decomposition are described in Section 3.2. Assume that we have `CONSTRUCTINGHLD()` subroutine for constructing $\mathcal{P}$.
- For each path $P \in \mathcal{P}$ we construct a segment tree that will be used for coloring problem that is described in Section 3.1. Assume that we have `CONSTRUCTINGSEGMENTTREE(P)` subroutine for constructing a segment tree for a path $P$. Let $ST_P$ be a segment tree for a path $P$.
- Additionally, for each vertex $v$ we compute a distance from the 1-st (root) node to $v$ node. We call it $dist(1,v)$. We can do it using Depth-first search algorithm [8].

The computing $dist(1,v)$ is simple algorithm, but we present it for completeness. Let us consider a vertex $u$ and set of children of the vertex $\text{CHILDREN}(u)$. Then, for any $v \in \text{CHILDREN}(u)$, we have $dist(1,v) = dist(1,u) + 1$. Additionally, $dist(1,1) = 0$. So, the distance computing is presented in recursive Algorithm 3.

#### Algorithm 3 COMPUTE_DISTANCE($u$). Computing distance to a vertex $u$.

```plaintext
for $v \in \text{CHILDREN}(u)$ do
    $dist(1,v) \leftarrow dist(1,u) + 1$
    COMPUTE_DISTANCE($v$)
end for
```

Finally, we have the following algorithm for preprocessing (Algorithm 4).

#### Algorithm 4 PREPROCESSING. Preprocessing procedure.

```plaintext
$\mathcal{P} \leftarrow \text{CONSTRUCTHLD}()$
for $P \in \mathcal{P}$ do
    $ST_P \leftarrow \text{CONSTRUCTSEGMENTTREE}(P)$
end for
$dist(1,1) \leftarrow 0$
COMPUTE_DISTANCE(1)
```
Let us discuss the properties of the preprocessing part of the algorithm.

**Theorem 1.** Algorithm 4 for preprocessing has time complexity $O(n)$

**Proof.** As it was mentioned in Section 3.2 the time complexity of Heavy-light decomposition $\mathcal{P}$ construction is $O(n)$.

Due to Lemma 1, time complexity of $\text{ConstructSegmentTree}(\mathcal{P})$ is $O(|\mathcal{P}|)$. The total time complexity of constructing all segment trees is $O\left(\sum_{\mathcal{P}\in\mathcal{P}} |\mathcal{P}| \right) = O(n)$ because of property of the decomposition.

Time complexity of $\text{ComputeDistance}$ is $O(n)$. Therefore, the total time complexity is $O(n)$.

### 4.2 Query Processing

Let us have a query on a vertex $q$ and servers are in vertexes $v_1, \ldots, v_k$. We do the following steps:

**Step 1.** Let us sort all servers by the distance to the node $q$. We can compute a distance $\text{dist}(v, q)$ between a node $v$ and a node $q$ by the following way. Let $l = \text{LCA}(v, q)$ be a lowest common ancestor of $v$ and $q$. Then, $\text{dist}(v, q) = \text{dist}(1, q) + \text{dist}(1, v) - 2 \cdot \text{dist}(1, l)$. We can use a HeapSort algorithm [20,8] or other fast sorting algorithms with time complexity $O(k \log k)$. Let $\text{Sort}(q, v_1, \ldots, v_k)$ be the sorting procedure. On the following steps we assume that $\text{dist}(v_i, q) \leq \text{dist}(v_{i+1}, q)$ for $i \in \{1, \ldots, k-1\}$.

**Step 2.** The first server from $v_1$ processes the query. We move them to $q$ node and color all nodes of a path from $v_1$ to $q$ to color 1. The color of a vertex shows the number of a server that visited the vertex. A detailed description of this step is in Section 4.2. Let the coloring process be implemented as a procedure $\text{ColorPath}(v_1, q, 1)$.

**Step 3.** For $i \in \{2, \ldots k\}$ we consider a server from $v_i$. It will be inactive when some other server with a smaller index becomes closer to a query than $i$-th server. Let $j$ be the index of the server such that the $i$-th server becomes inactive because of the $j$-th server. For obtaining $j$, we search the closest to $v_i$ colored vertex on the path from $v_i$ to $q$. The color of this vertex is $j$. Let the search of the closest colored vertex be implemented as a procedure $\text{GetClosestColor}(v_i, q)$. It is described in Section 4.2.

Let the obtained vertex be $w$ and its color is $j$. The $j$-th server reaches the node $w$ in $z = \text{dist}(v_j, w)$ steps. After that the $i$-th server becomes inactive. So, we should move the server to a vertex $v_i'$ to $\text{dist}(v_j, w)$ on the path from $v_i$ to $w$. Let the moving process be implemented as a procedure $\text{MOVE}(v_i, w, z)$. It is described in Section 4.2. Then, we color to color $i$ all vertexes on the path from $v_i$ to $v_i'$.

Let us describe the procedure as Algorithm 5.

**Coloring of a Path** Let us consider a problem of coloring vertexes on a path from a node $v$ to a node $u$. The color is $c$. 
Proof. Due to properties of Heavy-light decomposition from Section 3.2, the coloring process is $O(ColorUpdate)$ for $i \in \{2, \ldots, k\}$ do
$(w, j) \leftarrow GetClosestColor(v, q)$
$z \leftarrow dist(v, w)$
$v_i \leftarrow Move(v, w, z)$
$ColorPath(v, v_i, i)$
end for

Let $l = LCA(v, u)$ be a LCA of $v$ and $u$. Let $P_1, \ldots, P_t \in \mathcal{P}$ be paths that contains vertexes of a path from $v$ to $l$ and let $P'_1, \ldots, P'_t \in \mathcal{P}$ be paths that contains vertexes of a path from $l$ to $u$. Let
$$w_0 = v, \ w_0 \in P_1; \ w_1 = beg(P_1), \ Parent(w_1) \in P_2; \ w_2 = beg(P_2), \ Parent(w_2) \in P_3; \ldots \ w_{t-1} = beg(P_{t-1}), \ Parent(w_{t-1}) \in P_t; \ w_t = l;$$
and
$$w'_0 = u, \ w'_0 \in P'_1; \ w'_1 = beg(P'_1), \ Parent(w'_1) \in P'_2; \ w'_2 = beg(P'_2), \ Parent(w'_2) \in P'_3; \ldots \ w'_{t-1} = beg(P'_{t-1}), \ Parent(w'_{t-1}) \in P'_t; \ w'_t = l;$$
Then, the coloring process is $ColorUpdate(index_{P_i}(w_{i-1}), index_{P_i}(w_i), c, ST_{P_i})$ for $i \in \{1, \ldots, t\}$ and $ColorUpdate(index_{P'_i}(w'_{i-1}), index_{P'_i}(w'_i), c, ST_{P'_i})$ for $i \in \{1, \ldots, t\}$.

The procedure is presented as Algorithm 5.

Let us discuss time complexity of the algorithm.

Lemma 6. Time complexity of Algorithm 5 is $O((\log n)^2)$.

Proof. Due to properties of Heavy-light decomposition from Section 3.2, $t, t' = O(\log n)$. Due to Lemma 3 each invocation of ColorUpdate for $P$ has time complexity $O(\log |P|) = O(\log n)$. So, the total time complexity is $O((\log n)^2)$.

The Search of the Closest Colored Vertex Let us consider the problem of searching the closest colored vertex on the path from $v$ to $u$. The idea is similar to the idea of the previous section.

Let $l = LCA(v, u)$ be a LCA of $v$ and $u$. Let $P_1, \ldots, P_t \in \mathcal{P}$ be paths that contains vertexes of a path from $v$ to $l$ and let $P'_1, \ldots, P'_t \in \mathcal{P}$ be paths that contains vertexes of a path from $l$ to $u$. Let $w_0 = v, \ w_0 \in P_1; \ w_1 = beg(P_1), \ Parent(w_1) \in P_2; \ w_2 = beg(P_2), \ Parent(w_2) \in P_3; \ldots \ w_{i-1} = beg(P_{i-1}), \ Parent(w_{i-1}) \in P_i; \ w_i = l; \ and \ w'_0 = u, \ w'_0 \in P'_1; \ w'_1 = beg(P'_1), \ Parent(w'_1) \in P'_2; \ w'_2 = beg(P'_2), \ Parent(w'_2) \in P'_3; \ldots \ w'_{i-1} = beg(P'_{i-1}), \ Parent(w'_{i-1}) \in P'_{i}; \ w'_i = l; \ then, \ the \ searching \ process \ is \ the \ following. \ We \ invoke \ GetClosestColorRight(index_{P_i}(w_i), index_{P'_i}(w'_i))$ for $i \in \{1, \ldots, t\}$. We stop on the minimal $i$ such that a result is not NULL. If all

Algorithm 5 QUERY(q). Query procedure.

```
SORT(q, v_1, \ldots, v_k)
COLORPATH(v_1, q, 1)
for i \in \{2, \ldots, k\} do
  (w, j) \leftarrow GETCLOSESTCOLOR(v, q)
  z \leftarrow dist(v, w)
  v_i \leftarrow MOVE(v, w, z)
  COLORPATH(v, v_i, i)
end for
```
ColorPath($v, u, c$). Coloring the path between $v$ and $u$.

\begin{algorithm}
\begin{algorithmic}
\State $l \leftarrow \text{LCA}(v, u)$
\State $w \leftarrow v$
\State $P \leftarrow P(v)$
\While{$P \neq P(l)$}
\State $bw \leftarrow \text{beg}(P)$
\State \Call{ColorUpdate}{index$_{P}(w)$, index$_{P}(bw)$, $c$, $ST_P$}
\State $w \leftarrow \text{Parent}(bw)$
\State $P \leftarrow P(w)$
\EndWhile
\State \Call{ColorUpdate}{index$_{P}(w)$, index$_{P}(l)$, $c$, $ST_P$}
\State $w \leftarrow u$
\State $P \leftarrow P(u)$
\While{$P \neq P(l)$}
\State $bw \leftarrow \text{beg}(P)$
\State \Call{ColorUpdate}{index$_{P}(w)$, index$_{P}(bw)$, $c$, $ST_P$}
\State $w \leftarrow \text{Parent}(bw)$
\State $P \leftarrow P(w)$
\EndWhile
\State \Call{ColorUpdate}{index$_{P}(w)$, index$_{P}(l)$, $c$, $ST_P$}
\end{algorithmic}
\end{algorithm}

of them are $NULL$, then we continue. We invoke \text{GetClosestColorRight} (index$_{P'}(\text{Parent}(w'_{i-1})),$ index$_{P'}(w'_i),$ $ST_{P'})$ for $i \in \{t', \ldots, 2\}$. We stop on the maximal $i$ such that a result is not $NULL$.

The procedure is presented as Algorithm 7.
Algorithm 7 ColorPath(v, u, c). Coloring the path between v and u.

\[ l \leftarrow \text{LCA}(v, u) \]
\[ w \leftarrow v \]
\[ P \leftarrow P(v) \]
\[ g \leftarrow \text{NULL} \]
\[ \text{while } g = \text{NULL} \text{ and } P \neq P(l) \text{ do} \]
\[ \quad bw \leftarrow \text{beg}(P) \]
\[ \quad g \leftarrow \text{GetClosestColorRight}(\text{index}_P(w), \text{index}_P(bw), ST_P) \]
\[ \quad \text{if } g = \text{NULL} \text{ then} \]
\[ \quad \quad w \leftarrow \text{PARENT}(bw) \]
\[ \quad \quad P \leftarrow P(w) \]
\[ \quad \text{end if} \]
\[ \text{end while} \]
\[ \text{if } g = \text{NULL} \text{ then} \]
\[ \quad g \leftarrow \text{GetClosestColorRight}(\text{index}_P(w), \text{index}_P(l), ST_P) \]
\[ \text{end if} \]
\[ \text{if } g = \text{NULL} \text{ then} \]
\[ \quad i \leftarrow 0 \]
\[ \quad w_i' \leftarrow u \]
\[ \quad P \leftarrow P(u) \]
\[ \text{while } g = \text{NULL} \text{ and } P \neq P(l) \text{ do} \]
\[ \quad \quad i \leftarrow i + 1 \]
\[ \quad \quad w_i' \leftarrow \text{beg}(P) \]
\[ \quad \quad bw \leftarrow \text{PARENT}(w_i') \]
\[ \quad \quad P \leftarrow P(bw) \]
\[ \text{end while} \]
\[ \text{if } g = \text{NULL} \text{ then} \]
\[ \quad g \leftarrow \text{GetClosestColorLeft}(\text{index}_P(\text{PARENT}(w_i')), \text{index}_P(l), ST_P) \]
\[ \text{end if} \]
\[ \text{while } g = \text{NULL} \text{ do} \]
\[ \quad P \leftarrow P(w_i) \]
\[ \quad bw \leftarrow \text{PARENT}(w_i'_{i - 1}) \]
\[ \quad g \leftarrow \text{GetClosestColorRight}(\text{index}_P(bw), \text{index}_P(w_i'), ST_P) \]
\[ \quad i \leftarrow i - 1 \]
\[ \text{end while} \]
\[ \text{end if} \]
\[ \text{resW} \leftarrow \text{vertex}_P(g) \]
\[ \quad j \leftarrow \text{ColorRequest}(g, ST_P) \]
\[ \text{return } (\text{resW}, j) \]

Let us discuss time complexity of the algorithm

**Lemma 7.** Time complexity of Algorithm 7 is \( O((\log n)^2) \).

**Proof.** Due to properties of Heavy-light decomposition from Section 3.2, \( t, t' = O(\log n) \). Due to Lemma 3, each invocation of GetClosestColorLeft or GetClosestColorRight for \( P \) has time complexity \( O(\log |P|) = O(\log n) \). So, the total time complexity is \( O((\log n)^2) \).
Moving of a Server  Let us consider a moving of a server from \( v \) to distance \( g \) on a path from \( v \) to \( u \). The idea is similar to the idea from the previous section.

Let \( l = LCA(v, u) \) be a LCA of \( v \) and \( u \). Let \( P_1, \ldots, P_t \in \mathcal{P} \) be paths that contains vertexes of a path from \( v \) to \( l \) and let \( P_1', \ldots, P_t' \in \mathcal{P} \) be paths that contains vertexes of a path from \( l \) to \( u \). Let \( w_0 = v, w_0 \in P_1; w_1 = \text{beg}(P_1), \text{Parent}(w_1) = P_2; w_2 = \text{beg}(P_2), \text{Parent}(w_2) = P_3; \ldots; \text{w}_{t-1} = \text{beg}(P_{t-1}), \text{Parent}(w_{t-1}) = P_t; \text{w}_t = u, \text{w}_t \in P_t \) and \( v_0 \in P_1'; v_1 = \text{beg}(P_1'), \text{Parent}(v_1) = P_2'; v_2 = \text{beg}(P_2'), \text{Parent}(v_2) = P_3', \ldots; v_{t'} = \text{beg}(P_{t'}'), \text{Parent}(v_{t'}') \in P_t' \).

Then, the searching process is the following. We check whether distance \( \text{dist}(\text{Parent}(w_{i-1}), w_i) \leq g \). If \( \text{dist}(\text{Parent}(w_{i-1}), w_i) \leq g \), then we can return the vertex \( \text{vertex}_P(\text{index}_P(\text{Parent}(w_{i-1}))) + g \) as a result and stop the process. Otherwise, we reduce \( t \leftarrow t - \text{dist}(\text{Parent}(w_{i-1}), w_i) - 1 \) and move to the next \( i \), i.e. \( i \leftarrow i + 1 \). We do it for \( i \in \{1, \ldots, t\} \).

If \( g > 0 \), then we continue with path from \( l \) to \( u \). We check whether distance \( \text{dist}(\text{Parent}(w'_{i-1}), w'_i) \leq g \). If \( \text{dist}(\text{Parent}(w'_{i-1}), w'_i) \leq g \), then we can return the vertex \( \text{vertex}_P(\text{index}_P(w'_i) - g) \) as a result and stop the process. Otherwise, we reduce \( t \leftarrow t - \text{dist}(\text{Parent}(w'_{i-1}), w'_i) - 1 \) and move to the previous \( i \), i.e. \( i \leftarrow i - 1 \). We do it for \( i \in \{t', \ldots, 1\} \).

Lemma 8. Time complexity the moving is \( O(\log n) \).

Proof. Due to properties of Heavy-light decomposition from Section \ref{sec:heavy-light}, \( t, t' = O(\log n) \). The time complexity for processing of each path is \( O(1) \). So, the total time complexity is \( O(\log n) \).

Complexity of the Query Processing

Theorem 2. Time complexity of the query processing is \( O(k(\log n)^2) \).

Proof. The complexity of servers sorting by distance is \( O(k \log k) \). Due to Lemma \ref{lem:query-processing-complexity} and Lemma \ref{lem:query-processing-complexity-upper-bound}, the complexity for processing one server is \( O(\log n + (\log n)^2(\log n)^2) = O((\log n)^2) \). So, the total complexity of processing all servers is \( O(k \log k + k(\log n)^2) = O(k(\log n)^2) \) because \( k < n \).

References

1. F. Ablayev, M. Ablayev, K. Khadiev, and A. Vasiliev. Classical and quantum computations with restricted memory. LNCS, 11011:129–155, 2018.
2. Ganesh R Baliga and Anil M Shende. On space bounded server algorithms. In Proceedings of ICCI’93: 5th International Conference on Computing and Information, pages 77–81, IEEE, 1993.
3. L. Becchetti and E. Koutsoupias. Competitive analysis of aggregate max in windowed streaming. In ICALP, volume 5555 of LNCS, pages 156–170, 2009.
4. Michael A Bender and Martin Farach-Colton. The lca problem revisited. In Latin American Symposium on Theoretical Informatics, pages 88–94. Springer, 2000.
5. Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

6. J. Boyar, K. S. Larsen, and A. Maiti. The frequent items problem in online streaming under various performance measures. *International Journal of Foundations of Computer Science*, 26(4):413–439, 2015.

7. Marek Chrobak and Lawrence L Larmore. An optimal on-line algorithm for k servers on trees. *SIAM Journal on Computing*, 20(1):144–148, 1991.

8. T. H Cormen, C. E Leiserson, R. L Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

9. Michele Flammini, Alfredo Navarra, and Gaia Nicosia. Efficient offline algorithms for the bicriteria k-server problem and online applications. *Journal of Discrete Algorithms*, 4(3):414–432, 2006.

10. Y. Giannakopoulos and E. Koutsoupias. Competitive analysis of maintaining frequent items of a stream. *Theoretical Computer Science*, 562:23–32, 2015.

11. Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

12. Stephen Hughes. A new bound for space bounded server algorithms. In *Proceedings of the 33rd annual on Southeast regional conference*, pages 165–169, 1995.

13. A. R Karlin, M. S Manasse, L. Rudolph, and D. D Sleator. Competitive snoopy caching. In *FOCS, 1986., 27th Annual Symposium on*, pages 244–254. IEEE, 1986.

14. K. Khadiev, A. Khadieva, and I. Mannapov. Quantum online algorithms with respect to space and advice complexity. *Lobachevskii Journal of Mathematics*, 39(9):1210–1220, 2018.

15. Dennis Komm. *An Introduction to Online Computation: Determinism, Randomization, Advice*. Springer, 2016.

16. Antti Laaksonen. *Guide to Competitive Programming*. Springer, 2017.

17. Tomislav Rudec, Alfonzo Baumgartner, and Robert Manger. A fast work function algorithm for solving the k-server problem. *Central European Journal of Operations Research*, 21(1):187–205, 2013.

18. Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

19. Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

20. J. W. J. Williams. Algorithm 232 - heapsort. *Commun. ACM*, 7(6):347–349, June 1964.
A Algorithms for Coloring Problem on a Segment Tree

**Algorithm 8** COLORUPDATEPREFIX$(l, r, c, c', root)$. A query of update color of a prefix segment $[l, r]$ by a color $c$. The query is for a segment tree with $root$ node as a root. $c'$ is a color for rest part of the segment of $root$. If $c'$ is not assigned, then $c' = 0$

$v ← root$

while $v$ is not a leaf do

if $c' = 0$ and $C(v) ≥ 1$ then

$c' ← C(v)$

end if

Max$(v) ← \max$($Max(v), r$), Min$(v) ← \text{Left}(v)$

$u ← \text{LeftChild}(v)$

$w ← \text{RightChild}(v)$

if $r ≤ \text{Right}(u)$ then

if $c' ≥ 1$ then

$C(w) ← c'$

end if

$v ← u$

end if

if $r > \text{Right}(u)$ then

$C(u) ← c$, Min$(u) ← \text{Left}(u)$, Max$(u) ← \text{Right}(u)$

$v ← w$

end if

end while

$C(v) ← c$, Min$(v) ← \text{Left}(v)$, Max$(v) ← \text{Right}(v)$


Algorithm 9 ColorUpdateSuffix($l, r, c, c', root$). A query of update color of a suffix segment $[l, r]$ by a color $c$. The query is for a segment tree with root node as a root. $c'$ is a color for rest part of the segment of root. If $c'$ is not assigned, then $c' = 0$

$v \leftarrow \text{root}$

\textbf{while} $v$ is not a leaf \textbf{do}

\hspace{1em} \textbf{if} $c' = 0$ and $C(v) \geq 1$ \textbf{then}

\hspace{2em} $c' \leftarrow C(v)$

\hspace{1em} \textbf{end if}

$Min(v) \leftarrow \min(Min(v), l)$, $Max(v) \leftarrow \text{Right}(v)$

$u \leftarrow \text{LeftChild}(v)$

$w \leftarrow \text{RightChild}(v)$

\hspace{1em} \textbf{if} $l \geq \text{Left}(w)$ \textbf{then}

\hspace{2em} \textbf{if} $c' \geq 1$ \textbf{then}

\hspace{3em} $C(u) \leftarrow c'$

\hspace{2em} \textbf{end if}

$\text{v} \leftarrow w$

\hspace{1em} \textbf{end if}

\hspace{1em} \textbf{if} $l < \text{Left}(w)$ \textbf{then}

$C(w) \leftarrow c$, $Min(w) \leftarrow \text{Left}(w)$, $Max(w) \leftarrow \text{Right}(w)$

$\text{v} \leftarrow u$

\hspace{1em} \textbf{end if}

\hspace{1em} \textbf{end while}

$C(v) \leftarrow c$, $Min(v) \leftarrow \text{Left}(v)$, $Max(v) \leftarrow \text{Right}(v)$
Algorithm 10 COLORUpdate($l, r, c, root$). A query of update color of a segment $[l, r]$ by a color $c$. The query is for a segment tree with root node as a root.

$v \leftarrow$ root
$c' \leftarrow 0$

$Split \leftarrow False$

while $v$ is not a leaf and $Split = False$ do
  if $c' = 0$ and $C(v) \geq 1$ then
    $c' \leftarrow C(v)$
  end if
  $Min(v) \leftarrow \min(Min(v), l)$, $Max(v) \leftarrow \max(Max(v), r)$
  $u \leftarrow$ LEFTChild($v$)
  $w \leftarrow$ RIGHTChild($v$)
  if $l \geq LEFT(w)$ then
    if $c' \geq 1$ then
      $C(u) \leftarrow c'$
    end if
    $v \leftarrow w$
  end if
  if $r \leq RIGHT(u)$ then
    if $c' \geq 1$ then
      $C(w) \leftarrow c'$
    end if
    $v \leftarrow u$
  end if
  if $l \leq RIGHT(u)$ and $r \geq LEFT(w)$ then
    $Split \leftarrow True$
    COLORUpdateSuffix($l$, RIGHT($u$), $c, c', u$)
    COLORUpdatePrefix(LEFT($w$), $r, c, c', w$)
  end if
end while
if $v$ is a leaf then
  $C(v) \leftarrow c$
end if
**Algorithm 11** GETCLOSESTCOLORRIGHTPREFIX($l, r, root$). A query of the minimal index of a colored element of a prefix segment $[l, r]$. It returns NULL if there is no such elements

$v \leftarrow root$
$Result \leftarrow NULL$

if $Min(v) \neq 2^h + 1$ then
    $Found \leftarrow False$

    while $v$ is not a leaf and $Found = False$ do
        $u \leftarrow$ LEFTCHILD($v$)
        $w \leftarrow$ RIGHTCHILD($v$)
        if $r \leq$ RIGHT($u$) then
            $v \leftarrow u$
        end if
        if $r \geq$ LEFT($w$) and $Min(u) \neq 2^h + 1$ then
            $Result \leftarrow Min(u)$
            $Found \leftarrow True$
        end if
        if $r \geq$ LEFT($w$) and $Min(u) = 2^h + 1$ then
            $v \leftarrow w$
        end if
    end while

if $Found = False$ and $Min(v) \neq 2^h + 1$ then
    $Result = Min(v)$
end if

return $Result$
Algorithm 12 GetClosestColorRightSuffix(l, r, root). A query of the minimal index of a colored element of a suffix segment [l, r]. It returns NULL if there is no such elements.

\[ v \leftarrow \text{root} \]
\[ \text{Result} \leftarrow \text{NULL} \]
\[ \text{if } \text{Min}(v) \neq 2^h + 1 \text{ then} \]
\[ \text{Found} \leftarrow \text{False} \]
\[ \textbf{while } v \text{ is not a leaf and } \text{Found} = \text{False} \textbf{ do} \]
\[ u \leftarrow \text{LeftChild}(v) \]
\[ w \leftarrow \text{RightChild}(v) \]
\[ \text{if } l \geq \text{Left}(w) \text{ then} \]
\[ v \leftarrow w \]
\[ \textbf{end if} \]
\[ \text{if } l \leq \text{Right}(u) \text{ and } \text{Min}(u) \neq 2^h + 1 \text{ then} \]
\[ v \leftarrow u \]
\[ \textbf{end if} \]
\[ \text{if } l \leq \text{Right}(u) \text{ and } \text{Min}(u) = 2^h + 1 \text{ then} \]
\[ \text{Result} \leftarrow \text{Min}(w) \]
\[ \text{Found} \leftarrow \text{True} \]
\[ \textbf{end if} \]
\[ \textbf{end while} \]
\[ \text{if } \text{Found} = \text{False} \text{ and } \text{Min}(v) \neq 2^h + 1 \text{ then} \]
\[ \text{Result} = \text{Min}(v) \]
\[ \textbf{end if} \]
\[ \textbf{end if} \]
\[ \text{return } \text{Result} \]