Lazy Search Trees

Bryce Sandlund
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
Email: bcsandlund@gmail.com

Sebastian Wild
Department of Computer Science
University of Liverpool
Liverpool, UK
Email: wild@liverpool.ac.uk

***Abstract*—We introduce the lazy search tree data structure. The lazy search tree is a comparison-based data structure on the pointer machine that supports order-based operations such as rank, select, membership, predecessor, successor, minimum, and maximum while providing dynamic operations insert, delete, change-key, split, and merge. We analyze the performance of our data structure based on a partition of current elements into a set of *gaps* $\{\Delta_i\}$ based on rank. A query falls into a particular gap and *splits* the gap into two new gaps at a rank r associated with the query operation. If we define $B = \sum_i |\Delta_i| \log_2(n/|\Delta_i|)$, our performance over a sequence of n insertions and q distinct queries is $O(B + \min(n \log\log n, n \log q))$. We show B is a lower bound.**

Effectively, we reduce the insertion time of binary search trees from $\Theta(\log n)$ to $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$, where Δ_i is the gap in which the inserted element falls. Over a sequence of n insertions and q queries, a time bound of $O(n \log q + q \log n)$ holds; better bounds are possible when queries are non-uniformly distributed. As an extreme case of non-uniformity, if all queries are for the minimum element, the lazy search tree performs as a priority queue with $O(\log\log n)$ time insert and decrease-key operations. The same data structure supports queries for *any* rank, interpolating between binary search trees and efficient priority queues.

Lazy search trees can be implemented to operate mostly on arrays, requiring only $O(\min(q, n))$ pointers, suggesting smaller memory footprint, better constant factors, and better cache performance compared to many existing efficient priority queues or binary search trees. Via direct reduction, our data structure also supports the efficient access theorems of the splay tree, providing a powerful data structure for non-uniform element access, both when the number of accesses is small and large.

I. Introduction

We consider data structures supporting order-based operations such as rank, select, membership, predecessor, successor, minimum, and maximum while providing dynamic operations insert, delete, change-key, split, and merge. The classic solution is the binary search tree (BST), perhaps the most fundamental data structure in computer science. The original unbalanced structure dates back to (at least) the early 1960's; a plethora of *balanced* binary search tree data structures have since been proposed, notable examples including AVL trees [1], red-black trees [2], and splay trees [3]. A balanced binary search tree is a staple data structure included in nearly all major programming language's standard libraries and nearly every undergraduate computer science curriculum. The data structure is the dynamic equivalent of binary search in an array, allowing searches to be performed on a changing set of keys at nearly the same cost. Extending to multiple dimensions, the binary search tree is the base data structure on which range trees, segment trees, interval trees, kd-trees, and priority search trees are all built [4].

The theory community has long focused on developing binary search trees with efficient *query* times. Although $\Omega(\log n)$ is the worst-case time complexity of a query, on non-uniform access sequences binary search trees can perform better than logarithmic time per query by, for example, storing recently accessed elements closer to the root. The splay tree was devised as a particularly powerful data structure for this purpose [3], achieving desirable access theorems such as static optimality, working set, scanning theorem, static finger, and dynamic finger [3], [5], [6]. The most famous performance statement about the splay tree, however, is the unproven dynamic optimality conjecture, which claims that the performance of the splay tree is within a constant factor of any binary search tree on any sufficiently long access sequence, subsuming all other access theorems. Proving this conjecture is widely considered one of the most important open problems in theoretical computer science, receiving vast attention by data structure researchers [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. Despite ultimately remaining unsolved for nearly four decades, this topic continues to receive extensive treatment [10], [11], [15], [17], [18].

Although widely considered for the task in literature, the binary search tree is not the most efficient data structure for the standard dictionary abstract data type: in practice, dictionaries are almost always implemented by hash tables, which support $O(1)$ time insert, delete, and look-up in expectation [19], [20]. The advantage of binary search trees, over hash tables, is that they support *order-based* operations. We call dictionaries of this type *sorted dictionaries*, to differentiate them from the simpler data structures supporting only membership queries.

The full version of this extended abstract is available on arXiv.

2575-8454/20/$31.00 ©2020 IEEE
DOI 10.1109/FOCS46700.2020.00071

If we limit the order-based operations required of our sorted dictionary to queries for the minimum or maximum element (or both), a number of alternative solutions to the binary search tree have been developed, known as priority queues. The first of which was the binary heap, invented in 1964 for the heapsort algorithm [21]. The binary heap achieves asymptotic complexity equivalent to a binary search tree, though due to the storage of data in an array and fast average-case complexity, it is typically the most efficient priority queue in practice. Later, the invention of the binomial heap showed that the merging of two arbitrary priority queues could be supported efficiently [22], [23], thus proving that the smaller operation set of a priority queue allows more efficient runtimes. The extent to which priority queues can outperform binary search trees was fully realized with the invention of Fibonacci heaps, which showed insertion, merge, and an additional decrease-key operation can all be supported in $O(1)$ amortized time [24]. Since then, a number of priority queues with running times close to or matching Fibonacci heaps have been developed [25], [26], [27], [28], [29], [30], [31]. We refer to such priority queues with $o(\log n)$ insertion and decrease-key costs as *efficient* priority queues, to distinguish them from their predecessors and typically simpler counterparts with $O(\log n)$ insertion and/or decrease-key cost.

The history of efficient priority queues contrasts that of binary search trees. Efficient priority queues have been developed for the case when the number of queries is significantly less than the number of insertions or updates. On the other hand, research on binary search trees has focused on long sequences of element *access*. Indeed, the dynamic optimality conjecture starts with the assumption that n elements are already present in the binary search tree, placing any performance improvements by considering insertion cost entirely outside of the model. However, the theory of efficient priority queues shows that on some operation sequences, the efficiency gains due to considering insertion cost can be as much as a $\Theta(\log n)$ factor, showing an as-of-yet untapped area of potential optimization for data structures supporting the operations of a binary search tree. Further, aside from the theoretically-appealing possibility of the unification of the theories of efficient priority queues and binary search trees, the practicality of improved insertion performance is arguably greater than that of improved access times. For the purpose of maintaining keys in a database, for example, an insert-efficient data structure can provide superior runtimes when the number of insertions dominates the number of queries, a scenario that is certainly the case for some applications [32], [33] and is, perhaps, more likely in general. Yet in spite of these observations, almost no research has been conducted that seriously explores this frontier [34].

We attempt to bridge this gap. We seek a general theory of comparison-based sorted dictionaries that encompasses efficient priority queues and binary search trees, providing the operational flexibility of the latter with the efficiency of the former, when possible. We do not restrict ourselves to any particular BST or heap model; while these models with their stronger lower bounds are theoretically informative, for the algorithm designer these lower bounds in artificially constrained models are merely indications of what *not* to try. If we believe in the long-term goal of improving algorithms and data structures in practice – an objective we think will be shared by the theoretical computer science community at large – we must also seek the comparison with lower bounds in a more permissive model of computation.

We present *lazy search trees*. The lazy search tree is the first data structure to support the general operations of a binary search tree while providing superior insertion time when permitted by query distribution. We show that the theory of efficient priority queues can be generalized to support queries for any rank, via a connection with the multiple selection problem. Instead of sorting elements upon insertion, as does a binary search tree, the lazy search delays sorting to be completed incrementally while queries are answered. A binary search tree and an efficient priority queue are special cases of our data structure that result when queries are frequent and uniformly distributed or only for the minimum or maximum element, respectively. While previous work has considered binary search trees in a "lazy" setting (known as "deferred data structures") [35], [36] and multiple selection in a dynamic setting [37], [38], no existing attempts fully distinguish between insertion and query operations, severely limiting the generality of their approaches. The model we consider gives all existing results as corollaries, unifying several research directions and providing more efficient runtimes in many cases, all with the use of a single data structure.

A. Model and Results

We consider comparison-based data structures on the pointer machine. While we suggest the use of arrays in the implementation of our data structure in practice, constant time array access is not needed for our results. Limiting operations to a pointer machine has been seen as an important property in the study of efficient priority queues, particularly in the invention of strict Fibonacci heaps [27] compared to an earlier data structure with the same worst-case time complexities [30].

We consider data structures supporting the following operations on a dynamic multiset S with (current) size $n = |S|$. We call such data structures *sorted dictionaries*:

- `Construction(`S`)` := Construct a sorted dictionary on the set S.
- `Insert(`e`)` := Add element $e = (k, v)$ to S, using key k for comparisons; (this increments n).
- `RankBasedQuery(`r`)` := Perform a rank-based query pertaining to rank r on S.

- `Delete(ptr)` := Delete the element pointed to by `ptr` from S; (this decrements n).
- `ChangeKey(ptr,` k'`)` := Change the key of the element pointed to by `ptr` to k'.
- `Split(`r`)` := Split S at rank r, returning two sorted dictionaries T_1 and T_2 of r and $n - r$ elements, respectively, such that for all $x \in T_1$, $y \in T_2$, $x \leq y$.
- `Merge(`T_1`,` T_2`)` := Merge sorted dictionaries T_1 and T_2 and return the result, given that for all $x \in T_1$, $y \in T_2$, $x \leq y$.

We formalize what queries are possible within the stated operation `RankBasedQuery(`r`)` in the full version of the paper. For now, we informally define a rank-based query as any query computable in $O(\log n)$ time on a (possibly augmented) binary search tree and in $O(n)$ time on an unsorted array. Operations rank, select, contains, successor, predecessor, minimum, and maximum fit our definition. To each operation, we associate a rank r: for membership and rank queries, r is the rank of the queried element (in the case of duplicate elements, an implementation can break ties arbitrarily), and for select, successor, and predecessor queries, r is the rank of the element returned; minimum and maximum queries are special cases of select with $r = 1$ and $r = n$, respectively.

The idea of lazy search trees is to maintain a partition of current elements in the data structure into what we will call *gaps*. We maintain a set of m gaps $\{\Delta_i\}$, $1 \leq i \leq m$, where a gap Δ_i contains a bag of elements. Gaps satisfy a total order, so that for any elements $x \in \Delta_i$ and $y \in \Delta_{i+1}$, $x \leq y$. Internally, we will maintain structure within a gap, but the interface of the data structure and the complexity of the operations is based on the distribution of elements into gaps, assuming nothing about the order of elements within a gap. Intuitively, binary search trees fit into our framework by restricting $|\Delta_i| = 1$, so each element is in a gap of its own, and we will see that priority queues correspond to a single gap Δ_1 which contains all elements. Multiple selection corresponds to gaps where each selected rank marks a separation between adjacent gaps.

To insert an element $e = (k, v)$, where k is its key and v its value, we find a gap Δ_i in which it belongs without violating the total order of gaps (if $x \leq k$ for all $x \in \Delta_i$ and $k \leq y$ for all $y \in \Delta_{i+1}$, we may place e in either Δ_i or Δ_{i+1}; implementations can make either choice). Deletions remove an element from a gap; if the gap is now empty we can remove the gap. When we perform a query, we first narrow the search down to the gap Δ_i in which the query rank r falls (formally, $\sum_{j=1}^{i-1} |\Delta_j| < r \leq \sum_{j=1}^{i} |\Delta_j|$). We then answer the query using the elements of Δ_i and *restructure* the gaps in the process. We split gap Δ_i into two gaps Δ_i' and Δ_{i+1}' such that the total order on gaps is satisfied and the rank r element is either the largest in gap Δ_i' or the smallest in gap Δ_{i+1}'; specifically, either $|\Delta_i'| + \sum_{j=1}^{i-1} |\Delta_j| = r$ or $|\Delta_i'| + \sum_{j=1}^{i-1} |\Delta_j| = r - 1$. (Again, implementations can take either choice. We will assume a maximum query to take the latter choice and all other queries the former. More on the choice of r for a given query is discussed in the full version of the paper. Our analysis will assume two new gaps replace a former gap as a result of each query. Duplicate queries or queries that fall in a gap of size one follow similarly, in $O(\log n)$ time.) We allow duplicate insertions.

Our performance theorem is the following.

Theorem 1 (Lazy search tree runtimes)**.** *Let n be the total number of elements currently in the data structure and let $\{\Delta_i\}$ be defined as above (thus $\sum_{i=1}^{m} |\Delta_i| = n$). Let q denote the total number of queries. Lazy search trees support the operations of a sorted dictionary on a dynamic set S in the following runtimes*:

- `Construction(`S`)` *in $O(|S|)$ worst-case time.*
- `Insert(`e`)` *in $O(\min(\log(n/|\Delta_i|) + \log\log|\Delta_i|, \log q))$ worst-case time[1], where $e = (k, v)$ is such that $k \in \Delta_i$.*
- `RankBasedQuery(`r`)` *in $O(x \log c + \log n)$ amortized time, where the larger resulting gap from the split is of size cx and the other gap is of size x.*
- `Delete(ptr)` *in $O(\log n)$ worst-case time.*
- `ChangeKey(ptr,` k'`)` *in $O(\min(\log q, \log\log|\Delta_i|))$ worst-case time, where the element pointed to by* `ptr`*, $e = (k, v)$, has $k \in \Delta_i$ and k' moves e closer to its closest query rank[2] in Δ_i; otherwise,* `ChangeKey(ptr,` k'`)` *takes $O(\log n)$ worst-case time.*
- `Split(`r`)` *in time according to* `RankBasedQuery(`r`)`.
- `Merge(`T_1`,` T_2`)` *in $O(\log n)$ worst-case time.*

Define $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. Then over a series of insertions and queries with no duplicate queries, the total complexity is $O(B + \min(n \log\log n, n \log q))$.

We can also bound the number of pointers needed in the data structure.

Theorem 2 (Pointers)**.** *An array-based lazy search tree implementation requires $O(\min(q, n))$ pointers.*

By reducing multiple selection to the sorted dictionary problem, we can show the following lower bound.

Theorem 3 (Lower bound)**.** *Suppose we process a sequence of operations resulting in gaps $\{\Delta_i\}$. Again define $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. Then this sequence of operations requires $B - O(n)$ comparisons and $\Omega(B + n)$ time in the worst case.*

Theorem 3 indicates that lazy search trees are at most an additive $O(\min(n \log\log n, n \log q))$ term from optimality over a series of insertions and distinct queries.

[1] To simplify formulas, we distinguish between $\log_2(x)$, the binary logarithm for any $x > 0$, and $\log(x)$, which we define as $\max(\log_2(x), 1)$.

[2] The closest query rank of e is the closest boundary of Δ_i that was created in response to a query.

This gives a lower bound on the per-operation complexity of `RankBasedQuery(r)` of $\Omega(x \log c)$; the bound can be extended to $\Omega(x \log c + \log n)$ if we amortize the total work required of splitting gaps to each individual query operation. A lower bound of $\Omega(\min(\log(n/|\Delta_i|), \log m))$ can be established on insertion complexity via information theory. We describe all lower bounds in Section IV.

We give specific examples of how lazy search trees can be used and how to analyze its complexity according to Theorem 1 in the following subsection.

B. Example Scenarios

Below, we give examples of how the performance of Theorem 1 is realized in different operation sequences. While tailor-made data structures for many of these applications are available, lazy search trees provide a *single* data structure that seamlessly adapts to the actual usage pattern while achieving optimal or near-optimal performance for all scenarios in a uniform way.

Few Queries: The bound $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$ satisfies $B = O(n \log q + q \log n)$. In the worst case, queries are uniformly distributed, and the lower bound $B = \Theta(n \log q + q \log n)$. Over a sequence of insertions and queries without duplicate queries, our performance is optimal $O(n \log q + q \log n)$. If $q = n^{\epsilon}$ for constant $\epsilon > 0$, lazy search trees improve upon binary search trees by a factor $1/\epsilon$. If $q = O(\log^c n)$ for some c, lazy search trees serve the operation sequence in $O(cn \log \log n)$ time and if $q = O(1)$, lazy search trees serve the operation sequence in linear time. Although it is not very difficult to modify a previous "deferred data structure" to answer a sequence of n insertions and q queries in $O(n \log q + q \log n)$ time (see Section II-A), to the best of our knowledge, such a result has not appeared in the literature.

Clustered Queries: Suppose the operation sequence consists of q/k "range queries", each requesting k consecutive keys, with interspersed insertions following a uniform distribution. Here, $B = O(n \log(q/k) + q \log n)$, where q is the total number of keys requested. If the queried ranges are uniformly distributed, $B = \Theta(n \log(q/k) + q \log n)$, with better results possible if the range queries are non-uniform. Our performance on this operation sequence is $O(B + \min(n \log \log n, n \log q))$, tight with the lower bound if $k = \Theta(1)$ or $q/k = \Omega(\log n)$. Similarly to Scenario 1, we pay $O(n \log(q/k))$ in total for the first query of each batch; however, each successive query in a batch costs only $O(\log n)$ time as the smaller resulting gap of the query contains only a single element. We will see in Section IV that we must indeed pay $\Omega(\log n)$ amortized time per query in the worst case; again our advantage is to reduce insertion costs. Note that if an element is inserted within the elements of a previously queried batch, these insertions take $O(\log n)$ time. However, assuming a uniform distribution of element insertion throughout, this occurs on only an $O(q/n)$ fraction of insertions in expectation, at total cost $O(n \cdot q/n \cdot \log n) = O(q \log n)$. Other insertions only need an overall $O(n \log(q/k) + \min(n \log \log n, n \log q))$ time.

Selectable Priority Queue: If every query is for a minimum element, each query takes $O(\log n)$ time and separates the minimum element into its own gap and all other elements into another single gap. Removal of the minimum destroys the gap containing the minimum element, leaving the data structure with a single gap Δ_1. All inserted elements fall into this single gap, implying insertions take $O(\log \log n)$ time. Further, the `ChangeKey(ptr,` k'`)` operation supports decrease-key in $O(\log \log n)$ time, since all queries (and thus the closest query) are for rank 1. Queries for other ranks are also supported, though if queried, these ranks are introduced into the analysis, creating more gaps and potentially slowing future insertion and decrease-key operations, though speeding up future selections. The cost of a selection is $O(x \log c + \log n)$ amortized time, where x is the distance from the rank selected to the nearest gap boundary (which was created at the rank of a previous selection) and $c = |\Delta_i|/x - 1$, where the selection falls in gap Δ_i. If no selections have previously occurred, x is the smaller of the rank or n minus the rank selected and $c = n/x - 1$.

Interestingly, finding the kth smallest element in a binary min-heap can be done in $O(k)$ time [39], yet we claim our runtime optimal! The reason is that neither runtime dominates in an amortized sense over the course of n insertions. Our lower bound indicates that $\Omega(B + n)$ time must be taken over the course of multiple selections on n elements in the worst case. In Frederickson's algorithm, the speed is achievable because a binary heap is more structured than an unprocessed set of n elements and only a single selection is performed; the ability to perform further selection on the resulting pieces is not supported. On close examination, lazy search trees can be made to answer the selection query alone without creating additional gaps in $O(x + \log n)$ amortized time or only $O(x)$ time given a pointer to the gap in which the query falls (such modification requires fulfilling Rule (B) and the credit invariant (see the full version of the paper)).

Double-Ended Priority Queue: If every query is for the minimum or maximum element, again each query takes $O(\log n)$ time and will separate either the minimum or maximum element into its own gap and all other elements into another single gap. The new gap is destroyed when the minimum or maximum is extracted. As there is only one gap Δ_1 when insertions occur, insertions take $O(\log \log n)$ time. In this case, our data structure natively supports an $O(\log \log n)$ time decrease-key operation for keys of rank $n/2$ or less and an $O(\log \log n)$ time increase-key operation for keys of rank greater than $n/2$. Further flexibility of the change-key operation is discussed in the full version of the paper.

Online Dynamic Multiple Selection: Suppose the data structure is first constructed on n elements. After construction,

a set of ranks $\{r_i\}$ are selected, specified online and in any order. Lazy search trees will support this selection in $O(B)$ time, where $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$ is the lower bound for the multiple selection problem [40]. We can further support additional insertions, deletions and queries. Data structures for online dynamic multiple selection were previously known [37], [38], but the way we handle dynamism is more efficient, allowing for all the use cases mentioned here. We discuss this in Section II.

Split By Rank: Lazy search trees can function as a data structure for repeated splitting by rank, supporting construction on an initial set of n elements in $O(n)$ time, insertion into a piece of size n in $O(\log \log n)$ time, and all splitting within a constant factor of the information-theoretic lower bound. Here, the idea is that we would like to support the operations insert and split at rank r, returning two pieces of a data structure of the same form. In a sense, this is a generalization of priority queues, where instead of extracting the minimum, we may extract the k smallest elements, retaining the ability to perform further extractions on either of the two pieces returned. As in scenario 3, the cost of splitting is $O(x \log c + \log n)$, where x is the number of elements in the smaller resulting piece of the split, and we define c so that the number of elements in the larger resulting piece of the split is cx. Again, $O(x \log c + \log n)$ is optimal. Note that we could also use an $O(\log \log n)$ time change-key operation for this application, though this time complexity only applies when elements are moved closer to the nearest split rank. If repeatedly extracting the k smallest elements is desired, this corresponds to an $O(\log \log n)$ time decrease-key operation.

Incremental Quicksort: A version of our data structure can perform splits internally via selecting random pivots (details in the full version of the paper) with expected time complexity matching the bounds given in Theorem 1. The data structure can then be used to extract the q smallest elements in sorted order, online in q, via an incremental quicksort. Here, $B = \Theta(q \log n)$ and our overall time complexity is $O(n + q \log n)$, which is optimal up to constant factors. Previous algorithms for incremental sorting are known [41], [42], [43], [44]; however, our algorithm is extremely flexible, progressively sorting any part of the array in optimal time $O(B + n)$ while also supporting insertion, deletion, and efficient change-key. The heap operations insert and decrease-key are performed in $O(\log \log n)$ time instead of $O(\log n)$, compared to existing heaps based on incremental sorting [42]; see also [45], [46]. Our data structure also uses only $O(\min(q, n))$ pointers, providing many of the same advantages of sorting-based heaps. A more-complicated priority queue based on similar ideas to ours achieves Fibonacci heap amortized complexity with only a single extra word of space [47].

We discuss the advantages and disadvantages of our model and data structure in the following subsections.

C. Advantages

The advantages of lazy search trees are as follows:

(1) Superior runtimes to binary search trees can be achieved when queries are infrequent or non-uniformly distributed.
(2) A larger operation set, with the notable exception of efficient general merging, is made possible when used as a priority queue, supporting operations within an additive $O(n \log \log n)$ term of optimality, in our model.
(3) Lazy search trees can be implemented to use only $O(\min(q, n))$ pointers, operating mostly on arrays. This suggests smaller memory footprint, better constant factors, and better cache performance compared to many existing efficient priority queues or binary search trees. Our data structure is not built on the heap-ordered tree blueprint followed by many existing efficient priority queues [24], [25], [26], [27], [28], [29], [30], [31]. Instead, we develop a simple scheme based on unordered lists that may of independent interest.
(4) While not a corollary of the model we consider, lazy search trees can be made to satisfy all performance theorems with regards to access time satisfied by splay trees. In this way, lazy search trees often make sense as an alternative to the splay tree. Locality of access can decrease both access and insertion times. This is discussed in the full version of this paper.

D. Disadvantages

The weaknesses of lazy search trees are as follows:

(1) Our gap-based model requires inserted elements be placed in a gap immediately instead of delaying all insertion work until deemed truly necessary by query operations. In particular, a more powerful model would ensure that the number of comparisons performed on an inserted element depends only on the queries executed *after* that element is inserted. There are operation sequences where this can make a $\Theta(\log n)$ factor difference in overall time complexity, but it is not clear whether this property is important on operation sequences arising in applications.
(2) We currently do not know whether the additive $O(\min(n \log q, n \log \log n))$ term in the complexity described in Theorem 1 over a sequence of insertions and queries is necessary. Fibonacci heaps and its variants show better performance is achievable in the priority queue setting. In the full version of the paper, we show the $O(\log \log n)$ terms for insertion and change-key can be improved to a small constant if the (new) rank of the element is drawn *uniformly* at random from valid ranks in Δ_i. As a priority queue, this corresponds with operation sequences in which binary heaps [21] provide constant time insertion.

(3) The *worst-case* complexity of a single `RankBasedQuery(r)` can be $O(n)$. By delaying sorting, our lower bound indicates that we may need to spend $\Theta(n)$ time to answer a query that splits a gap of size $|\Delta_i| = \Theta(n)$ into pieces of size x and cx for $c = \Theta(1)$. Further, aside from an initial $O(\log n)$ time search, the rest of the time spent during query is on writes, so that over the course of the operation sequence the number of writes is $\Theta(B + n)$. In this sense, our algorithm functions more similarly to a lazy quicksort than a red-black tree [2], which requires only $\Theta(n)$ writes regardless of operation sequence.

E. Paper Organization & Comparison with Full Version

We organize the remainder of this extended abstract as follows. In the following section, Section II, we discuss related work. Section III gives a high-level overview of the technical challenge. In Section IV, we discuss lower bounds in our gap-based model. In Section V, we describe lazy search trees and how they perform insertions and queries. We give concluding remarks in Section VI.

Due to space constraints, several sections have been shortened and all proofs have been omitted. We also omit several sections, including: (1) a formal definition of rank-based queries, (2) the analysis of lazy search trees, (3) binary search tree bulk-update operations split and merge, (4) improved insertion and decrease-key algorithms under a weak average-case assumption, (5) a version of lazy search trees replacing exact median-finding with randomized pivoting, and (6) a version of lazy search trees based on splay trees and a proof that lazy search trees can support efficient access theorems. An eponymous full version of this extended abstract is available on arxiv.org.

II. Related Work

Lazy search trees unify several distinct research fields. The two largest, as previously discussed, are the design of efficient priority queues and balanced binary search trees. We achieved our result by developing an efficient priority queue and lazy binary search tree simultaneously. There are no directly comparable results to our work, but research in *deferred data structures* and *online dynamic multiple selection* comes closest. We further discuss differences between dynamic optimality and our work.

A. Deferred Data Structures

To our knowledge, the idea of deferred data structures was first proposed by Karp, Motwani, and Raghavan in 1988 [35]. Similar ideas have existed in slightly different forms for different problems [48], [49], [50], [51], [52], [53], [54], [55]. The term "deferred data structure" has been used more generally for delaying processing of data until queries make it necessary, but we focus on works for one-dimensional data here, as it pertains directly to the problem we consider.

Karp, Motwani and Raghavan [35] study the problem of answering membership queries on a static, unordered set of n elements in the comparison model. One solution is to construct a binary search tree of the data in $O(n \log n)$ time and then answer each query in $O(\log n)$ time. This is not optimal if the number of queries is small. Alternatively, we could answer each query in $O(n)$ time, but this is clearly not optimal if the number of queries is large. Karp et al. determine the lower bound of $\Omega((n+q)\log(\min(n,q))) = \Omega(n \log q + q \log n)$ time to answer q queries on a static set of n elements in the worst case and develop a data structure that achieves this complexity.

This work was extended in 1990 to a dynamic model. Ching, Melhorn, and Smid show that q' membership queries, insertions, and deletions on an initial set of n_0 unordered elements can be answered in $O(q' \log(n_0 + q') + (n_0 + q') \log q') = O(q' \log n_0 + n_0 \log q')$ time [36]. When membership, insertion, and deletion are considered as the same type of operation, this bound is optimal.

It is not very difficult (although not explicitly done in [36]) to modify the result of Ching et al. to obtain a data structure supporting n insertions and q'' membership or deletion operations in $O(q'' \log n + n \log q'')$ time, the runtime we achieve for uniform queries. We will see in Section III that the technical difficulty of our result is to achieve the fine-grained complexity based on the query-rank distribution. For more work in one-dimensional deferred data structures, see [48], [49], [50], [51], [52], [54].

B. Online Dynamic Multiple Selection

The optimality of Karp et al. [35] and Ching et al. [36] is in a model where the ranks requested of each query are not taken into account. In the multiple selection problem, solutions have been developed that consider this information in the analysis. Suppose we wish to select the elements of ranks $r_1 < r_2 < \cdots < r_q$ amongst a set of n unordered elements. Define $r_0 = 0$, $r_{q+1} = n$, and Δ_i as the set of elements of rank greater than r_{i-1} and at most r_i. Then $|\Delta_i| = r_i - r_{i-1}$ and as in Theorem 1, $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. The information-theoretic lower bound for multiple selection is $B - O(n)$ comparisons [56]. Solutions have been developed that achieve $O(B + n)$ time complexity [56] or $B + o(B) + O(n)$ comparison complexity [40].

The differences between the multiple selection problem and deferred data structuring for one-dimensional data are minor. Typically, deferred data structures are designed for online queries, whereas initial work in multiple selection considered the setting when all query ranks are given at the same time as the unsorted data. Solutions to the multiple selection problem where the ranks $r_1, \ldots, r_q$ are given online and in any order have also been studied, however [57]. Barbay et al. [37], [38] further extend this model to a dynamic setting: They consider online dynamic multiple selection where every insertion is preceded by a search for the inserted

element. Deletions are ultimately performed in $O(\log n)$ time. Their data structure uses $B + o(B) + O(n + q' \log n)$ comparisons, where q' is the number of search, insert, and delete operations. The crucial difference between our solution and that of Barbay et al. [37], [38] is how we handle insertions. Their analysis assumes every insertion is preceded by a search and therefore insertion must take $\Omega(\log n)$ time. Thus, for their result to be meaningful (i.e., allow $o(n \log n)$ performance), the algorithm must start with an initial set of $n_0 = n \pm o(n)$ elements. While Barbay et al. focus on online dynamic multiple selection algorithms with near-optimal comparison complexity, the focus of lazy search trees is on generality. We achieve similar complexity as a data structure for online multiple selection while also achieving near-optimal performance as a priority queue. We discuss the technical challenges in achieving this generality in Section III.

C. Dynamic Optimality

As mentioned, the dynamic optimality conjecture has received vast attention in the past four decades [7], [8], [9], [10], [11], [12], [13], [14], [15]. The original statement conjectures that the performance of the splay tree is within a constant factor of the performance of any binary search tree on any sufficiently long access sequence [3]. To formalize this statement, in particular the notion of "any binary search tree", the BST model of computation has been introduced, forcing the data structure to take the form of a binary tree with access from the root and tree rotations for updates. Dynamic optimality is enticing because it conjectures splay trees [3] and a related "greedy BST" [8] to be within a constant factor of optimality on *any* sufficiently long access sequence. This *per-instance* optimality [58] is more powerful than the sense of optimality used in less restricted models, where it is often unattainable. Any sorting algorithm, for example, must take $\Omega(n \log n)$ time in the *worst case*, but on any particular input permutation, an algorithm designed to first check for that specific permutation can sort it in $O(n)$ time: simply apply the inverse permutation and check if the resulting order is monotonic.

The bounds we give in Section IV are w. r. t. the *worst case* over operation sequences based on distribution of gaps $\{\Delta_i\}$, but hold for *any* comparison-based data structure. Hence, lazy search trees achieve a weaker notion of optimality compared to dynamic optimality, but do so against a vastly larger class of algorithms.

III. Technical Overview

This research started with the goal of generalizing a data structure that supports n insertions and $q \leq n$ rank-based queries in $O(n \log q)$ time. Via a reduction from multiple selection, $\Omega(n \log q)$ comparisons are necessary in the worst case. However, by applying the fine-grained analysis based on rank distribution previously employed in the multiple selection literature [56], a new theory which generalizes efficient priority queues and binary search trees is made possible.

As will be discussed in Section IV, to achieve optimality on sequences of insertion and distinct queries with regards to the fine-grained multiple selection lower bound, insertion into gap Δ_i should take $O(\log(n/|\Delta_i|))$ time. A query which splits a gap Δ_i into two gaps of sizes x and cx ($c \geq 1$), respectively, should take $O(x \log c + \log n)$ time. These complexities are the main goals for the design of the data structure.

The high-level idea will be to maintain elements in a gap Δ_i in an auxiliary data structure (the *interval data structure* of Section V). All such auxiliary data structures are then stored in a biased search tree so that access to the ith gap Δ_i is supported in $O(\log(n/|\Delta_i|))$ time. This matches desired insertion complexity and is within the $O(\log n)$ term of query complexity. The main technical difficulty is to support efficient insertion and repeated splitting of the auxiliary data structure.

Our high-level organization is similar to the selectable sloppy heap of Dumitrescu [59]. The difference is that while the selectable sloppy heap keeps fixed quantile groups in a balanced search tree and utilizes essentially a linked-list as the auxiliary data structure, in our case the sets of elements stored are dependent on previous query ranks, the search tree is biased, and we require a more sophisticated auxiliary data structure.

Indeed, in the priority queue case, the biased search tree has a single element Δ_1, and all operations take place within the auxiliary data structure. Thus, we ideally would like to support $O(1)$ insertion and $O(x \log c)$ split into parts of size x and cx ($c \geq 1$) in the auxiliary data structure. If the number of elements in the auxiliary data structure is $|\Delta_i|$, we can imagine finding the minimum or maximum as a split with $x = 1$ and $c = |\Delta_i| - 1$, taking $O(\log |\Delta_i|)$ time. However, the ability to split at any rank in optimal time complexity is not an operation typically considered for priority queues. Most efficient priority queues store elements in heap-ordered trees, providing efficient access to the minimum element but otherwise imposing intentionally little structure so that insertion, decrease-key, and merging can all be performed efficiently.

Our solution is to group elements within the auxiliary data structure in the following way. We separate elements into groups ("intervals") of unsorted elements, but the elements between each group satisfy a total order. Our groups are of exponentially increasing size as distance to the gap boundary increases. Within a gap Δ_i, we maintain $O(\log |\Delta_i|)$ such groups. Binary search then allows insertion and key change in $O(\log \log |\Delta_i|)$ time. While not $O(1)$, the structure created by separating elements in this way allows us to split the data structure in about $O(x)$ time, where x is the distance from the split point to the closest query rank. Unfortunately, several complications remain.

Consider if we enforce the exponentially-increasing group sizes in the natural way in data structure design. That is, we select constants $c_1 \le c_2$ such that as we get farther from the gap boundary, the next group is at least a factor $c_1 > 1$ larger than the previous but at most a factor c_2. We can maintain this invariant while supporting insertion and deletion, but splitting is problematic. After splitting, we must continue to use both pieces as a data structure of the same form. However, in the larger piece, the x elements removed require restructuring not only the new closest group to the gap boundary but could require a cascading change on all groups. Since the elements of each group are unstructured, this cascading change could take $\Omega(|\Delta_i|)$ time.

Thus, we must use a more flexible notion of "exponentially increasing" that does not require significant restructuring after a split. This is complicated by guaranteeing fast insertion and fast splits in the future. In particular, after a split, if the larger piece is again split close to where the previous split occurred, we must support this operation quickly, despite avoiding the previous cascading change that would guarantee this performance. Further, to provide fast insertion, we must keep the number of groups at $O(\log|\Delta_i|)$, but after a split, the best way to guarantee fast future splits is to create more groups.

We will show that it is possible to resolve all these issues and support desired operations efficiently by applying amortized analysis with a careful choice of structure invariants. While we do not achieve $O(1)$ insertion and decrease-key cost, our data structure is competitive as an efficient priority queue while having to solve the more complicated issues around efficient repeated arbitrary splitting.

IV. Lower and Upper Bounds

The balanced binary search tree is the most well-known solution to the sorted dictionary problem. It achieves $O(\log n)$ time for a rank-based query and $O(\log n)$ time for all dynamic operations. Via a reduction from sorting, for a sequence of n arbitrary operations, $\Omega(n \log n)$ comparisons and thus $\Omega(n \log n)$ time is necessary in the worst case.

However, this time complexity can be improved by strengthening our model. The performance theorems of the splay tree [3] show that although $\Omega(q \log n)$ time is necessary on a sequence of q arbitrary queries on n elements, many access sequences can be answered in $o(q \log n)$ time. Our model treats sequences of element *insertions* similarly to the splay tree's treatment of sequences of element access. Although $\Omega(n \log n)$ time is necessary on a sequence of n insert or query operations, on many operation sequences, $o(n \log n)$ time complexity is possible, as the theory of efficient priority queues demonstrates.

Our complexities are based on the distribution of elements into the set of gaps $\{\Delta_i\}$. We can derive a lower bound on a sequence of operations resulting in a set of gaps $\{\Delta_i\}$ via reducing multiple selection to the sorted dictionary problem. We prove Theorem 3 below.

Proof of Theorem 3: We reduce multiple selection to the sorted dictionary problem. The input of multiple selection is a set of n elements and ranks $r_1 < r_2 < \cdots < r_q$. We are required to report the elements of the desired ranks. We reduce this to the sorted dictionary problem by inserting all n elements in any order and then querying for the desired ranks $r_1, \ldots, r_q$, again in any order.

Define $r_0 = 0$, $r_{q+1} = n$, and Δ_i as the set of elements of rank greater than r_{i-1} and at most r_i. (This definition coincides with the gaps resulting in our data structure when query rank r falls in the new gap Δ'_i, described in Section I-A.) Then $|\Delta_i| = r_i - r_{i-1}$ and as in Theorem 1, $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. Note that here, $m = q + 1$. The information-theoretic lower bound for multiple selection is $B - O(n)$ comparisons [56]. Since any data structure must spend at least $O(n)$ time to read the input, this also gives a lower bound of $\Omega(B + n)$ time. This implies the sorted dictionary problem resulting in a set of gaps $\{\Delta_i\}$ must use at least $B - O(n)$ comparisons and take $\Omega(B + n)$ time. ■

To achieve the performance stated in Theorem 3 on any operation sequence, we will first consider how the bound $\Omega(B + n)$ changes with insertions and queries. This will dictate the allotted (amortized) time we can spend per operation to achieve an optimal complexity over the entire operation sequence. Recall our convention from Footnote 1 (page 3) that $\log(x) = \max(\log_2(x), 1)$ and $\log_2$ is the binary logarithm.

Lemma 4 (Influence of insert on lower bound)**.** *Suppose we insert an element into gap Δ_i. Then the bound $\Omega(B + n)$ increases by $\Omega(\log(n/|\Delta_i|))$.*

Lemma 5 (Influence of query on lower bound)**.** *Suppose a query splits a gap Δ_i into two gaps of size x and cx, respectively, with $c \ge 1$. Then the bound $\Omega(B + n)$ increases by $\Omega(x \log c)$.*

The proofs are simple calculations and given in the full version.

We can improve the query lower bound by considering the effect on B over a sequence of gap-splitting operations. Consider the overall bound $B = \sum_{i=1}^{m} |\Delta_i| \log_2(n/|\Delta_i|)$. It can be seen that $B = \Omega(m \log n)$. Therefore, we can afford amortized $O(\log n)$ time whenever a new gap is created, even if it is a split say with $x = 1$, $c = 1$.

V. Data Structure

We are now ready to discuss the details of lazy search trees. The high-level idea was discussed in Section III. The data structure as developed is relatively simple, though its analysis requires a somewhat tricky amortized time analysis (see the full version of this extended abstract).

We split the data structure into two levels. At the top level, we build a data structure on the set of gaps $\{\Delta_i\}$. In the second level, actual elements are organized into a set

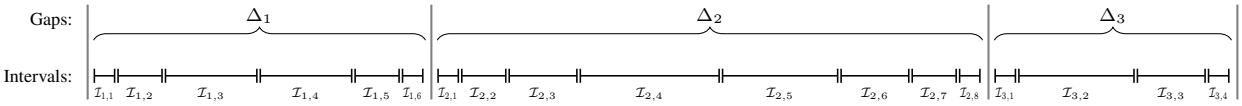

Figure 1. The two-level decomposition into gaps $\{\Delta_i\}$ and intervals $\{\mathcal{I}_{i,j}\}$.

of *intervals* within a gap. Given a gap Δ_i, intervals within Δ_i are labeled $\mathcal{I}_{i,1}, \mathcal{I}_{i,2}, \ldots, \mathcal{I}_{i,\ell_i}$, with ℓ_i the number of intervals in gap Δ_i. The organization of elements of a gap into intervals is similar to the organization of elements into a gap. Intervals partition a gap by rank, so that for elements $x \in \mathcal{I}_{i,j}$, $y \in \mathcal{I}_{i,j+1}$, $x \le y$. Elements within an interval are unordered. By convention, we will consider both gaps and intervals to be ordered from left to right in increasing rank. A graphical sketch of the high-level decomposition is given in Figure 1.

A. The Gap Data Structure

We will use the following data structure for the top level.

Lemma 6 (Gap Data Structure)**.** *There is a data structure for the gaps $\{\Delta_i\}$ that supports the following operations in the given worst-case time complexities. Note that $\sum_{i=1}^{m} |\Delta_i| = n$.*

1) *Given an element $e = (k, v)$, determine the index i such that $k \in \Delta_i$, in $O(\log(n/|\Delta_i|))$ time.*
2) *Given a Δ_i, increase or decrease the size of Δ_i by 1, adjusting n accordingly, in $O(\log(n/|\Delta_i|))$ time.*
3) *Remove Δ_i from the set, in $O(\log n)$ time.*
4) *Add a new Δ_i to the set, in $O(\log n)$ time.*

It is also possible to store aggregate functions within the data structure (on subtrees).

Proof: We can use, for example, a globally-biased $2, b$ tree [60]. We assign gap Δ_i the weight $w_i = |\Delta_i|$; the sum of weights, W, is thus equal to n. Access to gap Δ_i, operation 1, is handled in $O(\log(n/|\Delta_i|))$ worst-case time [60, Thm. 1]. By [60, Thm. 11], operation 2 is handled via weight change in $O(\log(n/|\Delta_i|))$ worst-case time. Again by [60, Thm. 11], operations 3 and 4 are handled in $O(\log n)$ worst-case time or better. ∎

The top level data structure allows us to access a gap in the desired time complexity for insertion. However, we must also support efficient queries. In particular, we need to be able to split a gap Δ_i into two gaps of size x and cx ($c \ge 1$) in amortized time $O(x \log c)$. We must build additional structure amongst the elements in a gap to support such an operation efficiently. At the cost of this organization, in the worst case we pay an additional $O(\log\log|\Delta_i|)$ time on insertion and key-changing operations.

B. The Interval Data Structure

We now discuss the data structure for the intervals. Given a gap Δ_i, intervals $\mathcal{I}_{i,1}, \mathcal{I}_{i,2}, \ldots, \mathcal{I}_{i,\ell_i}$ are contained within it and maintained in a data structure as follows. We maintain with each interval the two splitting keys (k_l, k_r) that separate this interval from its predecessor and successor (using $-\infty$ and $+\infty$ for the outermost ones), respectively; the interval only contains elements $e = (k, v)$ with $k_l \le k \le k_r$. We store intervals in sorted order in an array, sorted with respect to (k_l, k_r). We can then find an interval containing a given key k, i.e., with $k_l \le k \le k_r$, using binary search in $O(\log \ell_i)$ time. As we will see below, the number of intervals in one gap is always $O(\log n)$, and only changes during a query, so we can afford to update this array on query in linear time.

We conceptually split the intervals into two groups: intervals on the *left* side and intervals on the *right* side. An interval is defined to be in one of the two groups by the following convention.

(A) Left and right intervals: An interval $\mathcal{I}_{i,j}$ in gap Δ_i is on the *left side* if the closest query rank (edge of gap Δ_i if queries have occurred on both sides of Δ_i) is to the left. Symmetrically, an interval $\mathcal{I}_{i,j}$ is on the *right side* if the closest query rank is on the right. An interval with an equal number of elements in Δ_i on its left and right sides can be defined to be on the left or right side arbitrarily.

We balance the sizes of the intervals within a gap according to the following rule:

(B) Merging intervals: Let $\mathcal{I}_{i,j}$ be an interval on the left side, not rightmost of left side intervals. We merge $\mathcal{I}_{i,j}$ into adjacent interval to the right, $\mathcal{I}_{i,j+1}$, if the number of elements left of $\mathcal{I}_{i,j}$ in Δ_i equals or exceeds $|\mathcal{I}_{i,j}| + |\mathcal{I}_{i,j+1}|$. We do the same, reflected, for intervals on the right side.

The above rule was carefully chosen to satisfy several components of our analysis. As mentioned, we must be able to answer a query for a rank r near the edges of Δ_i efficiently. This implies we need small intervals near the edges of gap Δ_i, since the elements of each interval are unordered. However, we must also ensure the number of intervals within a gap does not become too large, since we must determine into which interval an inserted element falls at a time cost outside of the increase in B as dictated in Lemma 4. We end up using the structure dictated by Rule (B) directly in our analysis of query complexity; see the full version for details.

Note that Rule (B) causes the loss of information. Before a merge, intervals $\mathcal{I}_{i,j}$ and $\mathcal{I}_{i,j+1}$ are such that for any $x \in \mathcal{I}_{i,j}$ and $y \in \mathcal{I}_{i,j+1}$, $x \le y$. After the merge, this information is lost. Surprisingly, this does not seem to impact our analysis. Once we pay the initial $O(\log\log|\Delta_i|)$ cost to insert an element

via binary search, the merging of intervals happens seldom enough that no additional cost need be incurred.

It is easy to show that Rule (B) ensures the following.

Lemma 7 (Few intervals)**.** *Within a gap Δ_i, there are at most $4\log(|\Delta_i|)$ intervals.*

For ease of implementation, we will invoke Rule (B) only when a *query* occurs in gap Δ_i. In the following subsection, we will see that insertion does not increase the number of intervals in a gap, therefore Lemma 7 will still hold at all times even though Rule (B) might temporarily be violated after insertions. We can invoke Rule (B) in $O(\log|\Delta_i|)$ time during a query; since $|\Delta_i| \le n$ and we can afford $O(\log n)$ time per query.

C. Representation of Intervals

It remains to describe how a single interval is represented internally. Our analysis will require that merging two intervals can be done in $O(1)$ time and further that deletion from an interval can be performed in $O(1)$ time ($O(\log n)$ time actually suffices for $O(\log n)$ time delete overall, but on many operation sequences the faster interval deletion will yield better runtimes). Therefore, the container in which elements reside in intervals should support such behavior. An ordinary linked list certainly suffices; however, we can limit the number of pointers used in our data structure by representing intervals as a linked list of arrays. Whenever an interval is constructed, it can be constructed as a single (expandable) array. As intervals merge, we perform the operation in $O(1)$ time by merging the two linked lists of arrays. Deletions can be performed lazily, shrinking the array when a constant fraction of the entries have been deleted.

In the full version of this extended abstract, we show that this construction suffices for Theorem 2.

D. Insertion

Insertion of an element $e = (k, v)$ can be succinctly described as follows. We first determine the gap Δ_i such that $k \in \Delta_i$, according to the data structure of Lemma 6. We then binary search the $O(\log|\Delta_i|)$ intervals (by maintaining "router" keys separating the intervals) within Δ_i to find the interval $\mathcal{I}_{i,j}$ such that $k \in \mathcal{I}_{i,j}$. We increase the size of Δ_i by one in the gap data structure.

E. Query

To answer a query with associated rank r, we proceed as follows. We again determine the gap Δ_i such that $r \in \Delta_i$, possibly using aggregate functions stored in the data structure of Lemma 6. While we could now choose to rebalance the intervals of Δ_i via Rule (B), our analysis will not require application of Rule (B) until the *end* of the query procedure. We recurse into the interval $\mathcal{I}_{i,j}$ such that $r \in \mathcal{I}_{i,j}$, again possibly using aggregate functions on the intervals of Δ_i.

We proceed to process $\mathcal{I}_{i,j}$ by answering the query on $\mathcal{I}_{i,j}$ and replacing interval $\mathcal{I}_{i,j}$ with smaller intervals. First, we partition $\mathcal{I}_{i,j}$ into sets L and R, such that all elements in L are less than or equal to all elements in R and there are r elements in the entire data structure which are either in L or in an interval or gap left of L. This can typically be done in $O(|\mathcal{I}_{i,j}|)$ time using the result of the query itself; otherwise, linear-time selection suffices [61].

We further partition L into two sets of equal size L_l and L_r, again using linear-time selection, such that all elements in L_l are smaller than or equal to elements in L_r; if $|L|$ is odd, we give the extra element to L_l (unsurprisingly, this is not important). We then apply the same procedure *one more time* to L_r, again splitting into equal-sized intervals. Recursing further is not necessary. We do the same, reflected, for set R; after a total of 5 partitioning steps the interval splitting terminates. An example is shown in Figure 2.

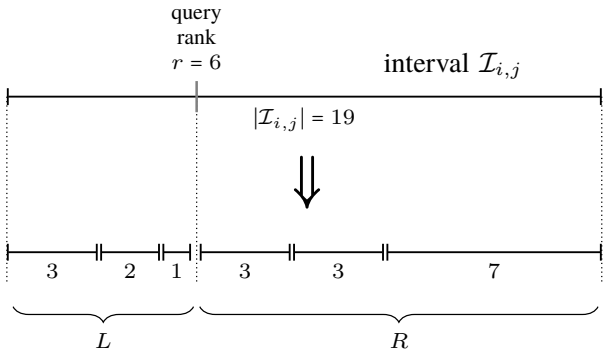

Figure 2. An interval $\mathcal{I}_{i,j}$ is split and replaced with a set of intervals.

After splitting the interval $\mathcal{I}_{i,j}$ as described above, we answer the query itself and update the gap and interval data structures as follows. We create two new gaps Δ'_i and Δ'_{i+1} out of the intervals of gap Δ_i including those created from sets L and R. Intervals that fall left of the query rank r are placed in gap Δ'_i, and intervals that fall right of the query rank r are placed in gap Δ'_{i+1}. We update the data structure of Lemma 6 with the addition of gaps Δ'_i and Δ'_{i+1} and removal of gap Δ_i. Finally, we apply Rule (B) to gaps Δ'_i and Δ'_{i+1}.

The remaining operations and the analysis of lazy search trees are presented in detail in the full version of this extended abstract.

VI. Conclusion and Open Problems

We have discussed a data structure that improves the insertion time of binary search trees, when possible. Our data structure generalizes the theories of efficient priority queues and binary search trees, providing powerful operations from both classes of data structures. As either a binary search tree or a priority queue, lazy search trees are competitive. From a theoretical perspective, our work opens the door to a new theory of insert-efficient order-based data structures.

This theory is not complete. Our runtime can be as much as an additive $O(n \log\log n)$ term from optimality in the model we study, providing $O(\log\log n)$ time insert and decrease-key operations as a priority queue when $O(1)$ has been shown to be possible [24]. Further room for improvement is seen in our model itself, where delaying insertion work further can yield improved runtimes on some operation sequences.

Acknowledgements

The authors of this paper would like to thank Ian Munro, Kevin Wu, Lingyi Zhang, Yakov Nekrich, and Meng He for useful discussions on the topics of this paper. We would also like to thank the anonymous reviewers for their helpful suggestions.

References

[1] G. Adelson-Velsky and E. Landis, "An algorithm for the organization of information," *Proceedings of the USSR Academy of Sciences*, vol. 146, pp. 263–266, 1962.

[2] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[3] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary seach trees," *Journal of the ACM*, vol. 32, no. 3, pp. 652–686, 1985.

[4] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars, *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

[5] R. Cole, B. Mishra, J. Schmidt, and A. Siegel, "On the dynamic finger conjecture for splay trees. part I: Splay sorting $\log n$-block sequences," *SIAM Journal on Computing*, vol. 30, no. 1, pp. 1–43, 2000.

[6] R. Cole, "On the dynamic finger conjecture for splay trees. part II: The proof," *SIAM Journal on Computing*, vol. 30, no. 1, pp. 44–85, 2000.

[7] B. Allen and I. Munro, "Self-organizing binary search trees," *Journal of the ACM*, vol. 25, no. 4, pp. 526–535, 1978.

[8] E. D. Demaine, D. Harmon, J. Iacono, D. Kane, and M. Patrascu, "The geometry of binary search trees," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 2009, pp. 496–505.

[9] E. D. Demaine, D. Harmon, J. Iacono, and M. Patrascu, "Dynamic optimality–almost." *Siam Journal of Computing*, vol. 37, no. 1, pp. 240–251, 2007.

[10] J. Iacono and S. Langerman, "Weighted dynamic finger in binary search trees," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 2016.

[11] P. Bose, J. Cardinal, J. Iacono, G. Koumoutsos, and S. Langerman, "Competitive online search trees on trees," in *Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, Jan. 2020, pp. 1878–1891.

[12] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of Computer and System Sciences*, vol. 26, no. 3, pp. 362–391, 1983.

[13] R. E. Wilber, "Lower bounds for accessing binary search trees with rotations," *Siam Journal of Computing*, vol. 18, no. 1, pp. 56–69, 1989.

[14] L. Kozma and T. Saranurak, "Smooth heaps and a dual view of self-adjusting data structures," *SIAM Journal on Computing*, pp. STOC18–45–STOC18–93, Nov. 2019.

[15] P. Chalermsook, M. Goswami, L. Kozma, K. Mehlhorn, and T. Saranurak, "Pattern-avoiding access in binary search trees," in *Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2015, pp. 410–423.

[16] J. Iacono, "Alternatives to splay trees with $o(\log n)$ worst-case access time," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 2001, pp. 516–522.

[17] M. Bădoiu, R. Cole, E. D. Demaine, and J. Iacono, "A unified access bound on comparison-based dynamic dictionaries," *Theoretical Computer Science*, vol. 382, no. 2, pp. 86–96, Aug. 2007.

[18] C. Levy and R. E. Tarjan, "A new path from splay to dynamic optimality," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 2019, pp. 1311–1330.

[19] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with $o(1)$ worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, Jun. 1984.

[20] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[21] J. W. J. Williams, "Algorithm 232 - heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[22] J. Vuillemin, "A data structure for manipulating priority queues," *Communications of the ACM*, vol. 21, no. 4, pp. 309–315, 1978.

[23] M. R. Brown, "Implementation and analysis of binomial queue algorithms," *SIAM Journal on Computing*, vol. 7, no. 3, pp. 298–319, 1978.

[24] M. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the Association for Computing Machinery*, vol. 34, no. 3, pp. 596–615, 1987.

[25] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, "The pairing heap: A new form of self-adjusting heap," *Algorithmica*, vol. 1, no. 1, pp. 111–129, 1986.

[26] T. M. Chan, "Quake heaps: A simple alternative to Fibonacci heaps," in *Space-Efficient Data Structures, Streams, and Algorithms*. Springer, 2009, pp. 27–32.

[27] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan, "Strict Fibonacci heaps," in *Symposium on Theory of Computing (STOC)*. ACM, 2012.

[28] A. Elmasry, "Pairing heaps with $O(\log\log n)$ decrease cost," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 2009.

[29] B. Haeupler, S. Sen, and R. E. Tarjan, "Rank-pairing heaps," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1463–1485, 2011.

[30] G. Brodal, "Worst-case efficient priority queues," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 1996.

[31] T. D. Hansen, H. Kaplan, R. E. Tarjan, and U. Zwick, "Hollow heaps," *ACM Transactions on Algorithms*, vol. 13, no. 3, pp. 1–27, 2017.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[33] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in *Symposium on Discrete Algorithms (SODA)*. SIAM, 2003, pp. 546–554.

[34] P. Bose, J. Howat, and P. Morin, "A history of distribution-sensitive data structures," in *Space-Efficient Data Structures, Streams, and Algorithms*. Springer, 2009, pp. 133–149.

[35] R. M. Karp, R. Motwani, and P. Raghavan, "Deferred data structuring," *SIAM Journal on Computing*, vol. 17, no. 5, pp. 883–902, 1988.

[36] Y.-T. Ching, K. Mehlhorn, and M. H. Smid, "Dynamic deferred data structuring," *Information Processing Letters*, vol. 35, no. 1, pp. 37 – 40, 1990.

[37] J. Barbay, A. Gupta, S. Rao Satti, and J. Sorenson, "Dynamic online multiselection in internal and external memory," in *WALCOM: Algorithms and Computation*. Springer, 2015, pp. 199–209.

[38] J. Barbay, A. Gupta, S. R. Satti, and J. Sorenson, "Near-optimal online multiselection in internal and external memory," *Journal of Discrete Algorithms*, vol. 36, pp. 3–17, 2016, wALCOM 2015.

[39] G. N. Frederickson, "An optimal algorithm for selection in a min-heap," *Information and Computation*, vol. 104, no. 2, pp. 197–214, 1993.

[40] K. Kaligosi, K. Mehlhorn, J. I. Munro, and P. Sanders, "Towards optimal multiple selection," in *International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2005, pp. 103–114.

[41] R. Parades and G. Navarro, "Optimal incremental sorting," in *Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, Jan. 2006.

[42] G. Navarro and R. Paredes, "On sorting, heaps, and minimum spanning trees," *Algorithmica*, vol. 57, no. 4, pp. 585–620, Mar. 2010.

[43] E. Regla and R. Paredes, "Worst-case optimal incremental sorting," in *Conference of the Chilean Computer Science Society (SCCC)*. IEEE, Nov. 2015.

[44] A. A. Aydin and K. M. Anderson, "Incremental sorting for large dynamic data sets," in *International Conference on Big Data Computing Service and Applications*. IEEE, 2015.

[45] S. Edelkamp, A. Elmasry, and J. Katajainen, "The weak-heap data structure: Variants and applications," *Journal of Discrete Algorithms*, vol. 16, pp. 187–205, Oct. 2012.

[46] G. S. Brodal, "A survey on priority queues," in *Space-Efficient Data Structures, Streams, and Algorithms*. Springer, 2013, pp. 150–163.

[47] C. W. Mortensen and S. Pettie, "The complexity of implicit and space efficient priority queues," in *Workshop on Algorithms and Data Structures (WADS)*. Springer, 2005, pp. 49–60.

[48] M. Smid, "Dynamic data structures on multiple storage media," Ph.D. dissertation, University of Amsterdam, 1989.

[49] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao, "Efficient searching using partial ordering," *Information Processing Letters*, vol. 12, no. 2, pp. 71–75, 1981.

[50] G. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders, "Towards optimal range medians," *Theoretical Computer Science*, vol. 412, no. 24, pp. 2588–2601, 2011.

[51] J. Barbay, "Optimal prefix free codes with partial sorting," *Algorithms*, vol. 13, no. 1, p. 12, Dec. 2019.

[52] J. Barbay, C. Ochoa, and S. R. Satti, "Synergistic solutions on multisets," in *Annual Symposium on Combinatorial Pattern Matching (CPM)*, ser. LIPIcs, vol. 78. Schloss Dagstuhl, 2017.

[53] S. Ar, G. Montag, and A. Tal, "Deferred, self-organizing BSP trees," *Computer Graphics Forum*, vol. 21, no. 3, pp. 269–278, Sep. 2002.

[54] B. Gum and R. Lipton, "Cheaper by the dozen: Batched algorithms," in *International Conference on Data Mining (ICDM)*. SIAM, 2001.

[55] A. Aggarwal and P. Raghavan, "Deferred data structure for the nearest neighbor problem," *Information Processing Letters*, vol. 40, no. 3, pp. 119–122, Nov. 1991.

[56] D. Dobkin and J. I. Munro, "Optimal time minimal space selection algorithms," *Journal of the Association for Computing Machinery*, vol. 28, no. 3, pp. 454–461, 1981.

[57] J. Barbay, A. Gupta, S. Jo, S. Rao Satti, and J. Sorenson, "Theory and implementation of online multiselection algorithms," in *European Symposium on Algorithms (ESA)*. Springer, 2013, pp. 109–120.

[58] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 614–656, Jun. 2003.

[59] A. Dumitrescu, "A selectable sloppy heap," *Algorithms*, vol. 12, no. 3, p. 58, 2019.

[60] S. W. Bent, D. D. Selator, and R. E. Tarjan, "Biased search trees," *SIAM Journal on Computing*, vol. 14, no. 3, pp. 545–568, 1985.

[61] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.