Rank-Sensitive Data Structures

Iwona Bialynicka-Birula and Roberto Grossi

Dipartimento di Informatica, Università di Pisa
Largo Bruno Pontecorvo 3, 56127 Pisa, Italy
{iwona,grossi}@di.unipi.it

Abstract. Output-sensitive data structures result from preprocessing n items and are capable of reporting the items satisfying an on-line query in $O(t(n) + \ell)$ time, where $t(n)$ is the cost of traversing the structure and $\ell \leq n$ is the number of reported items satisfying the query. In this paper we focus on *rank-sensitive* data structures, which are additionally given a *ranking* of the n items, so that just the top k best-ranking items should be reported at query time, *sorted in rank order*, at a cost of $O(t(n) + k)$ time. Note that k is part of the query as a parameter under the control of the user (as opposed to ℓ which is query-dependent). We explore the problem of adding rank-sensitivity to data structures such as suffix trees or range trees, where the ℓ items satisfying the query form $O(\text{polylog}(n))$ intervals of consecutive entries from which we choose the top k best-ranking ones. Letting $s(n)$ be the number of items (including their copies) stored in the original data structures, we increase the space by an additional term of $O(s(n) \lg^{\epsilon} n)$ memory words of space, each of $O(\lg n)$ bits, for any positive constant $\epsilon < 1$. We allow for changing the ranking on the fly during the lifetime of the data structures, with ranking values in $0 \ldots O(n)$. In this case, query time becomes $O(t(n)+k)$ plus $O(\lg n/\lg\lg n)$ per interval; each change in the ranking and each insertion/deletion of an item takes $O(\lg n)$ time; the additional term in space occupancy increases to $O(s(n) \lg n/\lg\lg n)$.

1 Introduction

Output-sensitive data structures are at the heart of text searching [13], geometric searching [5], database searching [28], and information retrieval in general [3, 31]. They are the result of preprocessing n items (these can be textual data, geometric data, database records, multimedia, or any other kind of data) into $O(n\,\text{polylog}(n))$ space in such a way, as to allow quickly answering on-line queries in $O(t(n) + \ell)$ time, where $t(n) = o(n)$ is the cost of querying the data structure (typically $t(n) = \text{polylog}(n)$). The term output-sensitive means that the query cost is proportional to ℓ, the number of reported items satisfying the query, assuming that $\ell \leq n$ can be much smaller than n. In literature, a lot of effort has been devoted to minimizing $t(n)$, while the dependency on the *variable* cost ℓ has been considered unavoidable because it depends on the items satisfying the given query and cannot be predicted before querying.

In recent years we have been literally overwhelmed by the electronic data available in fields ranging from information retrieval, through text processing and computational geometry to computational biology. For instance, the number ℓ of items reported by search engines can be so huge as to hinder any reasonable attempt at their post-processing. In other words, n is very large but ℓ is very large too (even if ℓ is much smaller than n). Output-sensitive data structures are too optimistic in a case such as this, and returning all the ℓ items is not the solution to the torrent of information.

Motivation. Search engines are just one example; many situations arising in large scale searching share a similar problem. But what if we have some preference regarding the items stored in the output-sensitive data structures? The solution in this case involves assigning an application-dependent *ranking* to the items, so that the top k best-ranking items among the ℓ ones satisfying an on-line query can be returned *sorted in rank order*. (We assume that $k \leq \ell$ although the general bound is indeed for $\min\{k, \ell\}$.) Note that the overload is significantly reduced when $k \ll \ell$. For example, PageRank [24] is at the heart of the Google engine, but many other rankings are available for other types of data. Z-order is useful in graphics, since it is the order in which geometrical objects are displayed on the screen [14]. Records in databases can be returned in the order of their physical location (to minimize disk seek time) or according to a time order (e.g. press news). Positions in biological sequences can be ranked according to their biological function and relevance [13]. These are just basic examples, but more can be found in statistics, geographic information systems, etc.

Our results for rank-sensitive data structures. In this paper, we study the theoretical framework for a class of *rank-sensitive* data structures. They are obtained from output-sensitive data structures such as suffix trees [30, 27] or range trees [5], where the ℓ items satisfying the query form $O(\text{polylog}(n))$ intervals of consecutive entries each. For example, string searching in suffix trees and tries goes along a path leading to a node v, whose descending leaves represent the ℓ occurrences to report, say, from leaf v_1 to leaf v_2 in symmetrical order. In one-dimensional range searching, two paths leading to two leaves v_1 and v_2 identify the ℓ items lying in the range. In both cases, we locate an interval of consecutive entries in the symmetrical order, from v_1 to v_2. In two-dimensional range trees, we locate $O(\lg n)$ such (disjoint) intervals. For higher dimensions, we have $O(\text{polylog}(n))$ disjoint intervals.

As previously said, for this class of output-sensitive data structures, we obtain the retrieved items as the union of $O(\text{polylog}(n))$ disjoint intervals. We provide a framework for transforming such intervals into rank-sensitive data structures from which we choose the top k best-ranking items satisfying a query. We aim at a cost dependency on the parameter k specified by the user rather than on the query-dependent term ℓ. Let *rank* denote a ranking function such that $\mathit{rank}(v_1) < \mathit{rank}(v_2)$ signifies that item v_1 should be preferred to item v_2. We do not enter into a discussion of the quality of the ranking adopted; for us, it just induces a total order on the importance of the items to store. Let $s(n)$ be the number of items (including their copies) stored in any such data structure D. Let $O(t(n) + \ell)$ be its query time and let $|D|$ be the number of memory words of space it occupies, each word composed of $O(\lg n)$ bits. We obtain a rank-sensitive data structure D', with $O(t(n) + k)$ query time, increasing the space to $|D'| = |D| + O(s(n) \lg^{\epsilon} n)$ memory words, for any positive constant $\epsilon < 1$.

We allow for changing *rank* on the fly during the lifetime of the data structure D', with ranking values in the range from 0 to $O(n)$. In this case, query time becomes $O(t(n) + k)$ plus $O(\lg n / \lg \lg n)$ per interval and each change in the ranking takes $O(\lg n)$ time per item copy. Our solution operates in *real time* as we discuss later. When D allows for insert and delete operations on the set of items, we obtain an additive cost of $O(\lg n / \lg \lg n)$ time per query operation and $O(\lg n)$ time per update operation in D'. The space occupancy is $|D'| = |D| + O(s(n) \lg n / \lg \lg n)$ memory words. The preprocessing cost of D' is dominated by sorting the items according to *rank*, plus the preprocessing cost of D.

Attacking the problem. While ranking itself has been the subject of intense theoretical investigation in the context of search engines [17, 18, 24], we could not find any explicit study pertaining to ranking in the context of data structures. The only published data structure of this kind is the inverted lists [31] in which the documents are sorted according to their rank order. McCreight's paper on priority search trees [19] refers to enumeration in increasing order along the y-axis but it does not indeed discuss how to report the items in sorted order along the y-axis. An indirect form of ranking can be found in the (dynamic) rectangular intersection with priorities [15] and in the document listing problem [21].

For our class of output-sensitive data structures, we can formulate the ranking problem as a geometric problem. We are given a (dynamic) list L of n entries, where each entry $e \in L$ has an associated value $\mathit{rank}(e) \in 0 \ldots O(n)$. The list induces a linear order on its entries, such that $e_i < e_j$ if and only if e_i precedes e_j in L. Let us indicate by $\mathit{pos}(e)$ the (dynamic) position of e in L (but we do not maintain *pos* explicitly). Hence, $e_i < e_j$ if and only if $\mathit{pos}(e_i) < \mathit{pos}(e_j)$. We associate point $(\mathit{pos}(e), \mathit{rank}(e))$ with each entry $e \in L$. Then, given e_i and e_j, let e' be the kth entry in rank order such that $\mathit{pos}(e_i) \le \mathit{pos}(e') \le \mathit{pos}(e_j)$. We perform a three-sided or $1\frac{1}{2}$-dimensional query on $\mathit{pos}(e_i) \ldots \mathit{pos}(e_j)$ along the x-axis, and $0 \ldots \mathit{rank}(e')$ along the y-axis.

Priority search trees [19] and Cartesian trees [29] are among the prominent data structures supporting these queries, but do not provide items in sorted order (they can end up with half of the items unsorted during their traversal). Since we can identify the aforementioned e' by a variation of [10], in $O(k)$ time, we can retrieve the top k best-ranking items in $O(k + \lg n)$ time in *unsorted order*. Improvements to get $O(k)$ time can be made using scaling [12] or persistent data structures [6, 8, 16]. Subsequent sorting reports the items in $O(k \lg k)$ time using $O(n)$ words of memory.

What if we adopt the above solution in a *real-time* setting? Think of a server that provides items in rank order on the fly, or any other similar real-time application in which guaranteed response time is mandatory. Given a query, the above solution and its variants can only start listing the first items after $O(t(n)+k\lg k)$ time! In contrast, our solution works in real-time by using more space. After $O(t(n))$ time, it provides each subsequent item in $O(1)$ worst-case time according to the rank order (i.e. the qth item in rank order is listed after $c_1 t(n) + c_2 q$ steps, for $1 \leq q \leq k$ and constants $c_1, c_2 > 0$).

Persistent data structures can attain real-time performance, in an efficient way, only in a static setting. Let us denote L's entries by $e_0, e_1, \ldots, e_{n-1}$, and build persistent sublists of 2^r consecutive entries, for $r = 0, 1, \ldots, \lg n$. Namely, for fixed r, we start from the sublist containing $e_0, e_1, \ldots, e_{2^r-1}$ in *sorted rank order*. Then, for $i = 2^r, \ldots, n-1$, we remove entry e_{i-2^r} and add e_i using persistence to create a new version of the sorted sublist. Now, given our query with e_i and e_j, we compute the largest r such that $2^r \leq j - i$. Among the versions of the sublists for 2^r entries, we take the one starting at e_i and the one ending in e_j. Merging these two lists on the fly for k steps solves the ranking problem. This solution has two drawbacks. First, it uses more space than our solution. Second, it is hard to *dynamize* since a single entry changing in L can affect $\Theta(n)$ versions in the worst case. (Also the previous solutions based on persistence, priority search trees and Cartesian trees suffer similar problems in the dynamic setting.) We extend the notion of Q-heap [11] to implement our solution, introducing *multi-Q-heaps* described in Section 3.

2 The Static Case and its Dynamization

Our starting point is a well-known scheme adopted for two-dimensional range trees [5]. Following the global rebuilding technique described in [23], we can restrict our attention to values of n in the range $0 \ldots O(N)$ where $n = \Theta(N)$. Consequently, we use lookup tables tailored for N, so that when the value of N must double or halve, we also rebuild these tables in $o(N)$ time. Our word size is $O(\lg N)$. As can be seen from [23], time bounds can be made worst-case.

We recall that the interval is taken from the list of items $L = e_0, e_1, \ldots, e_{n-1}$, indicating with $pos(e_i)$ the dynamic position of e_i in L (but we do not keep pos explicitly) and with $rank(e_i)$ its rank value in $0 \ldots O(N)$. We use a special rank value $+\infty$ that is larger than the other rank values; multiple copies of $+\infty$ are each different from the other (and take $O(\lg N)$ bits each).

2.1 Static case on a single interval

We employ a weight-balanced B-tree W [2] as the skeleton structure. At the moment, suppose that W has degree exactly two in the internal nodes and that the n items in L are stored in the leaves of W, assuming that each leaf stores a single item. For each node $u \in W$, let $R(u)$ denote the explicit sorted list of the items in the leaves descending from u, according to *rank* order. If u_0 and u_1 are the two children of u, we have that $R(u)$ is the merge of $R(u_0)$ and $R(u_1)$. Therefore, we can use 0s and 1s to mark the entries in $R(u)$ that originate, respectively, from $R(u_0)$ and $R(u_1)$. We obtain $B(u)$, a bitstring of $|R(u)|$ bits, totalizing $O(n \lg n)$ bits, hence $O(n)$ words of memory, for the entire W (see [4]).

Rank query works as expected [5]. Given entries e_i and e_j in L, we locate their leaves in W, say v_i and v_j. We find their least common ancestor w in W (the case $v_i = v_j$ is trivial). On the path from w to v_i, we traverse $O(\lg n)$ internal nodes. If during this traversal, we go from node u to its left child u_0, we consider the list $R(u_1)$, where u_1 is the right child of u. Analogously, on the path from w to v_j, if we go from node u to its right child u_1, we consider list $R(u_0)$ for its left child. In all other cases, we skip the nodes (including w and its two children). Clearly, we include v_i and v_j if needed.

At this point, we reduce the rank-sensitive query for v_i and v_j to the problem of selecting the top k best-ranking items from $O(\lg n)$ rank-sorted lists $R()$, containing integers in $0 \ldots O(N)$. Following Chazelle's approach, we do not explicitly store the lists $R()$, but keep only the bitstrings $B()$ and the additional machinery for translating bits in $B()$ into entries in $R()$, which occupies $O(n \lg^{\epsilon} n)$ words of memory, for any positive constant $\epsilon < 1$. (See Lemma 2 in Section 4 of [4].) As a result, we can retrieve the sorted items of lists $R()$ using Chazelle's approach.

2.2 Polylog intervals in the dynamic case

In the general case, we are left with the problem of selecting the top k best-ranking items from $O(\text{polylog}(n))$ rank-sorted *dynamic* lists $R()$, containing integers in $0 \ldots O(N)$. We cannot use Chazelle's machinery in the dynamic setting. We maintain the degree b of the nodes in the weight-balanced B-tree W, such that $(\beta/4) \lg n / \lg \lg n \leq b \leq (4\beta) \lg n / \lg \lg n$, for a suitable constant in $0 < \beta < 1$. As a result from [2], the height of the tree is $O(\lg n / \lg b) = O(\lg n / \lg \lg n)$. We also explicitly store the lists $R()$, totalizing $O(n)$ words per level of W, and thus yielding $O(n \lg n / \lg \lg n)$ words of memory. Note that the cost of splitting/merging a node $u \in W$ along with $R(u)$ can be deamortized [2].

To enable the efficient updating of all the lists $R()$, we use a variation of dynamic fractional cascading described in [25], which performs efficiently on graphs of a non-constant degree. Fractional cascading does not increase the overall space complexity. At the same time, for a given element e of list $R(u)$, it allows locating the predecessor (in rank order) of e in $R(u')$ when u' is a child or parent of u. This locating is performed in time $O(\lg b + \lg \lg n)$ which amounts to $O(\lg \lg n)$ under our assumption concerning b, the degree of the tree.

Let us consider a single interval identified by a rank query. It is described by two leaves v_i and v_j, along with their least common ancestor $w \in W$. However, we encounter $O(\lg n / \lg \lg n)$ lists $R()$ in each node u along the path from w to either v_i or v_j. For any such node u, we must consider the lists for u's siblings either to its left or its right. So we have to merge $O((\lg n / \lg \lg n)^2)$ lists on the fly. But we can only afford $O(\lg n / \lg \lg n)$ time.

We solve this multi-way merging problem by introducing *multi-Q-heaps* in Section 3, extending Q-heaps [11]. A multi-Q-heap stores $O(\lg n / \lg \lg n)$ items from a bounded universe $0 \ldots O(N)$, and performs constant-time search, insertion, deletion, and find-min operations. In particular, searching and finding can be *restricted* to any *subset* of its entries, still in $O(1)$ time. Each instance of a multi-Q-heap requires just $O(1)$ words of memory. These instances share common lookup tables occupying $o(N)$ memory words. We refer the reader to Theorem 2 in Section 3 for more details.

We employ our multi-Q-heap for the rank values in each node $u \in W$. This does not change the overall space occupancy, since it adds $O(n)$ words, but it allows us to handle rank queries in each node u in $O(1)$ time per item as follows. Let $d = \alpha \lg N / \lg \lg N$ be the maximum number of items that can be stored in a multi-Q-heap (see Theorem 2). We divide the lists $R()$ associated with u's children into d clusters of d lists each. For each cluster, we repeat the task recursively, with a constant number of levels and $O(\text{polylog}(n)/d)$ multi-Q-heaps. We organize these multi-Q-heaps in a hierarchical pipeline of constant depth. For the sake of discussion, let's assume that we have just depth 2. We employ a (first-level) multi-Q-heap, initially storing d items, which are the minimum entry for each list in the cluster. We employ further d (second-level) multi-Q-heap of d entries each, in which we store a copy of the minimum element of each cluster. To select the top k best-ranking leaves, we extract the k smallest entries from the lists by using the above multi-Q-heaps: We first find the minimum entry, x, in one of the second-level multi-Q-heaps, and identify the corresponding first-level multi-Q-heap. From this, we identify the list containing x. We take the entry, y, following x in its list. We extract x from the first-level multi-Q-heap and insert y. Let z be the new minimum thus resulting in the first level. We extract x from the suitable second-level multi-Q-heap and insert z. By repeating this task k times, we return the k leaves in rank-sensitive fashion.

This does not yet solve our problem. Consider the path from, say, v_i to its ancestor w. We have $O(\lg n / \lg \lg n)$ lists for each node along the path. Fortunately, our multi-Q-heaps allow us to handle any subset of these lists, in constant time. The net result is that we need to use just $O(\lg n / \lg \lg n)$ multi-Q-heaps for the entire path. For each node u in the path, the find-min operation is limited to the lists corresponding to a subset of u's sibling at its right. They form a contiguous range, which we can easily manage with multi-Q-heaps. Hence, we can apply the above 2-level organization, in which we have $O(\lg n / \lg \lg n)$ multi-Q-heaps in the path from v_i to w in the second level. (An analogous approach is for the path from v_j to w.) In this way, we can perform a multi-way merging on the fly for finding the least k keys in sorted rank order, in $O(k + \lg n / \lg \lg n)$ time. Note that the bound is real-time as claimed. In the case of polylog intervals, we use an additional multi-Q-heap hierarchical organization (of constant depth) to merge the items resulting from processing each interval separately.

We now describe how to handle rank changes of entries in L, as well as insertions and deletions in L. Changing the rank of entry e_i, say in leaf $v_i \in W$ is performed in a top-down fashion. It affects the nodes on the path from the root of W to v_i. The list $R(u)$ for each node u along this path contains a copy of e_i but whose rank no longer complies with the ordering of the list. This element is extracted from the list and inserted into the correct place on this list. Both the element itself and the new correct place can be located in the list associated with the root in $O(\lg n)$ time. Next, using the fractional cascading structure, we can relocate the copy of e_i in the list for the next node in the downward path to v_i, having already done it in the current node. This takes $O(\lg \lg n)$ time per node, thus yielding $O(\lg n)$ total time to relocate the copy of e_i in all the lists of the path. As for the insertions in L (and also in W), they follow the approach in [2]; moreover, the input item e has its $rank(e)$ value, in the range $0 \ldots O(N)$, inserted into the lists $R()$ of the ancestor nodes as described above. Deletions are simply implemented as weak, changing the rank value of deleted items to $+\infty$. When their number is sufficiently large, we apply rebuilding as in [23]. If the original data structure contains multiple copies of the same item (as in the case of a range tree) then the update in the rank-sensitive structure is applied separately to the individual copies.

We obtain the following result. Let D be an output-sensitive data structure for n items, where the ℓ items satisfying a query on D form $O(\text{polylog}(n))$ intervals of consecutive entries. Let $O(t(n) + \ell)$ be its query time and $s(n)$ be the number of items (including their copies) stored in D.

Theorem 1. *We can transform D into a static rank-sensitive data structure D', where query time is $O(t(n) + k)$ for any given k, thus reporting the top k best-ranking items among the ℓ listed by D. We increase the space by an additional term of $O(s(n) \lg^{\epsilon} n)$ memory words of space, each of $O(\lg n)$ bits, for any positive constant $\epsilon < 1$. For the dynamic version of D and D', we allow for changing the ranking of the items, with ranking values in $0 \ldots O(n)$. In this case, query time becomes $O(t(n) + k)$ plus $O(\lg n / \lg \lg n)$ per interval. Each change in the ranking and each insertion/deletion of an item take $O(\lg n)$ time for each item copy stored in the original data structure. The additional term in space occupancy increases to $O(s(n) \lg n / \lg \lg n)$.*

3 Multi-Q-Heaps

The multi-Q-heap is a relative of the Q-heap [11]. Q-heaps provide a way to represent a sub-logarithmic set of elements in the universe $[N] = 0 \ldots O(N)$, so that such operations as inserting, deleting or finding the smallest element can be executed in $O(1)$ time in the worst case. The price to pay for the speed is the need to set up and store lookup tables in $o(N)$ time and space. These tables, however, need only to be computed once as a bootstrap cost and can be shared among any number of Q-heap instances. Our multi-Q-heap is functionally more powerful than Q-heap, as it allows performing operations on any *subset* of d common elements, where $d \leq \alpha \lg N / \lg \lg N$ for a suitable positive constant $\alpha < 1$. Naturally, this could be emulated by maintaining Q-heaps for all the different subsets of the elements, but that solution would be exponential in d (for each instance!), while our multi-Q-heap representation requires two or three memory words and still supports constant-time operations. Our implementation based on lookup tables is quite simple and does not make use of multiplications or special instructions (see [9, 26] for a thorough discussion of this topic). We first describe a simpler version (to be later extended) supporting the following:

– Create a heap for a given list of elements.
– Find the minimum element within a given range.
– Find an element within a given range of items.
– Update the element at a given position.

In the rest of the section, we prove the following result.

Theorem 2. *There exists a constant $\alpha < 1$ such that d distinct integers in $0 \ldots O(N)$ (where $d \leq \alpha \lg N / \lg \lg N$) can be maintained in a multi-Q-heap supporting search, insert, delete, and find-min operations in constant time per operation in the worst case, with $O(d)$ words of space. The multi-Q-heap requires a set of pre-computed lookup tables taking $o(N)$ construction time and space.*

3.1 High-level implementation

The d elements are integers from $[N]$. We can refer to their binary representations of $w = \lceil \lg[N] \rceil$ bits each. These strings can be used to build a compacted trie on binary strings of length w. However, instead of labeling the leaves of the compact trie with the strings (elements) they correspond to, we keep just the trie shape and the skip values contained in its internal nodes, like in [1, 7]. We store the d elements and their satellite data in a separate table. To provide a connection between the trie and the values, we store a permutation which describes the relation between the order of elements in the trie and the order in which they are stored in the table.

When searching for an element, we first perform a blind search on the trie [1, 7]. Next we access the table corresponding to the found element and we compare it with the sought one. Note that this way we only access the table of values in one place, while the rest of the search is performed on the trie. With an assumption about the maximum number of elements stored in the multi-Q-heap, we can encode both the trie and the permutation as two single memory words. The operations are then performed on these encodings and only the relevant entries in the value table are accessed, which guarantees constant time. The operations on the encoded structures are realized using lookup tables and bit operations.

To support multi-Q-heap operations, we store a single structure containing all the elements. We implement all the extended operations so as to consider only the given subset of the elements while maintaining constant time. We assume a word size of $w = O(\lg N)$ bits. We use d to refer to the number of items stored in the multi-Q-heap. We assume $d \leq \alpha \lg N / \lg \lg N$ for some suitable constant $\alpha < 1$. We use $x_0, x_1, \ldots, x_{d-1}$ to refer to the list of items stored in the multi-Q-heap. For our case, the order defined by the indices is relevant (when using multi-Q-heaps in the nodes of the weight-balanced B-tree of Section 2).

3.2 Multi-Q-heap: Representation

The multi-Q-heap can be represented as a triplet (S, τ, σ), where S is the array of elements stored in the structure, τ is the encoding of the compact trie and σ is an encoding of the permutation. The array S stores the elements $x_0, x_1, \ldots, x_{d-1}$ in that order and their satellite data. Each element occupies a word of space.

The encoding of the trie, τ, can be defined in the following fashion. First, let us encode the shape of the binary tree of which it consists. This tree is binary, with no unary nodes and edges implicitly labeled with either 0 or 1. We can encode it by traversing the tree in inorder (visiting first 0 edges and then 1 edges) and outputting the labels of the edges traversed. This encoding can be decoded unambiguously and requires $4d - 4$ bits, since each edge is traversed twice and there are $2d - 2$ edges in the trie. Next, we encode the skip values. The internal nodes (in which the skip values are stored) are ordered according to their inorder which leads to an ordered list of skip values. Each skip value is stored in $\lceil \lg w \rceil$ bits, so the encoding of the list takes $(d-1)\lceil \lg w \rceil$ bits. For a suitable value of α the complete encoding of the trie does not exceed $1/4 \lg N$ bits and hence can be stored in one word of memory.

The permutation σ reflects the array order $x_0, x_1, \ldots, x_{d-1}$ with respect to the order of these elements sorted by their values (which is the same as the in-order of the corresponding leaves in the trie). There are $d!$ possible permutations, so we choose α so that $\lg d! < 1/4 \lg N$ and the encoding on the permutation fits in one word of memory. We use the encoding described in [22], which takes linear time to rank and unrank a permutation, hence to encode and decode it.

3.3 Multi-Q-heap: Supported operations

The *Init* operation sets up all the lookup tables required for implementing the multi-Q-heap. It needs to be performed only once. See section 3.4 for details concerning the lookup tables. These lookup tables are used in the implementations of the operations described below. If invoked multiple times, only the first is effective.

The *Create* operation takes the array S of values $x_0, x_1, \ldots, x_{d-1}$ and sets up the structures τ and σ. It takes the time required to construct the compact trie for d elements, hence $O(d)$.

The function *Findmin* returns the smallest element among the elements $x_i, \ldots, x_j$ stored in the multi-Q-heap. We implement it using the lookup table *Subheap* and *Index*. We use $\mathit{Subheap}[\tau, \sigma, i, j]$ to obtain τ' and σ', the structure for elements $x_i, \ldots, x_j$. We then use $\mathit{Index}[\sigma', 1]$ to obtain the array index of the smallest element in the range.

The function *Search* searches the subset of elements $x_i, \ldots, x_j$ stored in the multi-Q-heap and returns the index of the element in the multi-Q-heap which is smallest among those not smaller than y, where $y \in [N]$ can be any value. As previously, we use $\mathit{Subheap}[\tau, \sigma, i, j]$ to obtain τ' and σ', the subheap for elements $x_i, \ldots, x_j$. We then search the reduced trie for x', the first half (bitwise) of x, by looking up $u = \mathit{Top}[\tau', x']$. Next, using $\mathit{LDescendant}[\tau', u]$, we identify one of the strings descending from u and compare this string with x' to compute their longest common prefix length *lcp*. This computation can be done in constant time with another lookup table, which is standard and is not described. If $\mathit{lcp} < 1/2 \lg N$, then $\mathit{LDescendant}[\tau', u]$ identifies the sought element. If $\mathit{lcp} = 1/2 \lg N$, we continue the search in the bottom part of the trie by setting $u = Top[\tau', x', u]$. Also here $LDescendant[\tau', u]$ provides the answer.

The *Update* operation replaces the element x_r in the array S with y, where $y \in [N]$ can be any value. It updates τ and σ accordingly. We first simulate the search for y in τ, as described in the previous paragraph to find the rank i of y among $x_0, \dots x_d$ and use this together with the table $UpdatePermutation[\sigma, r, i]$ to produce the updated permutation. We then use values obtained during the simulated blind search for y in τ to obtain values needed to access the *UpdateTrie* table. During the search we find the node u at which the search for y ends (in the second half of the trie in the case the search gets that far) and the *lcp* obtained by comparing its leftmost descendant with y. We use $Ancestor[\tau, u, lcp]$ for identifying the node whose parent edge is to be split for inserting. The *lcp* is the skip value the parameter c depends on the bit at position $lcp + 1$ of y. With this information, we access *UpdateTrie*.

3.4 Multi-Q-heap:Lookup tables

This section describes the lookup tables required to perform the operations described in the previous section. The number of tables can be reduced, but at the expense of the clarity of the implementation description.

The *Index* table provides a way for obtaining the array index of an element given the inorder position of its corresponding leaf in the trie (let us call this the trie position). It contains the appropriate array index entry for every possible permutation and trie position. The space occupancy is $2^{1/4 \lg N} \times d \times \lg d = N^{1/4} \times d \times \lg d = o(N)$.

The $Index^{-1}$ table is the inverse of *Index* in the sense that it provides a way of obtaining a trie position from an index, by containing a position entry for every possible permutation and index. The space occupancy is the same as for *Index*.

The *Subheap* table provides a means of obtaining a new subheap structure, (S, τ', σ'), from a given one (S, τ, σ). The new subheap structure uses the same array S, but takes into account only the subset $x_i, \dots, x_j$ of its items. Note that only τ, σ, i, and j are needed to determine τ' and σ' and not the values stored in S. The new trie τ' is obtained from the old trie τ by removing leaves not corresponding to $x_i, \dots, x_j$ (these can be identified using σ). The new permutation σ' is obtained from the old one σ by extracting all the elements with values $i, \dots, j$ and moving them to the beginning of the permutation (without changing their relative order) so that they now correspond to the appropriate $j - i + 1$ leaves of the reduced trie. The space occupancy of *Subheap* is $2^{1/4 \lg N} \times 2^{1/4 \lg N} \times d \times d \times 1/4 \lg N \times 1/4 \lg N = N^{1/2} \times d^2 \times (1/4 \lg N)^2 = o(N)$.

The *Top* and *Bottom* tables allow searching for a value in the trie. The searching for a value must be divided into two stages, because a table which in one dimension is indexed with a full value, one of $O(N)$ possible, would occupy too much space. We therefore set up two tables: *Top* for searching for the first $1/2 \lg N$ bits of the value and *Bottom* for the remaining. The table *Top* contains entries for every possible trie τ and x', the first $1/2 \lg N$ bits of some sought value x. The value in the table specifies the node of τ (with nodes specified by their inorder position) at which the blind search [1, 7] for x' (starting from the root of the trie) ends. The table *Top* contains entries for every possible trie τ, x'' (the second $1/2 \lg N$ bits of some sought value x) and an internal node of the trie v. The value in the table specifies the node of τ at which the blind search [1, 7] for x'' ends, but in this case the blind search starts from v instead of from the root of the trie. The space occupancy of *Top* is $2^{1/4 \lg N} \times 2^{1/2 \lg N} \times \lg d = N^{3/4} \times \lg d = o(N)$ and the space occupancy of *Bottom* is $2^{1/4 \lg N} \times 2^{1/2 \lg N} \times d \times \lg d = N^{3/4} \times d \times \lg d = o(N)$.

The *UpdateTrie* table specifies a new multi-Q-heap and permutation which is created from a given one by removing the leaf number i from τ and inserting instead a new leaf. The new leaf is the c child of a node inserted on the edge leading to u. This new node has skip value s. The space occupancy is $2^{1/4 \lg N} \times d \times 2 \times 2^{\lg \lg N} \times d \times 1/4 \lg N \times 1/4 \lg N = N^{1/4} \times d^2 \times 1/8 \lg^3 N = o(N)$.

The *UpdatePermutation* table specifies the permutation obtained from σ if the element with index r is removed and an element ranking i among the original elements of the multi-Q-heap is inserted in its place. The space occupancy is $2^{1/4 \lg N} \times d \times d \times 1/4 \lg N = N^{1/4} \times d^2 \times 1/4 \lg N = o(N)$.

The *LDescendant* table specifies the leftmost descending leaf of node u in τ. Its space occupancy is $2^{1/4 \lg N} \times d \times \lg d = N^{1/4} \times d \times \lg d = o(N)$.

The *Ancestor* table specifies the shallowest ancestor of u having a skip value equal to or greater than s. The space occupancy is $2^{1/4 \lg N} \times d \times 2^{\lg \lg N} \times \lg d = N^{1/4} \times d \times \lg N \times \lg d = o(N)$

We will describe the general case of multi-Q-heap and discuss an example in the full version. Here we only say that we need also to encode a permutation π in a single word since $x_0, \ldots, x_{d-1}$ can be further permuted due to the insertions and deletions. An arbitrary subset is represented by a bitmask that replaces the two small integers i and j delimiting a range. The sizes of the lookup tables in Section 3.4 increase but still remain $o(N)$.

References

1. M. Ajtai, M. Fredman, and J. Komlòs. Hash functions for priority queues. *Information and Control*, 63(3):217–225, December 1984.
2. Lars Arge and Jeffrey S. Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32:1488–1508, 2003.
3. Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.
4. Bernard Chazelle. A functional approach to data structures and its use in multi-dimensional searching. *SIAM Journal on Computing*, 17(3):427–462, June 1988.
5. Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwartzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 1997.
6. James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Computer and System Sciences*, 38(1):86–124, 1989.
7. P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *J. ACM*, 46:236–280, 1999.

8. Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003.
9. Faith E. Fich. Class notes CSC 2429F: Dynamic data structures, 2003. Department of Computer Science, University of Toronto, Canada.
10. Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993, June.
11. Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *JCSS*, 48(3):533–551, 1994.
12. Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In *STOC '84*, 135–143, Washington, D.C., 1984.
13. Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.
14. D. Hearn and M. Baker. *Computer Graphics with OpenGL.* Prentice-Hall, 2003.
15. Haim Kaplan, Eyal Molad, and Robert E. Tarjan. Dynamic rectangular intersection with priorities. In *STOC '03*, 639–648. ACM Press, 2003.
16. Kitsios, Makris, Sioutas, Tsakalidis, Tsaknakis, and Vassiliadis. 2-D spatial indexing scheme in optimal time. In *ADBIS: East European Symposium on Advances in Databases and Information Systems.* LNCS, 2000.
17. Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, September 1999.
18. R. Lempel and S. Moran. SALSA: the stochastic approach for link-structure analysis. *ACM Transactions on Information Systems*, 19(2):131–160, 2001.
19. Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.
20. C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *SODA '03*, 618–627, 2003.
21. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666. Society for Industrial and Applied Mathematics, 2002.
22. Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, September 2001.
23. Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science.* Springer-Verlag, 1983.
24. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. *Tech. rep, Stanford University, Stanford, CA*, 1998.
25. Rajeev Raman. *Eliminating amortization: on data structures with guaranteed response time.* PhD thesis, Rochester, NY, USA, 1993.
26. Mikkel Thorup. On AC^0 implementations of fusion trees and atomic heaps. In *Proceedings of the fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-03)*, pages 699–707, New York, January 12–14 2003. ACM Press.
27. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.
28. Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.
29. Jean Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, April 1980.
30. P. Weiner. Linear pattern matching algorithms. In *Conference Record, IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.
31. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes: Compressing and indexing documents and images.* Morgan Kaufmann Pubs. Inc., 1999.