RANGE MODE AND RANGE MEDIAN QUERIES ON LISTS AND TREES*

Danny Krizanc
Wesleyan University
dkrizanc@wesleyan.edu

Pat Morin
Carleton University
morin@cs.carleton.ca

Michiel Smid
Carleton University
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 analogous notion of $-$. In this case, Yao [16] shows how to preprocess a list A using $O(n\alpha_k(n))$ space so that queries can be answered in $O(k)$ 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 [17].

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 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

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

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.

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 [16, 17].

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.

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. Finally, in Section 6 we summarize and conclude with open problems.

The proofs of some lemmata used in this paper are fairly technical, and this can distract from the presentation of the data structures. Therefore we defer the proofs of all lemmata to Appendix A.

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 occurrences 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.

Throughout this section we will use the set union notation $A \cup B$ to denote the union *with multiplicity* of multisets A and B. In other words the number of occurrences of x in $A \cup B$ is equal to the number of occurrences of x in A plus the number of occurences of x in B. We will also slightly abuse this notation by applying it to lists and arrays. In this case, it denotes the union of the underlying multisets.

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 occurrences 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

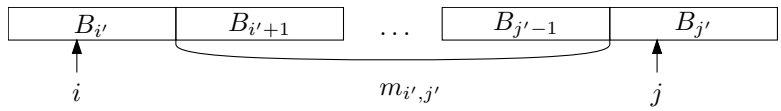

Figure 1: The mode of $a_i, \ldots, a_j$ is either an element of $B_{i'}$, an element of $B_{j'}$ or is the mode $m_{i',j'}$ of $B_{i'+1}, \ldots, B_{j'-1}$.

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:

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 $O(n^{\epsilon} \log n)$.*[1]

2.2 A Constant Query-Time Subquadratic Space Solution

At one extreme, Theorem 1 gives an $O(n)$ space, $O(\sqrt{n}\log n)$ query time data structure for range mode queries. Unfortunately, at the other extreme it gives an $O(n^2)$ space, $O(\log n)$ query time data structure. This is clearly non-optimal since with $O(n^2)$ space we could simply precompute the answer to each of the $\binom{n}{2}$ possible queries and then answer queries in constant time. In this section we show that it is possible to do even better than this by giving a data structure of subquadratic size that answers queries in constant time.

Let $k = n/b$ and consider any pair of blocks $B_{i'}$ and $B_{j'}$. There are k^2 possible range mode queries (i, j) such that i is in $B_{i'}$ and j is in $B_{j'}$. Each such query returns a result which is either an element of $B_{i'}$, an element of $B_{j'}$ or the mode of $B_{i'+1} \cup \cdots \cup B_{j'+1}$. Therefore, we could store the answers to all such queries in a table of size k^2, where each table entry is an integer in the range $0, \ldots, 2k$ that represents one of these $2k + 1$ possible outcomes. The total number of such tables is $(2k+1)^{k^2}$ and each table has size $O(k^2)$, so the total cost to store all such tables is only $O(k^2(2k+1)^{k^2})$. Therefore, if we choose $k = \sqrt{\log n / \log\log n}$, the total cost to store all these tables is only $O(n^2 \log\log n / \log n)$.

After computing all these tables, for each pair of blocks $B_{i'}$ and $B_{j'}$ we need only store a pointer to the correct table and the value of the mode $m_{i',j'}$ of $B_{i'+1} \cup \cdots \cup B_{j'-1}$. Then, for any range mode query with endpoints in $B_{i'}$ and $B_{j'}$ we need only perform a table lookup and use the integer result to report the mode either as an element of $B_{i'}$ an element of $B_{j'}$ or $m_{i',j'}$. The total size of this data structure is $O(b^2 + n) = O((n/k)^2 + n) = O(n^2 \log\log n / \log n)$.

To handle range mode queries (i, j) where i and j belong to the same block, we simply precompute all solutions to all possible queries where i and j are in the same block. The total space required for this is $O(bk^2) = O(n \log^c n)$ which is much smaller than the space already used.

[1] 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 [13, 12, 14], 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.

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 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 a tree 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

$$x(u) + x(v) - 2x(w) + \begin{cases} 1 & \text{if label of } w \text{ is } 1 \\ 0 & \text{otherwise} \end{cases}$$

(see Fig. 2).

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. 3). It is clear that the tree T_x has size proportional to the number of

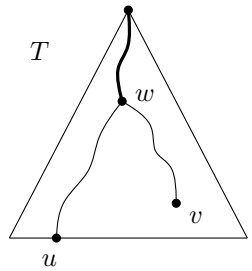

Figure 2: The number of nodes labelled x on the path from u to v can be computed from $x(u)$, $x(v)$ and $x(w)$.

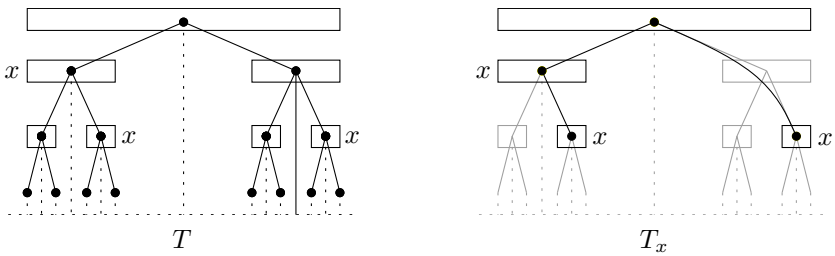

Figure 3: The trees T and T_x and their interval labelling.

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.

To achieve this mapping, we perform an *interval labelling* of the nodes in T (see Fig. 3): We label the nodes of T with consecutive integers by an in-order traversal of T. With each internal node v of T, we assign the minimum interval that contains all of the integer labels in the subtree rooted at v. Note that every node in T_x is also a node in T, so this also gives an interval labelling of the corresponding nodes in T_x (although the intervals are not minimal). Consider a node v of T whose integer label is g. Then it is easy to verify that the first node labelled x on the path from v to the root of T is the node of T_x with the smallest interval label that contains g. Next, observe that if we sort the endpoints of these intervals then in any subinterval defined by two consecutive endpoints the answer to a query is the same. Therefore, by sorting the endpoints of the intervals of nodes in T_x and storing these in a sorted array we can answer these queries in $O(\log n)$ time using a data structure of size $O(|T_x|)$.

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.

Throughout this section (and the proofs of Lemmata 2-4 in Appendix A) we will assume that the input $A = a_1, \ldots, a_n$ consists of n distinct values. If this is not the case, we can easily make the elements distinct by mapping the element a_i onto the ordered pair (a_i, i) and replacing the standard comparison operator with lexigraphic comparison.

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 to 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 analogous to that of Lemma 1.

Lemma 4. *Let A, B, and C be three sets such that $|A| = |C| = k$ and $|B| \geq 2k$. 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 $[\lfloor |B|/2 \rfloor - k, \lceil |B|/2 \rceil + k]$.*

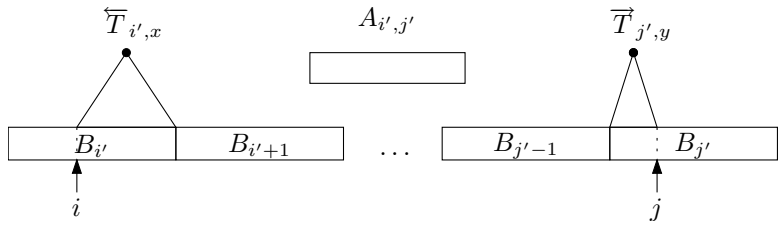

Figure 4: The median of $a_i, \ldots, a_j$ can be computed from two persistent search trees.

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* [5, 7, 8, 9, 11]. 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 $O(n \log n)$.[2]

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 $\overrightarrow{T}_{j',y}$ (see Fig. 4). 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

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

[2]Although there are persistent binary search trees that require only $O(n)$ space for n operations [4, 10], 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.

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

Next we sketch a range median query data structure with constant query time and subquadratic space. The data structure is essentially the same as the range mode query data structure described in Section 2.2 modified to perform median queries. The modifications are as follows: For each pair of blocks $B_{i'}$ and $B_{j'}$ we need only consider the set of $6k$ elements that are potential medians of queries with endpoints i and j in $B_{i'}$ and $B_{j'}$. We can also create a normalized version of these elements, so that each element is a unique integer in the range $1, \ldots, 6k$. In this way, we only need to create $(6k)!$ different lookup tables, each of size $O(k^2)$.

To summarize, storing all the lookup tables takes $O(k^2(6k)!)$ space. For each pair of blocks we must store a pointer to a lookup table as well as an array of size $6k$ that translates ranks in the lookup table to elements of A, for a total space of $O(b^2k)$. For each block we precompute and store all the solutions to queries with both endpoints in that block using a table of size $O(bk^2)$. The total space required by all the pieces of this data structure is

$$O(k^2(6k)! + b^2k + bk^2) = O(k^2(6k)! + n^2/k + nk)$$

Setting $k = c\log n/\log\log n$ for sufficiently small c, we obtain an overall space bound of $O(n^2 \log\log n/\log n)$.

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.*

4.4 A Data Structure Based on Range Trees

Next we describe a range median data structure based on the same principle as Lueker and Willard's range trees [6, 15]. This data structure stores $a_1, \ldots, a_n$ at the leaves of a complete b-ary tree T in the order in which they appear in A. At each internal node v of this tree we keep a sorted array containing all the elements of A that appear at leaves in the subtree rooted at v. It is clear that this tree, including the arrays stored at all the nodes, has size $O(n \log_b n)$.

To use this tree to answer a range query (i, j), consider the two paths P_i and P_j from the root of T to the leaf containing a_i and the leaf containing a_j, respectively (see Fig. 5). These two paths share some nodes for a period of time and then diverge. Observe that, after this point, by looking at the sorted arrays at nodes to the right of P_i and to the left of P_j we obtain a partition of $a_i, \ldots, a_j$ into a set of sorted arrays. The number of these arrays is at most $b\log_b n$ and their total size is at most n. Therefore, by Lemma 3 we can answer the range median query (i, j) in $O(b\log^2 n/\log b)$ time.

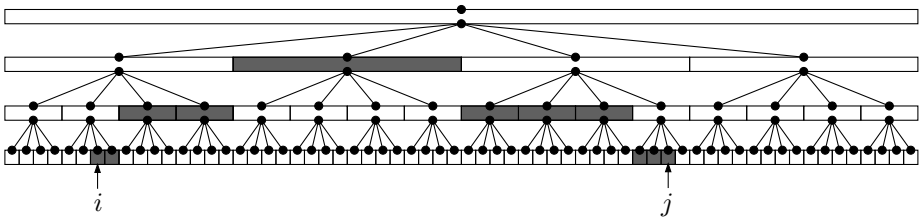

Figure 5: Using range trees to perform range median queries. The median of $a_i, \ldots, a_j$ is the median of the elements in the $O(b \log_b n)$ shaded arrays.

Theorem 6. *For any integer $1 \leq b \leq n$, there exists a data structure of size $O(n \log_b n)$ 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 converting nodes with $d > 2$ children into complete binary trees. In these little trees we subdivide edges to ensure that the number of internal nodes in any root to leaf path is even and label these nodes alternately with $-\infty$, $+\infty$ so as not to affect the median on any path between two of the original nodes of T.

5.1 More Space, Faster Queries

Our method is simply the binary version of the basic method in Section 4.2 for lists. We first find a centroid edge (a, b) of T whose removal partitions T into two subtrees T_a and T_b each of size at most $2/3$ the original size of T. For each node u in T_a, we would like to have access to an augmented search tree that contains exactly the labels on the path from u to a. To achieve this, we proceed as follows: To initialize the algorithm we insert the label of a into a persistent augmented binary search tree, mark a and define this new tree to be the *tree of* a. While some marked node u of T_a has an unmarked child v, we insert the label of v into the tree of u, mark v, and define this new tree to be the tree of v. Note that because we are using persistent search trees, this leaves the tree of u unchanged. In this way, for any node u in T_a, the tree of u contains exactly the labels of the nodes on the path from u to a. We repeat the same procedure for T_b, and this creates a data structure of size $O(n \log n)$.

To answer a range median query (u, v) where u is in T_a and v is in T_b, we only need to find the median of all labels stored in the tree of a and the tree of b. By Lemma 2 this can be done in $O(\log n)$ time. To answer range median queries (u, v) where both u and v are in T_a (or T_b) we recursively build data structures for range median queries in T_a and T_b. The total size of all these data structures is

$$T_n = T_{\alpha n} + T_{(1-\alpha)n} + O(n \log n) = O(n \log^2 n) \ ,$$

where $1/3 \leq \alpha \leq 2/3$ and they can answer range median queries in $O(\log n)$ time.

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.*

It is tempting to try and shave a $\log \log n$ factor off the storage requirement of Theorem 7 by using a $\log n$-ary version of the above scheme as we did in Section 4.2. However, the reason this worked for lists is that, for any block, a query either extends to the left or right boundary of that block, so only two persistent search trees are needed. However, if we try to make a $\log n$-ary partition of a tree we find that each subtree (block) can have $\Omega(\log n)$ vertices that share an edge with another subtree, which would require $\Omega(\log n)$ persistent search trees per subtree.

5.2 Less Space, Slower Queries

We now describe a data structure that uses less space. For any node u of T, let $size(u)$ denote the number of leaves in the subtree of u, and let $\ell(u) = \lfloor \log size(u) \rfloor$. Observe that $\ell(u)$ is an integer in the range from zero to $\lfloor \log n \rfloor$. Moreover, on the path from any leaf to the root of T, the $\ell(u)$-values form a non-decreasing sequence.

The values $\ell(u)$ can be used to partition T into a collection of pairwise disjoint paths: For any node u of T, let P_u be the subgraph of T consisting of all nodes v for which (i) $\ell(v) = \ell(u)$, and (ii) $\ell(w) = \ell(u)$ for all nodes w on the path in T between u and v. It is not difficult to see that P_u is a path; we call this path a *piece*.

We extend the tree T by storing with each node u the value of $size(u)$, a pointer to the parent of u, and a pointer to the node $gpar(u)$ of P_u that is closest to the root. We call $gpar(u)$ the *group parent* of u. Using these pointers, we can walk from any node u to any other node v in $O(\log n)$ time. During this walk, we visit only $O(\log n)$ pieces.

For each piece, we build the data structure of Section 4.4. Therefore, the entire data structure has size $O(n \log_b n)$. To answer a query (u, v), we determine the $O(\log n)$ pieces encountered when walking in T from u to v. The path in T between u and v visits a group of consecutive nodes on each such piece. Using the data structure of such a piece, we partition this group into $O(b \log_b n)$ sorted arrays. Hence, overall, we have to find the median in a sequence of $O(b \log^2 n / \log b)$ sorted arrays which takes, by Lemma 3 $O(b \log^3 n / \log b)$ time.

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.*

6 Summary and Conclusions

We have given data structures for answering range mode and range median queries on lists and trees. To the best of our knowledge, we are the first to study these problems. These problems do not seem to admit the same techniques used to develop optimal data structures for range queries involving group or semigroup operators.

Essentially every result in this paper is an open problem. There are no lower bounds for these problems and it seems unlikely that any of our data structures are optimal. Thus, there is still a significant amount of work to be done on these problems, either by improving these results and/or showing non-trivial lower bounds for these problems.

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 the 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] 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.

[5] T. Krijnen and L. G. L. T. Meertens. Making B-trees work for B. Technical Report 219/83, The Mathematical Center, Amsterdam, 1983.

[6] G. S. Lueker. A data structure for orthogonal range queries. In *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science*, pages 28–34, 1978.

[7] E. W. Myers. AVL dags. Technical Report 82-9, Department of Computer Science, University of Arizona, 1982.

[8] E. W. Myers. Efficient applicative data structures. In *Conference Record eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, 1984.

[9] 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.

[10] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

[11] G. Swart. Efficient algorithms for computing geometric intersections. Technical Report #85-01-02, Department of Computer Science, University of Washington, Seattle, 1985.

[12] M. Thorup. On RAM priority queues. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, 1996.

[13] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[14] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[15] D. E. Willard. New data structures for orthogonal queries. *SIAM Journal on Computing*, pages 232–253, 1985.

[16] 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.

[17] A. C. Yao. On the complexity of maintaining partial sums. *SIAM Journal on Computing*, 14:277–288, 1985.

A Proofs of Lemmata

Proof of Lemma 1. Suppose, for the sake of contradiction, that the mode x of $A \cup B \cup C$ does not appear in A or C and is not a mode of B. But then there is a mode $y \in B$ of B that occurs more frequently in B than x. Since x does not occur in A or C then y also occurs more frequently in $A \cup B \cup C$ than x, so x is not a mode of $A \cup B \cup C$, a contradiction. □

The next three proofs deal with medians. For a set X and an element $x \in X$, we define $rank_X(x)$ to be the rank of x in the set X, i.e., the number of elements $y \in X$ such that $y \leq x$.

Proof of Lemma 2. We describe an algorithm to find the element of rank i in $A \cup B \cup C$ using T_A, T_B and T_C. Let r_A, r_B and r_C denote the values stored at the roots of T_A, T_B and T_C and assume, without loss of generality that $r_A < r_B < r_C$. Let n_A, n_B and n_C denote the sizes of the left subtrees of T_A, T_B and T_C respectively. There are three cases to consider:

1. $n_A + n_B + n_C < i - 1$. In this case, the rank of r_A in $A \cup B \cup C$ is less than i and the ranks of all elements stored in the left subtree of T_A are also less than i. Therefore, we can safely proceed by searching for the element of rank $i - n_A - 1$ in the three trees T_B, T_C and the right subtree of T_A.
2. $n_A + n_B + n_C = i - 1$. In this case, the rank of r_A in $A \cup B \cup C$ is i, and we are done searching.
3. $n_A + n_B + n_C > i - 1$. In this case, the rank of r_C in $A \cup B \cup C$ is at least $n_A + n_B + n_C + 2 > i + 1$ and the same is true of all elements in the right subtree of T_C. Therefore, we can safely proceed by searching for the element of rank i in the three trees T_A, T_B and the left subtree of T_C.

We can continue in this manner until one of T_A, T_B and T_C is empty in which case a slightly modified version of the above algorithm will find the element of rank i in two trees. Each iteration of this algorithm takes constant time and either finds the element of rank i or reduces the height of one of the subtrees by 1, so the total running time is $O(h_A + h_B + h_C)$, as required. □

Proof of Lemma 3. Let $n_1, \ldots, n_k$ denote the sizes of $A_1, \ldots, A_k$, respectively, and let $A = A_1 \cup \cdots \cup A_k$. We are looking for an $O(k \log n)$ time algorithm to find the element of rank i in A. Without loss of generality we may assume that $i \leq n/2$, otherwise we can reverse the roles of i and $n - i$ as well the roles of "greater than" and "less than."

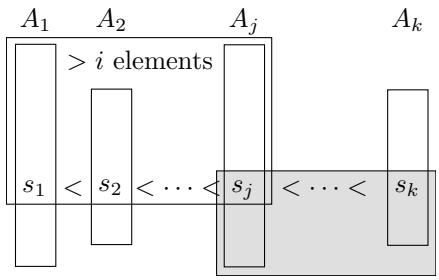

Figure 6: When searching for the element of rank i we can discard all elements in the shaded area.

Consider the set $S = \{s_1, \ldots, s_k\}$ where s_j is the element of A_j such that $rank_{A_j}(s_j) = r_j$ and

$$r_j = \left\lceil \frac{2}{3} \times n_j \right\rceil$$

For ease of notation, we will assume that $s_1 < s_2 < \cdots < s_k$ though our algorithm will not make use of this assumption. Refer to Fig. 6.

Let j be the smallest index such that $\sum_{\ell=1}^{j} r_\ell > i$ so we are guaranteed that $rank_A(s_j) > i$. It is clear that such a value of j exists since $\sum_{\ell=1}^{k} r_j \geq 2n/3$. Finding the index j is a *weighted selection* operation. This operation can be performed in $O(k)$ time as follows: Find the median $s_{\lfloor k/2 \rfloor}$ of $s_1, \ldots, s_k$ in $O(k)$ time using any linear-time median finding algorithm. If $\sum_{\ell=1}^{\lfloor k/2 \rfloor} r_\ell \leq i$ then recursively search for s_j in the set $s_1, \ldots, s_{\lfloor k/2 \rfloor - 1}$. Otherwise, recursively search for s_j in the set $s_{\lfloor k/2 \rfloor}, \ldots, s_k$. The running time of this algorithm is given by the classic reccurence $T_k = O(k) + T_{k/2}$ which solves to $O(k)$.

Observe that, since $rank_A(s_j) > i$, any element greater than or equal to s_j can be excluded from our search. This includes all the elements $s_j, \ldots, s_k$ and parts of their respective arrays $A_j, \ldots, A_k$ (the shaded region in Fig. 6). How much does this allow us to discard? We know that $\sum_{\ell=1}^{j-1} r_\ell < i$, so

$$\sum_{\ell=1}^{j-1} n_\ell \leq \frac{3}{2} \sum_{\ell=1}^{j-1} r_\ell \leq 3i/2 \leq 3n/4 \ .$$

Therefore,

$$\sum_{\ell=j}^{k} n_\ell = n - \sum_{\ell=1}^{j-1} n_\ell \geq n/4 \ .$$

But from each array A_ℓ, $j \leq \ell \leq k$ we discard a piece of size at least

$$d_\ell \geq n_\ell - r_\ell + 1 \geq n_\ell/3 \ .$$

Therefore, the total amount we discard is at least

$$\sum_{\ell=j}^{k} d_\ell \geq \sum_{\ell=j}^{k} n_\ell/3 \geq n/12 \ .$$

Therefore, after iterating this process at most $\log_{\frac{12}{11}} n$ we will have found the element of rank i. Since each iteration takes $O(k)$ time this results in a total running time of $O(k \log n)$, as required. □

Proof of Lemma 4. Let b_1 be the element of B such that $rank_B(b_1) = \lfloor |B|/2 \rfloor - k$ and let b_2 be the element of B such that $rank_B(b_2) = \lceil |B|/2 \rceil + k$. Since A and C each contain only k elements, it follows that

$$rank_{A \cup B \cup C}(b_1) \leq rank_B(b_1) + 2k = \lfloor |B|/2 \rfloor + k$$

and similarly

$$rank_{A \cup B \cup C}(b_2) \geq rank_B(b_2) = \lceil |B|/2 \rceil + k \ .$$

Observe that the median m of $A \cup B \cup C$ satisfies $rank_{A \cup B \cup C}(m) = \lfloor |B|/2 + k \rfloor$. However, if we choose any $x \in B$ such that $x < b_1$ then it follows that

$$rank_{A \cup B \cup C}(x) < \lfloor |B|/2 \rfloor + k = \lfloor |B|/2 + k \rfloor \ .$$

Similarly, if $x > b_2$ then

$$rank_{A \cup B \cup C}(x) > \lceil |B|/2 \rceil + k > \lfloor |B|/2 + k \rfloor \ .$$

In either case, x is definitely not the median of $A \cup B \cup C$, so the only elements of B that can be the median are those elements y such that $b_1 \leq y \leq b_2$, as required. □