Range Mode and Range Median Queries on Lists and Trees*

Danny Krizanc[1], Pat Morin[2], and Michiel Smid[2]

[1] Department of Mathematics and Computer Science, Wesleyan University, Middletown, CT 06459 USA dkrizanc@wesleyan.edu

[2] School of Computer Science, Carleton University, 1125 Colonel By Drive, Ottawa, ON K1S 5B6, CANADA {morin,michiel}@cs.carleton.ca

Abstract. We consider algorithms for preprocessing labelled lists and trees so that, for any two nodes u and v we can answer queries of the form: What is the mode or median label in the sequence of labels on the path from u to v.

1 Introduction

Let $A = a_1, \ldots, a_n$ be a list of elements of some data type. Many researchers have considered the problem of preprocessing A to answer *range queries*. These queries take two indices $1 \leq i \leq j \leq n$ and require computing $F(a_i, \ldots a_j)$ where F is some function of interest.

When the elements of A are numbers and F computes the sum of its inputs, this problem is easily solved using linear space and constant query time. We create an array B where b_i is the sum of the first i elements of A. To answer queries, we simply observe that $a_i + \cdots + a_j = b_j - b_{i-1}$. Indeed this approach works even if we replace $+$ with any group operator for which each element x has an easily computable inverse $-x$.

A somewhat more difficult case is when $+$ is only a semigroup operator, so that there is no analagous notion of $-$. In this case, Yao [18] shows how to preprocess a list A using $O(nk)$ space so that queries can be answered in $O(\alpha_k(n))$ time, for any integer $k \geq 1$. Here α_k is a slow growing function at the kth level of the primitive recursion hierarchy. To achieve this result the authors show how to construct a graph G with vertex set $V = \{1, \ldots, n\}$ such that, for any pair of indices $1 \leq i \leq j \leq n$, G contains a path from i to j of length at most $\alpha_k(n)$ that visits nodes in increasing order. By labelling each edge (u, v) of G with the sum of the elements $a_u, \ldots, a_v$, queries are answered by simply summing the edge labels along a path. This result is optimal when F is defined by a general semigroup operator [19].

A special case of a semigroup operator is the min (or max) operator. In this case, the function F is the function that takes the minimum (respectively

* This work was partly funded by the Natural Sciences and Engineering Research Council of Canada.

maximum) of its inputs. By making use of the special properties of the min and max functions several researchers [1, 2] have given data structures of size $O(n)$ that can answer range minimum queries in $O(1)$ time. The most recent, and simplest, of these is due to Bender and Farach-Colton [1].

Range queries also have a natural generalization to trees, where they are sometimes call *path queries*. In this setting, the input is a tree T with labels on its nodes and a query consists of two nodes u and v. To answer a query, a data structure must compute $F(l_1, \ldots, l_k)$, where $l_1, \ldots, l_k$ is the set of labels encountered on the path from u to v in T. For group operators, these queries are easily answered by an $O(n)$ space data structure in $O(1)$ time using data structures for lowest-common-ancestor queries. For semi-group operators, these queries can be answered using the same resource bounds as for lists [18, 19].

In this paper we consider two new types of range queries that, to the best of our knowledge, have never been studied before. In particular, we consider range queries where F is the function that computes a mode or median of its input. A mode of a multiset S is an element of S that occurs at least as often as any other element of S. A median of S is the element that is greater than or equal to exactly $\lfloor |S|/2 \rfloor$ elements of S. Our results for range mode and range median queries are summarized in Table 1. Note that neither of these queries is easily expressed as a group, semi-group, or min/max query so they require completely new data structures.

Range Mode Queries on Lists				
§	Space	Query Time	Space × Time	Restrictions
2.1	$O(n^{2-2\epsilon})$	$O(n^{\epsilon} \log n)$	$O(n^{2-\epsilon} \log n)$	$0 < \epsilon \leq 1/2$
2.2	$O(n^2 \log\log n / \log n)$	$O(1)$	$O(n^2 \log\log n / \log n)$	–

Range Mode Queries on Trees				
§	Space	Query Time	Space × Time	Restrictions
2.1	$O(n^{2-2\epsilon})$	$O(n^{\epsilon} \log n)$	$O(n^{2-\epsilon} \log n)$	$0 < \epsilon \leq 1/2$

Range Median Queries on Lists				
§	Space	Query Time	Space × Time	Restrictions
4.2	$O(n \log^2 n / \log\log n)$	$O(\log n)$	$O(n \log^3 n / \log\log n)$	–
4.3	$O(n^2 \log\log n / \log n)$	$O(1)$	$O(n^2 \log\log n / \log n)$	–
4.4	$O(n \log_b n)$	$O(b \log^2 n / \log b)$	$O(nb \log^3 n / \log^2 b)$	$2 \leq b \leq n$
4.4	$O(n)$	$O(n^{\epsilon})$	$O(n^{1+\epsilon})$	$\epsilon > 0$

Range Median Queries on Trees				
§	Space	Query Time	Space × Time	Restrictions
5.1	$O(n \log^2 n)$	$O(\log n)$	$O(n \log^3 n)$	–
5.2	$O(n \log_b n)$	$O(b \log^3 n / \log b)$	$O(nb \log^4 n / \log^2 b)$	$2 \leq b \leq n$
5.2	$O(n)$	$O(n^{\epsilon})$	$O(n^{1+\epsilon})$	–

Table 1. Summary of results in this paper.

The remainder of this paper is organized as follows: In Section 2 we consider range mode queries on lists. In Section 3 we discuss range mode queries on trees. In Section 4 we study range median queries on lists. In Section 5 we present data structures for range median queries on trees.

Because of space constraints, we do not include proofs of any lemmata and only sketch proofs of some theorems. Details are available in the full version of the paper.

2 Range Mode Queries on Lists

In this section, we consider range mode queries on a list $A = a_1, \ldots, a_n$. More precisely, our task is to preprocess A so that, for any indices i and j, $1 \leq i \leq j \leq n$, we can return an element of $a_i, \ldots, a_j$ that occurs at least as frequently as any other element. Our approach is to first preprocess A for *range counting queries* so that, for any i, j and x we can compute the number of occurences of x in $a_i, \ldots, a_j$. Once we have done this, we will show how a range mode query can be answered using a relatively small number of these range counting queries.

To answer range counting queries on A we use a collection of sorted arrays, one for each unique element of A. The array for element x, denoted A_x contains all the indices $1 \leq i \leq n$ such that $a_i = x$, in sorted order. Now, simply observe that if we search for i and j in the array A_x, we find two indices k and l, respectively, such that, the number of occurences of x in $a_i, \ldots, a_j$ is $l - k + 1$. Thus, we can answer range counting queries for x in $O(\log n)$ time. Furthermore, since each position in A contributes exactly one element to one of these arrays, the total size of these arrays is $O(n)$, and they can all be computed easily in $O(n \log n)$ time.

The remainder of our solution is based on the following simple lemma about modes in the union of three sets.

Lemma 1 *Let A, B and C be any multisets. Then, if a mode of $A \cup B \cup C$ is not in A or C then it is a mode of B.*

In the next two subsections we show how to use this observation to obtain efficient data structures for range mode queries. In the first section we show how it can be used to obtain an efficient time-space tradeoff. In the subsequent section we show how to it can be used to obtain a data structure with $O(1)$ query time that uses subquadratic space.

2.1 A Time-Space Tradeoff

To obtain a time-space tradeoff, we partition the list A into b blocks, each of size n/b. We denote the ith block by B_i. For each pair of blocks B_i and B_j, we compute the mode $m_{i,j}$ of $B_{i+1} \cup \cdots \cup B_{j-1}$ and store this value in a lookup table of size $O(b^2)$. At the same time, we convert A into an array so that we can access any element in constant time given its index. This gives us a data structure of size $O(n + b^2)$.

To answer a range mode query (i, j) there are two cases to consider. In the first case, $j - i \leq n/b$, in which case we can easily compute the mode of $a_i, \ldots, a_j$ in $O((n/b)\log n)$ time by, for example, sorting $a_i, \ldots, a_j$ and looking for the longest run of consecutive equal elements.

The second case occurs when $j - i > n/b$, in which case a_i and a_j are in two different blocks (see Fig. 1). Let $B_{i'}$ be the block containing i and let $B_{j'}$ be the block containing j. Lemma 1 tells us that the answer to this query is either an element of $B_{i'}$, an element of $B_{j'}$, or is the mode $m_{i',j'}$ of $B_{i'+1} \cup \cdots \cup B_{j'+1}$. Thus, we have a set of at most $2n/b + 1$ candidates for the mode. Using the range counting arrays we can determine which of these candidates is a mode by performing at most $2n/b + 1$ queries each taking $O(\log n)$ time, for a query time of $O((n/b)\log n)$. By setting $b = n^{1-\epsilon}$, we obtain the following theorem:

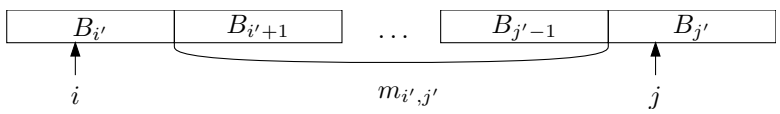

Fig. 1. The mode of $a_i, \ldots, a_j$ is either an element of $B_{i'}$, an element of $B_{j'}$ or is the mode $m_{i',j'}$ of $B_{i'+1}, \ldots, B_{j'+1}$.

Theorem 1 *For any $0 < \epsilon \leq 1/2$, there exists a data structure of size $O(n^{2-2\epsilon})$ that answers range mode queries on lists in time $O(n^{\epsilon}\log n)$.*[3]

2.2 A Constant Query-Time Subquadratic Space Solution

Initially, one might suspect that any data structure that achieves $O(1)$ query time must use $\Omega(n^2)$ space. However, in the full version of the paper we prove:

Theorem 2 *There exists a data structure of size $O(n^2 \log\log n/\log n)$ that can answer range mode queries on lists in $O(1)$ time.*

3 Range Mode Queries on Trees

In this section we consider the problem of range mode queries on trees. The outline of the data structure is essentially the same as our data structure for

[3] The query time of Theorem 1 can be improved by observing that our range counting data structure operates on the universe $1, \ldots, n$ so that using complicated integer searching data structures [15, 14, 16], the logarithmic term in the query time can be reduced to a doubly-logarithmic term. We observed this but chose not to pursue it because the theoretical improvement is negligible compared to the polynomial factor already in the query time. The same remarks apply to the data structure of Section 3.

lists, but there are some technical difficulties which come from the fact that the underlying graph is a tree.

We begin by observing that we may assume the underlying tree T is a rooted binary tree. To see this, first observe that we can make T rooted by choosing any root. We make T binary by expanding any node with $d > 2$ children into a complete binary tree with d leaves. The root of this little tree will have the original label of the node we expanded and all other nodes that we create are assigned unique labels so that they are never the answer to a range mode query (unless no element in the range occurs more than once, in which case we can correctly return the first element of the range). This transformation does not increase the size of T by more than a small constant factor.

To mimic our data structure for lists we require two ingredients: (1) we should be able to answer range counting queries of the form: Given a label x and two nodes u and v, how many times does the label x occur on the path from u to v? and (2) we must be able to partition our tree into $O(b)$ subtrees each of size approximately n/b.

We begin with the second ingredient, since it is the easier of the two. To partition T into subtrees we make use of the well-known fact (see, e.g., Reference [3]) that every binary tree has an edge whose removal partitions the tree into two subtrees neither of which is more than $2/3$ the size of the original tree. By repeatedly applying this fact, we obtain a set of edges whose removal partitions our tree into $O(b)$ subtrees none of which has size more than n/b. For each pair of these subtrees, we compute the mode of the labels on the path from one subtree to the other and store all these modes in a table of size $O(b^2)$. Also, we give a new data field to each node v of T so that in constant time we can determine the index of the subtree to which v belongs.

Next we need a concise data structure for answering range counting queries. Define the lowest-common-ancestor (LCA) of two nodes u and v in T to be the node on the path from u to v that is closest to the root of T. Let $x(v)$ denote the number of nodes labelled x on the path from the root of T to v, or 0 if v is nil. Suppose w is the LCA of u and v. Then it is easy to verify that the number of nodes labelled x on the path from u to v in T is exactly $x(u) + x(v) - 2x(\text{parent}(w))$, where $\text{parent}(w)$ denotes the parent of w in T or nil if w is the root of T.

There are several data structures for preprocessing T for LCA queries that use linear space and answer queries in $O(1)$ time the simplest of which is due to Bender and Farach-Colton [1]. Thus all that remains is to give a data structure for computing $x(u)$ for any value x and any node u of T. Consider the minimal subtree of T that is connected and contains the root of T as well as all nodes whose label is x. Furthermore, contract all degree 2 vertices in this subtree with the possible exception of the root and call the resulting tree T_x (see Fig. 2). It is clear that the tree T_x has size proportional to the number of nodes labelled x in the original tree. Furthermore, by preprocessing T_x with an LCA data structure and labelling the nodes of T_x with their distance to the root, we can compute,

for any nodes u and v in T_x, the number of nodes labelled x on the path from u to v in T.

The difficulty now is that we can only do range counting queries between nodes u and v that occur in T_x and we need to answer these queries for any u and v in T. What we require is a mapping of the nodes of T onto corresponding nodes in T_x. More precisely, for each node v in T we need to be able to identify the first node labelled T_x encountered on the path from v to the root of T. Furthermore, we must be able to do this with a data structure whose size is related to the size of T_x, not T. Omitting the details, which can be found in the full version of the paper, we claim that this mapping can be achieved by performing an interval labelling of the nodes in T [11].

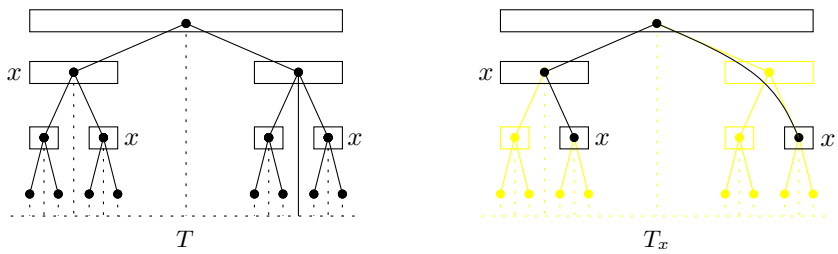

Fig. 2. The trees T and T_x and their interval labelling.

To summarize, we have described all the data structures needed to answer range counting queries in $O(\log n)$ time using a data structure of size $O(n)$. To answer a range mode query (u, v) we first lookup the two subtrees T_u and T_v of T that contain u and v as well as a mode $m_{u,v}$ of all the labels encountered on the path from T_u to T_v. We then perform range counting queries for each of the distinct labels in T_u and T_v as well as $m_{u,v}$ to determine an overall mode. The running time and storage requirements are identical to the data structure for lists.

Theorem 3 *For any $0 < \epsilon \leq 1/2$, there exists a data structure of size $O(n^{2-2\epsilon})$ that answers range mode queries on trees in $O(n^{\epsilon} \log n)$ time.*

4 Range Median Queries on Lists

In this section we consider the problem of answering range median queries on lists. To do this, we take the same general approach used to answer range mode queries. We perform a preprocessing of A so that our range median query reduces to the problem of computing the median of the union of several sets.

4.1 The Median of Several Sorted Sets

In this section we present three basic results that will be used in our range median data structures.

An *augmented* binary search tree is a binary search tree in which each node contains a *size* field that indicates the number of nodes in the subtree rooted at that node. This allows, for example, determining the rank of the root in constant time (it is the size of the left subtree plus 1) and indexing an element by rank in $O(\log n)$ time. Suppose we have three sets A, B, and C, stored in three augmented binary search trees T_A, T_B and T_C, respectively, and we wish find the element of rank i in $A \cup B \cup C$. The following lemma says that we can do this very quickly.

Lemma 2 *Let T_A, T_B, and T_C be three augmented binary search trees on the sets A, B, and C, respectively. There exists an $O(h_A + h_B + h_C)$ time algorithm to find the element with rank i in $A \cup B \cup C$, where h_A, h_B and h_C are the heights of T_A, T_B and T_C, respectively.*

Another tool we will make use of is a method of finding the median in the union of many sorted arrays.

Lemma 3 *Let $A_1, \ldots, A_k$ be sorted arrays whose total size is $O(n)$. There exists an $O(k \log n)$ time algorithm to find the element with rank i in $A_1 \cup \cdots \cup A_k$.*

Finally, we also make use of the following fact which plays a role analagous to that of Lemma 1.

Lemma 4 *Let A, B, and C be three sets such that $|A| = |C| = k$ and $|B| > 4k$. Then the median of $A \cup B \cup C$ is either in A, in C or is an element of B whose rank in B is in the range $[|B|/2 - 2k, |B|/2 + 2k]$.*

4.2 A First Time-Space Tradeoff

To obtain our first data structure for range median queries we proceed in a manner similar to that used for range mode queries. We partition our list A into b blocks $B_1, \ldots, B_b$ each of size n/b. We will create two types of data structures. For each block we will create a data structure that summarizes that block. For each pair of blocks we will create a data structure that summarizes all the elements between that pair of blocks.

To process each block we make use of *persistent augmented binary search trees*. These are search trees in which, every time an item is inserted or deleted, a new *version* of the tree is created. These trees are called persistent because they allow accesses to all previous versions of the tree. The simplest method of implementing persistent augmented binary search trees is by *path-copying* [6, 8–10, 13]. This results in $O(\log n)$ new nodes being created each time an element

is inserted or deleted, so a sequence of n update operations creates a set of n trees that are represented by a data structure of size $O(n \log n)$.[4]

For each block $B_{i'} = b_{i',1}, \ldots, b_{i',n/b}$, we create two persistent augmented search trees $\overrightarrow{T}_{i'}$ and $\overleftarrow{T}_{i'}$. To create $\overrightarrow{T}_{i'}$ we insert the elements $b_{i',1}, b_{i',2}, \ldots, b_{i',n/b}$ in that order. To create $\overleftarrow{T}_{i'}$ we insert the same elements in reverse order, i.e., we insert $b_{i',n/b}, b_{i',n/b-1}, \ldots, b_{i',1}$. Since these trees are persistent, this means that, for any j, $1 \leq j \leq n/b$, we have access to a search tree $\overrightarrow{T}_{i',j}$ that contains exactly the elements $b_{i',1}, \ldots, b_{i',j}$ and a search tree $\overleftarrow{T}_{i',j}$ that contains exactly the elements $b_{i',j}, \ldots, b_{i',n/b}$.

For each pair of blocks $B_{i'}$ and $B_{j'}$, $1 \leq i' < j' \leq n$, we sort the elements of $B_{i'+1} \cup \cdots \cup B_{j'-1}$ and store the elements whose ranks are within $2n/b$ of the median in a sorted array $A_{i',j'}$. Observe that, by Lemma 4, the answer to a range median query (i, j) where $i = i'n/b + x$ is in block i' and $j = j'n/b + y$ is in block j', is in one of $\overleftarrow{T}_{i',x}$, $A_{i',j'}$ or $\overrightarrow{T}_{j',y}$. Furthermore, given these two trees and one array, Lemma 2 allows us to find the median in $O(\log n)$ time.

Thus far, we have a data structure that allows us to answer any range median query (i, j) where i and j are in different blocks i' and j'. The size of the data structure for each block is $O((n/b) \log n)$ and the size of the data structure for each pair of blocks is $O(n/b)$. Therefore, the overall size of this data structure is $O(n(b + \log n))$. To obtain a data structure that answers queries for *any* range median query (i, j) including i and j in the same block, we build data structures recursively for each block. The size of all these data structures is given by the recurrence

$$T_n = bT_{n/b} + O(n(b + \log n)) = O(n(b + \log n) \log_b n) \ .$$

Theorem 4 *For any $1 \leq b \leq n$, there exists a data structure of size $O(n(b + \log n) \log_b n)$ that answers range median queries on lists in time $O(\log(n/b))$.*

At least asymptotically, the optimal choice of b is $b = \log n$. In this case, we obtain an $O(n \log^2 n / \log \log n)$ space data structure that answers queries in $O(\log n)$ time. In practice, the choice $b = 2$ is probably preferable since it avoids having to compute the $A_{i',j'}$ arrays altogether and only ever requires finding the median in two augmented binary search trees. The cost of this simplification is only an $O(\log \log n)$ factor in the space requirement.

4.3 A Constant Query Time Subquadratic Space Data Structure

In the full version of the paper we prove:

Theorem 5 *There exists a data structure of size $O(n^2 \log \log n / \log n)$ that can answer range median queries on lists in $O(1)$ time.*

[4] Although there are persistent binary search trees that require only $O(n)$ space for n operations [5, 12], these trees are not *augmented* and thus do not work in our application. In particular, they do not allow us to make use of Lemma 2.

4.4 A Data Structure Based on Range Trees

Using the method of *range trees* [7, 17] we can reduce the range median problem to a problem of finding the median of $O(b \log_b n)$ sorted arrays. Applying Lemma 3 we obtain the following theorem (details are in the full paper):

Theorem 6 *For any integer $1 \leq b \leq n$, there exists a data structure of size $O(n \log_b n)$ size that answers range median queries on lists in $O(b \log^2 n / \log b)$ time. In particular, for any constant $\epsilon > 0$ there exists a data structure of size $O(n)$ that answers range median queries in $O(n^\epsilon)$ time.*

5 Range Median Queries on Trees

Next we present two data structures for answering range median queries on trees. As with range mode queries, we may assume that T is a binary tree by doing a small amount of preprocessing on T. Details of this preprocessing are included in the full version.

5.1 More Space, Faster Queries

Our first method is essentially a binary version of the data structure of Section 4.2 modified to work on trees instead of lists. Details are included in the full version.

Theorem 7 *There exists a data structure of size $O(n \log^2 n)$ that can answer range median queries in trees in $O(\log n)$ time.*

5.2 Less Space, Slower Queries

There exists a method of decomposing a tree T into a set of paths such that the path between any two nodes u and v in T visits $O(\log n)$ of these paths [4]. By treating each of these paths as a list and using the range-tree method of Section 4.4 on each list we are able to obtain the following result (details are in the full version):

Theorem 8 *For any integer $1 \leq b \leq n$, there exists a data structure of size $O(n \log_b n)$ that can answer range median queries in trees in $O(b \log^3 n / \log b)$ time. In particular, for any constant $\epsilon > 0$, there exists a data structure of size $O(n)$ that answers range median queries in $O(n^\epsilon)$ time.*

Acknowledgement

The second author would like to thank Stefan Langerman for helpful discussions.

References

1. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings of Latin American Theoretical Informatics (LATIN 2000)*, pages 88–94, 2000.
2. O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proceedings of teh 21st Annual ACM Symposium on the Theory of Computing*, pages 309–319, 1989.
3. B. Chazelle. A theorem on polygon cutting with applications. In *In Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 339–349, 1982.
4. R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.
5. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.
6. T. Krijnen and L. G. L. T. Meertens. Making B-trees work for B. Technical Report 219/83, The Mathematical Center, Amsterdam, 1983.
7. G. S. Luecker. A data structure for orthogonal range queries. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pages 28–34, 1978.
8. E. W. Myers. AVL dags. Technical Report 82-9, Department of Computer Science, University of Arizona, 1982.
9. E. W. Myers. Efficient applicative data structures. In *Conference Record eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, 1984.
10. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5:449–477, 1983.
11. N. Santoro and R. Khatib. Labelling and implicit routing in networks. *The Computer Journal*, 1:5–8, 1985.
12. N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.
13. G. Swart. Efficient algorithms for computing geometric intersections. Technical Report #85-01-02, Department of Computer Science, University of Washington, Seattle, 1985.
14. M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.
15. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.
16. D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.
17. D. E. Willard. New data structures for orthogonal queries. *SIAM Journal on Computing*, pages 232–253, 1985.
18. A. C. Yao. Space-time tradeoff for answering range queries. In *Proceedings of the 14th Annual ACM Symposium on the Theory of Computing*, pages 128–136, 1982.
19. A. C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14:277–288, 1985.