Static Dictionaries Supporting Rank

Venkatesh Raman and S. Srinivasa Rao

The Institute of Mathematical Sciences, C. I. T. Campus,
Chennai 600 113. India

Abstract. A static dictionary is a data structure for storing a subset S of a finite universe U so that membership queries can be answered efficiently. We explore space efficient structures to also find the rank of an element if found. We first give a representation of a static dictionary that takes $n \lg m + O(\lg\lg m)$ bits of space and supports membership and rank (of an element present in S) queries in constant time, where $n = |S|$ and $m = |U|$. Using our structure we also give a representation of a m-ary cardinal tree with n nodes using $n\lceil\lg m\rceil + 2n + o(n)$ bits of space that supports the tree navigational operations in $O(1)$ time, when m is $o(2^{\lg n/\lg\lg n})$. For arbitrary m, we give a structure that takes the same space and supports all the navigational operations, except finding the child labeled i (for any i), in $O(1)$ time. Finding the child labeled i in this structure takes $O(\lg\lg\lg m)$ time.

1 Introduction and Motivation

A static dictionary is a data structure for storing a subset S of a finite universe U so that membership queries can be answered efficiently. This problem has been widely studied and various structures have been proposed to support membership in constant time [8, 7, 4, 12] in slightly different models. Our focus in this paper is to also support the rank operation which asks for the number of elements in the set less than or equal to the given element. Our model of computation is an extended RAM machine model that permits constant time arithmetic and boolean bitwise operations.

Our motivation for studying rank operation comes from the recent succinct representation of m-ary cardinal trees[3]. A *cardinal tree* of degree k is a rooted tree in which each node has k positions for an edge to a child. A binary tree is a cardinal tree of degree 2. An *ordinal tree* is a rooted tree of arbitrary degree in which the children of each node are ordered. The cardinal tree representation of [3] essentially has two parts: one part giving the ordinal information of the tree using $2n + o(n)$ bits and the other part storing the children information of each node in the tree using $n \lg m$ bits where n is the number of nodes. To navigate around the tree, in particular, to find the child labeled i of a node, we need to find the rank of the element i in the ordinal part information of the node. The representation of Benoit et. al.[3] supports this operation in $O(\lg\lg m)$ time. Clearly, to perform this operation in constant time, a structure for static dictionary taking $n \lg m + o(n)$ bits supporting the rank operation in constant time suffices. Though we could achieve this when m is $o(2^{\lg n/\lg\lg n})$, for the general case we have a structure that supports rank and membership in $O(\lg\lg\lg m)$ time.

If we want to support the rank operation for every element in the universe, there is a lower bound of $\Omega(\lg\lg m)$ time per query even if the space used is polynomial in n [1]. Willard [14] gave a structure that answers rank (and hence membership) queries in $O(\lg\lg m)$ time using $O(n\lg m)$ bits of space. Fiat et. al.[7] gave a structure that answers membership queries in constant time and rank queries in $O(\lg n)$ time using $n\lg m + O(\lg\lg m + \lg n)$ bits of space. The structure given by Pagh [12] answers membership and rank queries in constant time when the size of the set n is $\omega(m\lg\lg m/\lg m)$, using $n\lg(m/n) + O(n)$ bits. Our focus here is to support rank queries for only the elements that are present in the given set.

The only structure we know of to support membership and rank for those elements found is due to Benoit et. al.[3] en route to their efficient cardinal tree representation. Their structure supports membership and rank in $O(\lg\lg m)$ time using at most $n\lg m$ bits. We give an alternate structure that supports both these operations in $O(1)$ time using $n\lg m + O(\lg\lg m) - \Theta(n)$ bits. As a matter of fact, the structure due to Benoit et. al. supports both these operations in constant time using at most $n\lg m$ bits as long as $n > \lg m$ whereas our structure supports both these operations using the same time and space as long as $n > \lg\lg m$. In the smaller range, both these structures take $O(\lg n)$ time if only $n\lg m$ bits are allowed.

In Section 2, we give a space efficient static dictionary structure that answers membership and rank queries in constant time. This structure builds up on the recent enhancement of Pagh[12] of the FKS[8] dictionary and uses $n\lg m + O(n + \lg\lg m)$ bits. In Section 3, we use an interesting idea to remove the $O(n)$ term in the space complexity of the structure. In this section, we also outline space efficient structures to support the select operation (find the j-th smallest element in the given set). In Section 4, we outline the m-ary cardinal tree representation of Benoit et. al.[3] and explain how our rank dictionary structure can be used to improve the running time from $O(\lg\lg m)$ to $O(\lg\lg\lg m)$ for finding the child labeled i, if exists, for any i. We also illustrate another improvement to the structure so that all the navigational operations can be supported in constant time if m is $o(2^{\lg n/\lg\lg n})$.

2 A rank structure taking $n\lg m + O(n + \lg\lg m)$ bits

Fredman et. al.[8] have given a structure that takes $n\lg m + O(\lg\lg m + n\sqrt{\lg n})$ bits and supports membership in $O(1)$ time. Schmidt and Seigel [13] have improved this space complexity to $n\lg m + O(\lg\lg m + n)$ bits. We refer to this structure as the FKS dictionary in the later sections.

The original FKS construction to store a set S has four basic steps:

- A function $h_{k,p}(x)$ is found that maps S into $[0, n^2 - 1]$ without collisions. It suffices to choose $h_{k,p}(x) = (kx \; mod \; p) \; mod \; n^2$ with suitable $k < p < n^2 \lg m$ where p is a prime. Here, the values k and p depend on the set S.
- Next, a function $h_{\kappa,r}(z)$ is found that maps $h_{k,p}(S)$ into $[0, n-1]$ so that the sum of the squares of the collision sizes is not too large. Again, it suffices to choose $h_{\kappa,r}(z) = (\kappa z \; mod \; r) \; mod \; n$, where r is any prime greater than n^2 and $\kappa \in [0, r]$ so that $\sum_{0 \leq j < n} |h_{\kappa,r}^{-1}(j) \cap h_{k,p}(S)|^2 < 3n$.
- For each non-empty bucket i, a secondary hash function h_i is found that is one-to-one on the collision set. We choose $h_i(z) = (k_i z \; mod \; r) \; mod \; {c_i}^2$, where $k_i \in [0, r-1]$ and c_i is the size of the collision set. The element $x \in S$, is stored in location $C_i + h_i(h_{\kappa,r}(h_{k,p}(x)))$, where $C_i = {c_0}^2 + {c_1}^2 + \cdots + {c_{i-1}}^2$. This locates all n items within a table of size $3n$, say $A^{\star}[1 \ldots 3n]$.
- Finally the table is stored without any vacant locations in an array $A[1...n]$ in the same order.

The composite hash function requires the parameters k, p, κ and r for $h_{k,p}$ and $h_{\kappa,r}$, a table $K[0 \ldots n]$ storing the parameters k_i for secondary hash functions h_i, a table $C[0 \ldots n]$ listing the locations C_i and finally a compression table $D[1 \ldots 3n]$, where $D[j]$ gives the index, within A, of the item (if any) that hashes to the value j in $A^{\star}$. Thus this composite hash function description requires $O(n \lg n + \lg\lg m)$ bits of space.

Schmidt and Siegel [13] first observe that up to $\lfloor \lg n \rfloor + 1$ secondary hash functions are sufficient to store the elements of the set. They also show how to represent this composite hash function using $O(n + \lg\lg m)$ bits of space. We briefly describe their representation below. Parameters k and p require $O(\lg n + \lg\lg m)$ bits each and κ and r take $O(\lg n)$ bits each. The parameters for the secondary hash functions $k_1, k_2, \ldots, k_{(\lfloor \lg n \rfloor + 1)}$ are stored in an array, which takes $O((\lg n)^2)$ bits. The table K contains, for its ith sequence of bits, the integer α_i in unary, if k_{α_i} is the multiplier (the secondary hash key) associated to hash the bucket i, $0 \leq i < n$. This table is an $O(n)$ bit string. We store a $o(n)$ bit auxiliary structure along with this bit string to support rank and select operations[9, 5, 10] on it in constant time. Using this and the array of multipliers, given an i, we can find the multiplier associated with the bucket i in constant time. The table C, which contains the values C_i $(= {c_0}^2 + {c_1}^2 + \cdots + {c_{i-1}}^2)$, is encoded as follows. First the values c_i^2 are stored in a table T_0 in unary notation (in order of appearance, separated by 0's), which is of length at most $4n$. We also store an auxiliary structure of $o(n)$ bits to support rank and select on both the bits, in constant time. Now, given an i, C_i is nothing but the rank of the ith 0 (i.e. $C_i = rank_1(select_0(i))$), which can be found in constant time. For the compression table D, we store a bit string of length $3n$ where the ith bit is a 0 if $A^{\star}[i]$ is empty and 1 otherwise. We also store a $o(n)$ bit auxiliary structure to support rank and select operations on this in constant time. When an element is hashed to a location in D, the rank of the bit in that location in the bit vector representation of D gives the location of the element in the array A.

Now, to obtain rank for the element in the set, we could simply store the rank with each element in the FKS table. However this takes $n \lg m + n \lg n +$

$O(n + \lg\lg m)$ bits. In the rest of this section, we describe how we can get rid of the $n \lg n$ term.

Pagh [12] has observed that each bucket j of the hash table may be resolved with respect to the part of the universe hashing to bucket j. Thus we can save space by compressing the hash table part (i.e. table A above) of the data structure, storing in each location not the element itself, but only a *quotient* information that distinguishes it from the part of U that hashes to this location. The quotient function, slightly modified from that of Pagh is as follows:

$$q_{k,p}(x) = ((x \ div \ p).\lceil p/r \rceil + (k.x \ mod \ p) \ div \ n^2).\lceil r/n \rceil + (\kappa.z \ mod \ r) \ div \ n$$

where $z = (k.x \ mod \ p) \ mod \ n^2$ and the parameters k, p, κ and r are as defined in the FKS perfect hash function. It is easy to see that $q_{k,p}(x)$ for $x \in U$ is $O(m/n)$ (as $(\kappa.z \ mod \ r) \ div \ n < r/n$ and $(k.x \ mod \ p) \ div \ n^2 < p/r$).

Thus the total space to store all the quotient values along with the hash function will be $n \lg(m/n) + O(n + \lg\lg m)$ bits. To find an element, we compute its quotient value, apply the composite hash function to determine a location and check whether the quotient value appears in that location. Now, with each element we also store the rank of the element in the set for an extra space of $n\lceil \lg n \rceil$ bits. Thus if the element is found, we can get its rank from the rank information stored in its location.

Thus we have

Theorem 1. *A static dictionary for a subset S of size n of a finite universe $U = \{1, \ldots, m\}$ can be constructed using $n \lg m + O(n + \lg\lg m)$ bits of space so that membership and rank queries can be answered in $O(1)$ time.*

3 A rank structure taking $n \lg m + O(\lg\lg m)$ bits

In this section we illustrate a method by which the space used by the structure in the last section can be reduced by cn bits for any parameter $c < (1-\epsilon)\lg n$, $0 < \epsilon < 1$. The trick is to store only the last $(\lg n - c)$ bits of the rank instead of storing the entire value of the rank along with each element. Suppose the sorted list of the elements of the set is divided into 2^c blocks of size roughly $n/2^c$ each. Then the information stored with each element is precisely its rank within its block. In another array, we store the index of the $(in/2^c)$th element in the sorted order of the elements (i.e. the first element of the i-th block), for $1 \leq i \leq 2^c$.

Given an element, the membership proceeds as in the case of our modified FKS strategy (as mentioned in the last section). Once an element is found, the block to which it belongs (in the sorted order) can be found by doing a binary search (using c steps) on the first elements of each block stored in the separate array. The rank of an element within its block is stored with the element in the FKS dictionary. From these two information, we can obtain the rank of the element.

The space required, in addition to the $n \lg m/n + O(n + \lg\lg m)$ bits used to store the quotient values and the hash function information, is $n \lg n - cn + 2^c \lg n$

bits. Let the space occupied by the hash function and the auxiliary storage for the FKS dictionary be $d(n + \lg\lg m)$ bits. Choose c such that $cn > 2^c \lg n + dn$. Then the total space requirement will be $n \lg m + O(\lg\lg m) - \Theta(n)$ bits. If n is $\Omega(\lg\lg m)$, we can choose c such that $cn > 2^c \lg n + d(n + \lg\lg m)$ so that the total space will be $n \lg m - \Theta(n)$ bits.

Note that we actually don't store the elements in the array locations, but store only the quotient value of the element in the location to which it hashes to. So we describe below, how given a location, we can actually find the element of the set, whose quotient is stored in that location, in constant time.

For this purpose, we store the values of k^{-1} and κ^{-1} along with other parameters, which require $O(\lg\lg m + \lg n)$ bits of extra space. Now, given a location l, let q be the quotient value stored in that location and let x be the actual element of the given set that hashes to that location.

Table D (given in the last section) can be used to find the location $l^\star$ in the virtual array $A^\star$ in which the element should have been stored, using a select operation on the bit representation of D. Now, using the table T_0 (i.e. the compressed form of table C), we can find the bucket into which the element has hashed to, which is nothing but the value of $(\kappa.z \ mod \ r) \ mod \ n$. Also $q \ mod \ r/n$ gives us the value $(\kappa.z \ mod \ r) \ div \ n$. From these two values, we can find the value of $\kappa.z \ mod \ r$ from which, using κ^{-1} we can find z. Note that z is nothing but $(k.x \ mod \ p) \ mod \ n^2$. Now, $(q \ mod \ r/n) \ mod \ p/r$ gives the value of $(k.x \ mod \ p) \ div \ n^2$. Using these two values, we can find $(k.x \ mod \ p)$ from which the value of $x \ mod \ p$ can be found using the value of k^{-1}. Again, $(q \ mod \ r/n) \ div \ p/r$ gives the value of $x \ div \ p$. Using these two values, we can find the value x.

Thus we have,

Theorem 2. *There exists a static dictionary for a subset S of size n of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg\lg m) - \Theta(cn)$ bits of space and answers membership queries in constant time and rank queries in $O(c)$ time where $1 \leq c \leq (1 - \epsilon) \lg n$ for any positive constant $\epsilon < 1$.*

Corollary 1. *There exists a static dictionary for a subset S of size n of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg\lg m) - \Theta(n)$ bits of space and answers membership and rank queries in $O(1)$ time.*

Corollary 2. *There exists a static dictionary for a subset S of size n of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m - \Theta(n)$ bits of space and answers membership and rank queries in $O(1)$ time when $n = \Omega(\lg\lg m)$.*

Note the time-space tradeoff in the main theorem above. In particular, if we are willing to support the rank operation in $O(\lg\lg n)$ time, then space complexity comes down to $n \lg m + O(\lg\lg m) - \Theta(n \lg\lg n)$ bits.

3.1 Static dictionary supporting select

Suppose we want to support only membership and select operations efficiently. To support select, besides the modified FKS dictionary to support membership, we can store in an array, the pointer to the ith smallest element of the set, for $1 \leq i \leq n$. This requires an additional $n \lg n$ bits of space.

To further reduce space, we again store only the last $\lg n - c$ bits (i.e. the position of the jth element within a block of size $n/2^c$) for some parameter c (to be determined) and in a separate array store the index of the $(ni/2^c)$th element in the sorted order of the elements, for $1 \leq i \leq 2^c$. Given a j, to find the jth element, we do the following. Find the last $\lg n - c$ bits of the position of the jth element from the first array. Now for each choice of the first c bits, find the element stored in the location given by the $\lg n$ bits. If that element lies between the elements ranked $n(j-1)/2^c$ and $nj/2^c$ (which can be found using the pointers stored in the second array), output that element as the jth element. Clearly, there will be a unique choice of the first c bits, which gives the location of the j-th smallest element.

As in the last section, c can be chosen in such a way that $cn > 2^c \lg n + dn$.

Thus we get

Theorem 3. *There exists a static dictionary for a subset S of size n of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg \lg m) - \Theta(cn)$ bits of space and answers membership queries in constant time and select queries in $O(2^c)$ time for any parameter $c < \lg n$.*

Corollary 3. *There exists a static dictionary for a subset S of size n of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg \lg m) - \Theta(n)$ bits of space and answers membership and select queries in $O(1)$ time. When $n = \Omega(\lg \lg m)$, the space used is simply $n \lg m - \Theta(n)$ bits.*

3.2 Static dictionary supporting rank and select

When n is a constant, we can support membership, rank and select using $n \lg m$ bits by storing the elements in a sorted array. Also when $n > \frac{m}{\lg m} + \frac{m \lg \lg m}{(\lg m)^2}$, we can store a bit vector of the subset (m bits) and some auxiliary structures ($o(m)$ bits) to support membership, rank and select in constant time[5, 3].

Fiat et. al.[7] have given a structure to store multi key records where search can be performed under any key in constant time. By storing an element and its rank as a two key record, using this structure, one can support membership, rank and select queries in constant time. This structure takes $n \lg mn + O(\lg \lg m + \lg n)$ bits of space.

Another obvious way to support both rank and select operations is to take either of the previous two structures (supporting rank or select) given in the last subsections and augment it with an array (of $n \lg n$) bits to support the other operation also in constant time. Thus we can support membership, rank and select in constant time using a structure that takes $n \lg mn + O(\lg \lg m) - \Theta(n)$ bits of space.

One can also find the rank by performing a binary search in a structure that supports membership and select. Thus we can support membership and select in constant time and rank in $\lg n$ time using a structure that takes $n \lg m + O(\lg\lg m) - O(n)$ bits of space.

It would be interesting to know whether we can support all these operations in constant time using $n \lg m + O(\lg\lg m) + o(n)$ bits.

4 Representing m-ary cardinal trees

In this section, we look at the problem of representing an m-ary cardinal tree. In this tree, each node has m positions for an edge to a child, some of which can be empty. Benoit et. al.[3] have given an optimal representation of a cardinal tree that takes $n\lceil \lg m \rceil + 2n + o(n)$ bits and supports all navigational operations in constant time, except finding a child labeled i, which takes at most $O(\lg\lg m)$ time. This encoding has two parts. The first one uses the succinct encoding of ordinal trees [11, 3] which takes $2n + o(n)$ bits to store an ordinal tree of n nodes that supports all navigational operations (on ordinal trees) in constant time. In the second part, the $n\lceil \lg m \rceil$ bits of storage is used to store, for each node, $d\lceil \lg m \rceil$ bits to encode which children are present, where d is the number of children at that node.

In this structure, when the given subset is very sparse (namely when $n \leq \lg m$), they store the elements in sorted order, so that a search for an element or finding the rank of it takes $O(\lg n)$ time. When $n > \lg m$ the universe is split into equal sized buckets and the values that fall into each bucket are stored using perfect hash functions. By choosing the number of buckets appropriately, one can make the space occupied by this structure to be at most $n \lg m$ bits.

We observe that by using a static dictionary that supports rank and membership in constant time that requires at most $n \lg m$ bits of space, we can construct a m-ary cardinal tree structure that supports all navigational operations in constant time.

If the number of children of a node is less than $\lg\lg m$, we store them in a sorted array. Membership and rank queries in this array can be answered in $O(\lg\lg\lg m)$ time. When n is at least $\lg\lg m$, we store it using the structure of Corollary 4.

Thus we have

Theorem 4. *There exists an $n\lceil \lg m \rceil + 2n + o(n)$ bit representation of m-ary cardinal trees on n nodes that supports the operations of finding the parent of a node or the size of the subtree rooted at any node in constant time and supports finding the child with label j in $O(\lg\lg\lg m)$ time.*

When n is $\omega(m^{\lg\lg m + 1 + o(1)})$, we propose an alternate structure. We follow the above encoding except for vertices whose degree is at most $\lg\lg m$. We construct a table in which each entry represents a set of size at most $\lg\lg m$ which is stored as an m bit vector. Along each entry, we also store an auxiliary structure which takes $o(m)$ bits to support rank operation on the m bit characteristic vector in constant time. We will have a two level ordering of the table. In the first level, we order the sets based on their cardinalities. In the second level, we order sets with the same cardinality lexicographically (in the bit vector representation). Now in the cardinal representation, when a node has degree at most $\lg\lg m$ instead of storing them in sorted order, we simply keep the position of the set in the second level of the table (since we can compute the cardinality of the set, which is the same as the degree of the node, from the ordinal information, we obtain the position in the first level of the table).

The space occupied by the table is $O(m^{\lg\lg m})(m + o(m))$ which is $o(n)$. The space used by the index at each small degree node is $\lg\binom{m}{d}$ which is at most $d\lg m$ bits, where d is the degree of the node. Now for these nodes, to search for a child or to find its rank, we first find the subtree size from the ordinal information of the node and using the index stored with the node in the cardinal part, find the bitmap of the subset, which can be used to search for the element or find its rank (using the $o(m)$ auxiliary structure) in constant time.

Thus we have,

Theorem 5. *There exists an $n\lceil\lg m\rceil + 2n + o(n)$ bit representation of m-ary cardinal trees on n nodes that supports all the navigational operations in constant time when n is $\Omega(m^{\lg\lg m+1})$.*

5 Conclusions and open problems

We have given representations of static dictionaries that support rank (or select) and membership queries in constant time using $n\lg m + O(\lg\lg m)$ bits of space. This gives us a structure that supports rank, select and membership queries in constant time using $n\lg mn + O(\lg\lg m)$ bits of space. We also gave a representation of a m-ary cardinal tree that supports all navigational operations in constant time except finding a child with label j which takes at most $O(\lg\lg\lg m)$ time using $\lceil n\lg m\rceil + 2n + o(n)$ bits of space.

Some open problems that arise/remain are:

- Find the space optimal structures for supporting membership, rank (for elements in the set) and/or select queries in constant time.
- Find a representation of m-ary cardinal trees that takes $\lceil n\lg m\rceil + 2n + o(n)$ bits of space and supports all navigational operations in constant time for all values of n.

References

1. M. Ajtai, "A lower bound for finding predecessors in Yao's cell probe model", *Combinatorica* **8** (1988) 235-247.
2. D. Benoit, " Compact Tree Representations", Master's Thesis, Department of Computer Science, University of Waterloo, Canada (1998).
3. D. Benoit, E. D. Demaine, J. I. Munro and V. Raman "Representing Trees of Higher Degree", To appear in *The Proceedings of the Workshop on Algorithms and Data-structures* (1999).

4. A. Brodnik and J. I. Munro, "Membership in constant time and almost minimum space", to appear in *SIAM Journal on Computing.*
5. D. R. Clark, "Compact Pat Trees", Ph.D. Thesis, University of Waterloo, 1996.
6. D. R. Clark and J. I. Munro, "Efficient Suffix Trees on Secondary Storage", *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms* (1996) 383-391.
7. A. Fiat, M. Noar, J. P. Schmidt and A. Siegel, "Non-oblivious hashing", *Journal of the Association for Computing Machinery,* **39**(4) (1992) 764-782.
8. M. L. Fredman, J. Komlós and E. Szemerédi, "Storing a sparse table with $O(1)$ access time", *Journal of the Association for Computing Machinery,* **31** (1984) 538-544.
9. G. Jacobson, "Space-efficient Static Trees and Graphs", *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1989) 549-554.
10. J. I. Munro, "Tables", *Proceedings of the 16th FST & TCS conference,* Lecture Notes in Computer Science **1180** (1996) 37-42.
11. J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs", *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1997) 118-126.
12. Rasmus Pagh, "Low redundancy in dictionaries with O(1) worst case lookup time", to appear in *Proceedings of the International Colloquium on Automata, Languages and Programming* (1999).
13. J. P. Schmidt and A. Siegel, "The spatial complexity of oblivious k-probe hash functions", *SIAM Journal on Computing* **19**(5) (1990) 775-786.
14. D. E. Willard, "Log-Logarithmic worst case range queries are possible in space $\Theta(n)$", *Information Processing Letters* **17** (1983) 81-89.