11

Finger Search Trees

Gerth Stølting Brodal
University of Aarhus

11.1 Finger Searching

One of the most studied problems in computer science is the problem of maintaining a sorted sequence of elements to facilitate efficient searches. The prominent solution to the problem is to organize the sorted sequence as a balanced search tree, enabling insertions, deletions and searches in logarithmic time. Many different search trees have been developed and studied intensively in the literature. A discussion of balanced binary search trees can be found in Chapter 10.

This chapter is devoted to *finger search trees*, which are search trees supporting *fingers*, i.e., pointers to elements in the search trees and supporting efficient updates and searches in the vicinity of the fingers.

If the sorted sequence is a *static* set of n elements then a simple and space efficient representation is a sorted array. Searches can be performed by binary search using $1+\lfloor \log n \rfloor$ comparisons (we throughout this chapter let $\log x$ to denote $\log_2 \max\{2, x\}$). A finger search starting at a particular element of the array can be performed by an *exponential search* by inspecting elements at distance $2^i - 1$ from the finger for increasing i followed by a binary search in a range of $2^{\lfloor \log d \rfloor} - 1$ elements, where d is the rank difference in the sequence between the finger and the search element. In Figure 11.1 is shown an exponential search for the element 42 starting at 5. In the example $d = 20$. An exponential search requires

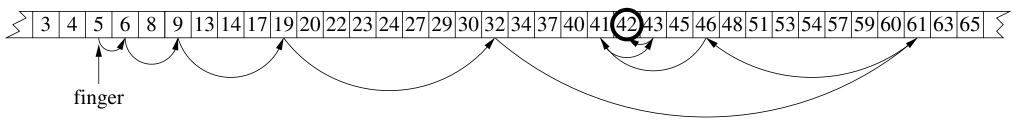

FIGURE 11.1: Exponential search for 42.

$2 + 2\lfloor \log d \rfloor$ comparisons.

Bentley and Yao [5] gave a close to optimal static finger search algorithm which performs $\sum_{i=1}^{\log^* d-1} \log^{(i)} d + \mathcal{O}(\log^* d)$ comparisons, where $\log^{(1)} x = \log x$, $\log^{(i+1)} x = \log(\log^{(i)} x)$, and $\log^* x = \min\{i \mid \log^{(i)} x \leq 1\}$.

11.2 Dynamic Finger Search Trees

A dynamic finger search data structure should in addition to finger searches also support the insertion and deletion of elements at a position given by a finger. This section is devoted to an overview of existing dynamic finger search data structures. Section 11.3 and Section 11.4 give details concerning how three constructions support efficient finger searches: The level linked (2,4)-trees of Huddleston and Mehlhorn [26], the randomized skip lists of Pugh [36, 37] and the randomized binary search trees, treaps, of Seidel and Aragon [39].

Guibas et al. [21] introduced finger search trees as a variant of B-trees [4], supporting finger searches in $\mathcal{O}(\log d)$ time and updates in $\mathcal{O}(1)$ time, assuming that only $\mathcal{O}(1)$ movable fingers are maintained. Moving a finger d positions requires $\mathcal{O}(\log d)$ time. This work was refined by Huddleston and Mehlhorn [26]. Tsakalidis [42] presented a solution based on AVL-trees, and Kosaraju [29] presented a generalized solution. Tarjan and van Wyk [41] presented a solution based on red-black trees.

The above finger search tree constructions either assume a fixed constant number of fingers or only support updates in amortized constant time. Constructions supporting an arbitrary number of fingers and with worst case update have been developed. Levcopoulos and Overmars [30] presented a search tree that supported updates at an arbitrary position in worst case $\mathcal{O}(1)$ time, but only supports searches in $\mathcal{O}(\log n)$ time. Constructions supporting $\mathcal{O}(\log d)$ time searches and $\mathcal{O}(\log^* n)$ time insertions and deletions were developed by Harel [22, 23] and Fleischer [19]. Finger search trees with worst-case constant time insertions and $\mathcal{O}(\log^* n)$ time deletions were presented by Brodal [7], and a construction achieving optimal worst-case constant time insertions and deletions were presented by Brodal et al. [9].

Belloch et al. [6] developed a space efficient alternative solution to the level linked (2,4)-trees of Huddleston and Mehlhorn, see Section 11.3. Their solution allows a single finger, that can be moved by the same performance cost as (2,4)-trees. In the solution no level links and parent pointers are required, instead a special $\mathcal{O}(\log n)$ space data structure, *hand*, is created for the finger that allows the finger to be moved efficiently.

Sleator and Tarjan introduced *splay trees* as a class of self-adjusting binary search trees supporting searches, insertions and deletions in amortized $\mathcal{O}(\log n)$ time [40]. That splay trees can be used as efficient finger search trees was later proved by Cole [15, 16]: Given an $\mathcal{O}(n)$ initialization cost, the amortized cost of an access at distance d from the preceding access in a splay tree is $\mathcal{O}(\log d)$ where accesses include searches, insertions, and deletions. Notice that the statement only applies in the presence of one finger, which always points to the last accessed element.

All the above mentioned constructions can be implemented on a pointer machine where the only operation allowed on elements is the comparison of two elements. For the Random Access Machine model of computation (RAM), Dietz and Raman [17, 38] developed a finger search tree with constant update time and $\mathcal{O}(\log d)$ search time. This result is achieve by tabulating small tree structures, but only performs the comparison of elements. In the same model of computation, Andersson and Thorup [2] have surpassed the logarithmic bound in the search procedure by achieving $\mathcal{O}\left(\sqrt{\frac{\log d}{\log \log d}}\right)$ query time. This result is achieved by considering elements as bit-patterns/machine words and applying techniques developed for the RAM to surpass lower bounds for comparison based data structures. A survey on RAM dictionaries can be found in Chapter 39.

11.3 Level Linked (2,4)-Trees

In this section we discuss how (2,4)-trees can support efficient finger searches by the introduction of *level links*. The ideas discussed in this section also applies to the more general class of height-balanced trees denoted (a, b)-trees, for $b \geq 2a$. A general discussion of height balanced search trees can be found in Chapter 10. A throughout treatment of level linked (a, b)-trees can be found in the work of Huddleston and Mehlhorn [26, 32].

A (2,4)-tree is a height-balanced search tree where all leaves have the same depth and all internal nodes have degree two, three or four. Elements are stored at the leaves, and internal nodes only store search keys to guide searches. Since each internal node has degree at least two, it follows that a (2,4)-tree has height $\mathcal{O}(\log n)$ and supports searches in $\mathcal{O}(\log n)$ time.

An important property of (2,4)-trees is that insertions and deletions given by a finger take amortized $\mathcal{O}(1)$ time (this property is not shared by $(2, 3)$-trees, where there exist sequences of n insertions and deletions requiring $\Theta(n \log n)$ time). Furthermore a (2,4)-tree with n leaves can be split into two trees of size n_1 and n_2 in amortized $\mathcal{O}(\log \min(n_1, n_2))$ time. Similarly two (2,4)-trees of size n_1 and n_2 can be joined (concatenated) in amortized $\mathcal{O}(\log \min(n_1, n_2))$ time.

To support finger searches (2,4)-trees are augmented with level links, such that all nodes with equal depth are linked together in a double linked list. Figure 11.2 shows a (2,4)-tree augmented with level links. Note that all edges represent bidirected links. The additional level links are straightforward to maintain during insertions, deletions, splits and joins of (2,4)-trees.

To perform a finger search from x to y we first check whether y is to the left or right of x. Assume without loss of generality that y is to the right of x. We then traverse the path from x towards the root while examining the nodes v on the path and their right neighbors until it has been established that y is contained within the subtree rooted at v or v's right neighbor. The upwards search is then terminated and at most two downwards searches for y is started at respectively v and/or v's right neighbor. In Figure 11.2 the pointers followed during a finger search from J to T are depicted by thick lines.

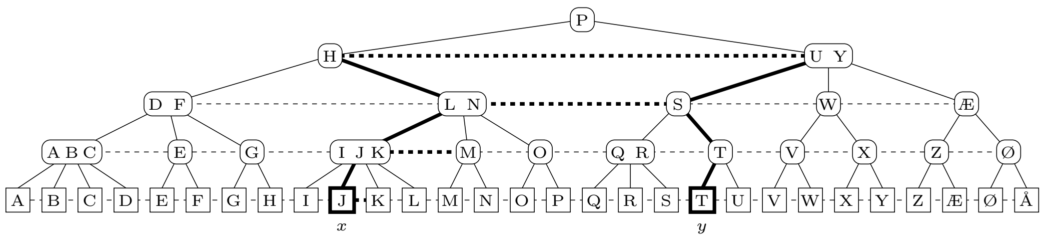

FIGURE 11.2: Level linked (2,4)-trees.

The $\mathcal{O}(\log d)$ search time follows from the observation that if we advance the upwards search to the parent of node v then y is to the right of the leftmost subtree of $v's$ right neighbor, i.e. d is at least exponential in the height reached so far. In Figure 11.2 we advance from the internal node labeled "L N" to the node labeled "H" because from "S" we know that y is to the right of the subtree rooted at the node "Q R".

The construction for level linked (2,4)-trees generalizes directly to level linked (a, b)-trees that can be used in external memory. By choosing $a = 2b$ and b such that an internal node fits in a block in external memory, we achieve *external memory* finger search trees supporting insertions and deletions in $\mathcal{O}(1)$ memory transfers, and finger searches with $\mathcal{O}(\log_b n)$ memory transfers.

11.4 Randomized Finger Search Trees

Two randomized alternatives to deterministic search trees are the randomized binary search trees, *treaps*, of Seidel and Aragon [39] and the *skip lists* of Pugh [36, 37]. Both treaps and skip lists are elegant data structures, where the randomization facilitates simple and efficient update operations.

In this section we describe how both treaps and skip lists can be used as efficient finger search trees without altering the data structures. Both data structures support finger searches in *expected* $\mathcal{O}(\log d)$ time, where the expectations are taken over the random choices made by the algorithm during the construction of the data structure. For a general introduction to randomized dictionary data structures see Chapter 13.

11.4.1 Treaps

A treap is a rooted binary tree where each node stores an element and where each element has an associated random priority. A treap satisfies that the elements are sorted with respect to an inorder traversal of tree, and that the priorities of the elements satisfy heap order, i.e., the priority stored at a node is always smaller than or equal to the priority stored at the parent node. Provided that the priorities are distinct, the shape of a treap is uniquely determined by its set of elements and the associated priorities. Figure 11.3 shows a treap storing the elements A,B,...,T and with random integer priorities between one and hundred.

The most prominent properties of treaps are that they have expected $\mathcal{O}(\log n)$ height, implying that they provide searches in expected $\mathcal{O}(\log n)$ time. Insertions and deletions of elements can be performed in expected at most two rotations and expected $\mathcal{O}(1)$ time, provided that the position of insertion or deletion is known, i.e. insertions and deletions given by a finger take expected $\mathcal{O}(1)$ time [39].

The essential property of treaps enabling expected $\mathcal{O}(\log d)$ finger searches is that for two elements x and y whose ranks differ by d in the set stored, the expected length of the path between x and y in the treap is $\mathcal{O}(\log d)$. To perform a finger search for y starting with a finger at x, we ideally start at x and traverse the ancestor path of x until we reach the *least common ancestor* of x and y, $\mathrm{LCA}(x, y)$, and start a downward tree search for y. If we can decide if a node is $\mathrm{LCA}(x, y)$, this will traverse exactly the path from x to y. Unfortunately, it is nontrivial to decide if a node is $\mathrm{LCA}(x, y)$. In [39] it is assumed that a treap is extended with additional pointers to facilitate finger searches in expected $\mathcal{O}(\log d)$ time. Below an alternative solution is described not requiring any additional pointers than the standard left, right and parent pointers.

Assume without loss of generality that we have a finger at x and have to perform a finger search for $y \geq x$ present in the tree. We start at x and start traversing the ancestor path of x. During this traversal we keep a pointer ℓ to the last visited node that can potentially

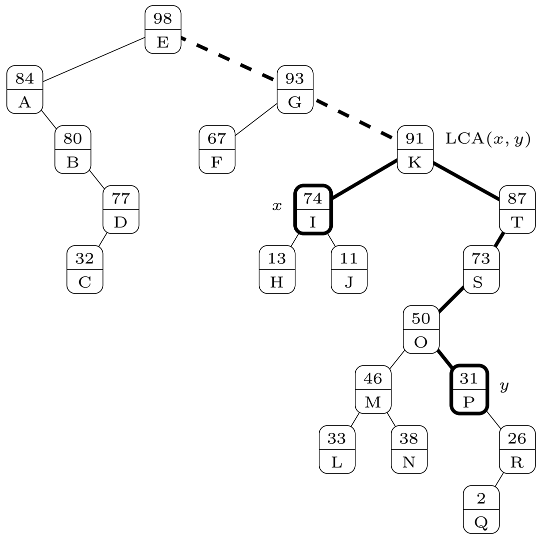

FIGURE 11.3: Performing finger searches on treaps.

be LCA(x, y). Whenever we visit a node v on the path from x to the root there are three cases:

(1) $v \leq x$, then x is in the right subtree of v and cannot be LCA(x, y); we advance to the parent of v.

(2) $x < v \leq y$, then x is in the left subtree of v and LCA(x, y) is either y or an ancestor of y; we reset $\ell = v$ and advance to the parent of v.

(3) $x < y < v$, then LCA(x, y) is in the left subtree of v and equals ℓ.

Unfortunately, after LCA(x, y) has been visited case (1) can happen $\omega(\log d)$ times before the search is terminated at the root or by case (3). Seidel and Aragon [39] denote these extra nodes visited above LCA(x, y) the *excess path* of the search, and circumvent this problem by extending treaps with special pointers for this.

To avoid visiting long excess paths we extend the above upward search with a concurrent downward search for y in the subtree rooted at the current candidate ℓ for LCA(x, y). In case (1) we always advance the tree search for y one level down, in case (2) we restart the search at the new ℓ, and in (3) we finalize the search. The concurrent search for y guarantees that the distance between LCA(x, y) and y in the tree is also an upper bound on the nodes visited on the excess path, i.e. we visit at most twice the number of nodes as is on the path between x and y, which is expected $\mathcal{O}(\log d)$. It follows that treaps support finger searches in $\mathcal{O}(\log d)$ time. In Figure 11.3 is shown the search for $x = I$, $y = P$, LCA$(x, y) = K$, the path from x to y is drawn with thick lines, and the excess path is drawn with dashed lines.

11.4.2 Skip Lists

A skip list is a randomized dictionary data structure, which can be considered to consists of expected $\mathcal{O}(\log n)$ levels. The lowest level being a single linked list containing the elements in sorted order, and each succeeding level is a random sample of the elements of the previous level, where each element is included in the next level with a fixed probability, e.g. 1/2. The pointer representation of a skip is illustrated in Figure 11.4.

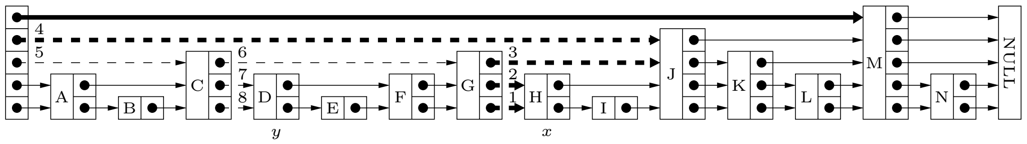

FIGURE 11.4: Performing finger searches on skip list.

The most prominent properties of skip lists are that they require expected linear space, consist of expected $\mathcal{O}(\log n)$ levels, support searches in expected $\mathcal{O}(\log n)$ time, and support insertions and deletions at a given position in expected $\mathcal{O}(1)$ time [36, 37].

Pugh in [36] elaborates on the various properties and extensions of skip lists, including pseudo-code for how skip lists support finger searches in expected $\mathcal{O}(\log d)$ time. To facilitate backward finger searches, a finger to a node v is stored as an expected $\mathcal{O}(\log n)$ space finger data structure that for each level i stores a pointer to the node to the left of v where the level i pointer either points to v or a node to the right of v. Moving a finger requires this list of pointers to be updated correspondingly.

A backward finger search is performed by first identifying the lowest node in the finger data structure that is to the left of the search key y, where the nodes in the finger data structure are considered in order of increasing levels. Thereafter the search proceeds downward from the identified node as in a standard skip list search.

Figure 11.4 shows the situation where we have a finger to H, represented by the thick (solid or dashed) lines, and perform a finger search for the element D to the left of H. Dashed (thick and thin) lines are the pointers followed during the finger search. The numbering indicate the other in which the pointers are traversed.

If the level links of a skip list are maintained as double-linked lists, then finger searches can be performed in expected $\mathcal{O}(\log d)$ time by traversing the existing links, without having a separate $\mathcal{O}(\log n)$ space finger data structure

11.5 Applications

Finger search trees have, e.g., been used in algorithms within computational geometry [3, 8, 20, 24, 28, 41] and string algorithms [10, 11]. In the rest of this chapter we give examples of the efficiency that can be obtained by applying finger search trees. These examples typically allow one to save a factor of $\mathcal{O}(\log n)$ in the running time of algorithms compared to using standard balanced search trees supporting $\mathcal{O}(\log n)$ time searches.

11.5.1 Optimal Merging and Set Operations

Consider the problem of merging two sorted sequences X and Y of length respectively n and m, where $n \leq m$, into one sorted sequence of length $n + m$. The canonical solution is to repeatedly insert each $x \in X$ in Y. This requires that Y is searchable and that there can be inserted new elements, i.e. a suitable representation of Y is a balanced search tree. This immediately implies an $\mathcal{O}(n \log m)$ time bound for merging. In the following we discuss how finger search trees allow this bound to be improved to $\mathcal{O}(n \log \frac{m}{n})$.

Hwang and Lin [27] presented an algorithm for merging two sorted sequence using optimal $\mathcal{O}(n \log \frac{m}{n})$ comparisons, but did not discuss how to represent the sets. Brown and Tarjan [12] described how to achieve the same bound for merging two AVL trees [1]. Brown and Tarjan subsequently introduced *level linked* (2,3)-trees and described how to achieve the same merging bound for level linked (2,3)-trees [13].

Optimal merging of two sets also follows as an application of finger search trees [26]. Assume that the two sequences are represented as finger search trees, and that we repeatedly insert the n elements from the shorter sequence into the larger sequence using a finger that moves monotonically from left to right. If the ith insertion advances the finger d_i positions, we have that the total work of performing the n finger searches and insertions is $\mathcal{O}(\sum_{i=1}^{n} \log d_i)$, where $\sum_{i=1}^{n} d_i \le m$. By convexity of the logarithm the total work becomes bounded by $\mathcal{O}(n \log \frac{m}{n})$.

Since sets can be represented as sorted sequences, the above merging algorithm gives immediately raise to optimal, i.e. $\mathcal{O}\left(\log \binom{n+m}{n}\right) = \mathcal{O}(n \log \frac{m}{n})$ time, algorithms for set union, intersection, and difference operations [26]. For a survey of data structures for set representations see Chapter 33.

11.5.2 Arbitrary Merging Order

A classical $\mathcal{O}(n \log n)$ time sorting algorithm is binary merge sort. The algorithm can be viewed as the merging process described by a balanced binary tree: Each leaf corresponds to an input element and each internal node corresponds to the merging of the two sorted sequences containing respectively the elements in the left and right subtree of the node. If the tree is balanced then each element participates in $\mathcal{O}(\log n)$ merging steps, i.e. the $\mathcal{O}(n \log n)$ sorting time follows.

Many divide-and-conquer algorithms proceed as binary merge sort, in the sense that the work performed by the algorithm can be characterized by a treewise merging process. For some of these algorithms the tree determining the merges is unfortunately fixed by the input instance, and the running time using linear merges becomes $\mathcal{O}(n \cdot h)$, where h is the height of the tree. In the following we discuss how finger search trees allow us to achieve $\mathcal{O}(n \log n)$ for unbalanced merging orders to.

Consider an arbitrary binary tree $\mathcal{T}$ with n leaves, where each leaf stores an element. We allow $\mathcal{T}$ to be arbitrarily unbalanced and that elements are allowed to appear at the leaves in any arbitrary order. Associate to each node v of $\mathcal{T}$ the set $\mathcal{S}_v$ of elements stored at the leaves of the subtree rooted at v. If we for each node v of $\mathcal{T}$ compute $\mathcal{S}_v$ by merging the two sets of the children of v using finger search trees, cf. Section 11.5.1, then the total time to compute all the sets $\mathcal{S}_v$ is $\mathcal{O}(n \log n)$.

The proof of the total $\mathcal{O}(n \log n)$ bound is by structural induction where we show that in a tree of size n, the total merging cost is $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$. Recall that two sets of size n_1 and n_2 can be merged in $\mathcal{O}\left(\log \binom{n_1+n_2}{n_1}\right)$ time. By induction we get that the total merging in a subtree with a root with two children of size respectively n_1 and n_2 becomes:

$$\begin{aligned}
&\log(n_1!) + \log(n_2!) + \log \binom{n_1+n_2}{n_1} \\
&= \log(n_1!) + \log(n_2!) + \log((n_1+n_2)!) - \log(n_1!) - \log(n_2!) \\
&= \log((n_1+n_2)!) \ .
\end{aligned}$$

The above approach of arbitrary merging order was applied in [10, 11] to achieve $\mathcal{O}(n \log n)$ time algorithms for finding repeats with gaps and quasiperiodicities in strings. In both these algorithms $\mathcal{T}$ is determined by the suffix-tree of the input string, and the S_v sets denote the set of occurrences (positions) of the substring corresponding to the path label of v.

11.5.3 List Splitting

Hoffmann et al. [25] considered how finger search trees can be used for solving the following *list splitting* problem, that e.g. also is applied in [8, 28]. Assume we initially have a sorted list of n elements that is repeatedly split into two sequences until we end up with n sequences each containing one element. If the splitting of a list of length k into two lists of length k_1 and k_2 is performed by performing a simultaneous finger search from each end of the list, followed by a split, the searching and splitting can be performed in $\mathcal{O}(\log \min(k_1, k_2))$ time. Here we assume that the splitting order is unknown in advance.

By assigning a list of k elements a potential of $k - \log k \geq 0$, the splitting into two lists of size k_1 and k_2 releases the following amount of potential:

$$\begin{aligned} &(k - \log k) - (k_1 - \log k_1) - (k_2 - \log k_2) \\ &\quad = \; -\log k + \log \min(k_1, k_2) + \log \max(k_1, k_2) \\ &\quad \geq \; -1 + \log \min(k_1, k_2) \ , \end{aligned}$$

since $\max(k_1, k_2) \geq k/2$. The released potential allows each list splitting to be performed in amortized $\mathcal{O}(1)$ time. The initial list is charged $n - \log n$ potential. We conclude that starting with a list of n elements, followed by a sequence of at most $n - 1$ splits requires total $\mathcal{O}(n)$ time.

11.5.4 Adaptive Merging and Sorting

The area of *adaptive sorting* addresses the problem of developing sorting algorithms which perform $o(n \log n)$ comparisons for inputs with a limited amount of disorder for various definitions of measures of disorder, e.g. the measure Inv counts the number of pairwise insertions in the input. For a survey of adaptive sorting algorithms see [18].

An adaptive sorting algorithm that is optimal with respect to the disorder measure Inv has running time $\mathcal{O}(n \log \frac{\text{Inv}}{n})$. A simple adaptive sorting algorithm optimal with respect to Inv is the *insertion sort* algorithm, where we insert the elements of the input sequence from left to right into a finger search tree. Insertions always start at a finger on the last element inserted. Details on applying finger search trees in insertion sort can be found in [13, 31, 32].

Another adaptive sorting algorithm based on applying finger search trees is obtained by replacing the linear merging in binary merge sort by an *adaptive merging* algorithm [14, 33–35]. The classical binary merge sort algorithm alway performs $\Omega(n \log n)$ comparisons, since in each merging step where two lists each of size k is merged the number of comparisons performed is between k and $2k - 1$.

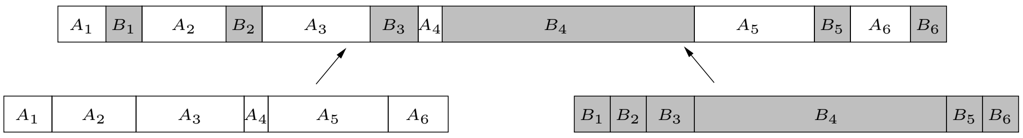

FIGURE 11.5: Adaptive merging.

The idea of the adaptive merging algorithm is to identify consecutive blocks from the input sequences which are also consecutive in the output sequence, as illustrated in Figure 11.5. This is done by repeatedly performing a finger search for the smallest element of the two input sequences in the other sequence and deleting the identified block in the other sequence by a split operation. If the blocks in the output sequence are denoted $Z_1, \ldots, Z_k$, it follows from the time bounds of finger search trees that the total time for this adaptive merging operation becomes $\mathcal{O}(\sum_{i=1}^{k} \log |Z_i|)$. From this merging bound it can be argued that merge sort with adaptive merging is adaptive with respect to the disorder measure Inv (and several other disorder measures). See [14, 33, 34] for further details.

Acknowledgment

This work was supported by the Carlsberg Foundation (contract number ANS-0257/20), BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), and the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

References

[1] G. M. Adel'son-Vel'skii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 146:263–266, 1962. English translation in *Soviet Math. Dokl.*, 3:1259–1262.

[2] A. Anderson and M. Thorup. Tight(er) worst case bounds on dynamic searching and priority queues. In *Proc. 32nd Annual ACM Symposium On Theory of Computing*, pages 335–342, 2000.

[3] M. Atallah, M. Goodrich, and K.Ramaiyer. Biased finger trees and three-dimensional layers of maxima. In *Proc. 10th ACM Symposium on Computational Geometry*, pages 150–159, 1994.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[5] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, 1976.

[6] G. E. Blelloch, B. M. Maggs, and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 374–383. Society for Industrial and Applied Mathematics, 2003.

[7] G. S. Brodal. Finger search trees with constant insertion time. In *Proc. 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 540–549, 1998.

[8] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proc. 43rd Annual Symposium on Foundations of Computer Science*, pages 617–626, 2002.

[9] G. S. Brodal, G. Lagogiannis, C. Makris, A. Tsakalidis, and K. Tsichlas. Optimal finger search trees in the pointer machine. *Journal of Computer and System Sciences, Special issue on STOC 2002*, 67(2):381–418, 2003.

[10] G. S. Brodal, R. B. Lyngsø, C. N. S. Pedersen, and J. Stoye. Finding maximal pairs with bounded gap. *Journal of Discrete Algorithms, Special Issue of Matching Patterns*, 1(1):77–104, 2000.

[11] G. S. Brodal and C. N. S. Pedersen. Finding maximal quasiperiodicities in strings. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 397–411. Springer-Verlag, 2000.

[12] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[13] M. R. Brown and R. E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing*, 9:594–614, 1980.

[14] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear merging and natural mergesort. *Algorithmica*, 9(6):629–648, 1993.

[15] R. Cole. On the dynamic finger conjecture for splay trees. part II: The proof. *SIAM Journal of Computing*, 30(1):44–85, 2000.

[16] 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 of Computing*, 30(1):1–43, 2000.

[17] P. F. Dietz and R. Raman. A constant update time finger search tree. *Information Processing Letters*, 52:147–154, 1994.

[18] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24:441–476, 1992.

[19] R. Fleischer. A simple balanced search tree with $O(1)$ worst-case update time. *International Journal of Foundations of Computer Science*, 7:137–149, 1996.

[20] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. *Algorithmica*, 2:209–233, 1987.

[21] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. 9th Ann. ACM Symp. on Theory of Computing*, pages 49–60, 1977.

[22] D. Harel. Fast updates of balanced search trees with a guaranteed time bound per update. Technical Report 154, University of California, Irvine, 1980.

[23] D. Harel and G. S. Lueker. A data structure with movable fingers and deletions. Technical Report 145, University of California, Irvine, 1979.

[24] J. Hershberger. Finding the visibility graph of a simple polygon in time proportional to its size. In *Proc. 3rd ACM Symposium on Computational Geometry*, pages 11–20, 1987.

[25] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time using level/linked search trees. *Information and Control*, 68(1-3):170–184, 1986.

[26] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[27] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal of Computing*, 1(1):31–39, 1972.

[28] R. Jacob. *Dynamic Planar Convex Hull.* PhD thesis, University of Aarhus, Denmark, 2002.

[29] S. R. Kosaraju. Localized search in sorted lists. In *Proc. 13th Ann. ACM Symp. on Theory of Computing*, pages 62–69, 1981.

[30] C. Levcopoulos and M. H. Overmars. A balanced search tree with $O(1)$ worst-case update time. *Acta Informatica*, 26:269–277, 1988.

[31] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

[32] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984.

[33] A. Moffat. Adaptive merging and a naturally natural merge sort. In *Proceedings of the 14th Australian Computer Science Conference*, pages 08.1–08.8, 1991.

[34] A. Moffat, O. Petersson, and N. Wormald. Further analysis of an adaptive sorting

algorithm. In *Proceedings of the 15th Australian Computer Science Conference*, pages 603–613, 1992.

[35] A. Moffat, O. Petersson, and N. C. Wormald. Sorting and/by merging finger trees. In *Algorithms and Computation: Third International Symposium, ISAAC '92*, volume 650 of *Lecture Notes in Computer Science*, pages 499–508. Springer-Verlag, 1992.

[36] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Dept. of Computer Science, University of Maryland, College Park, 1989.

[37] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[38] R. Raman. *Eliminating Amortization: On Data Structures with Guaranteed Response Time.* PhD thesis, University of Rochester, New York, 1992. Computer Science Dept., U. Rochester, tech report TR-439.

[39] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[40] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[41] R. Tarjan and C. van Wyk. An $o(n \log\log n)$ algorithm for triangulating a simple polygon. *SIAM Journal of Computing*, 17:143–178, 1988.

[42] A. K. Tsakalidis. AVL-trees for localized search. *Information and Control*, 67(1-3):173–194, 1985.