TELKOMNIKA Indonesian Journal of Electrical Engineering
Vol. 12, No. 3, March 2014, pp. 2406 ∼ 2413
DOI: http://dx.doi.org/10.11591/telkomnika.v12.i3.3629

A Practical Algorithm and Data Structures for Range Selection Queries

Daxin Zhu and Xiaodong Wang*[1]
Quanzhou Normal University
362000 Quanzhou, Fujian, China
[1]*Corresponding author, e-mail: wangxiaodong@qztc.edu.cn

Abstract

In this work, we consider the problem of building an efficient data structure for range selection queries. While there are several theoretical solutions to the problem, only a few have been tried out, and there is little idea on how the others would perform. The computation model used in this paper is the RAM model with word-size $\Theta(\log n)$. Our data structure is a practical linear space data structure that supports range selection queries in $O(\log n)$ time with $O(n \log n)$ preprocessing time.

Keywords: *Range selection queries, data structures, preprocessing time*

1. Introduction

A generalized median finding problem is studied in this paper. The classical problem of median finding is to find the median item, the item of rank $n/2$ in an unsorted array of size n. In the comparison model, where items in the array can be compared only using comparisons, the complexities of the algorithms are defined by the number of comparisons performed by the algorithms. It has been known for a long time that this problem can be solved using $O(n)$ comparisons in the worst case. Many researches showed that the number of comparisons needed for solving the median finding algorithm is between $(2 + \varepsilon)n$ and $2.95n$ in the worst case[13]. Closing this gap for a deterministic algorithm is an open problem. However, the median of an unsorted array of size n can be found in $1.5n + o(n)$ comparisons by using a randomized algorithm.

In this work, we study the following generalization of the median problem. We consider the problem of building an efficient data structure for range selection queries. The problem is to preprocess an input array A of n integers, such that given a query (i, j, k), we can report the kth smallest integer in the subarray $A[i], A[i + 1], \cdots, A[j]$ efficiently. In the rest of the paper, the subarray $A[i], A[i + 1], \cdots, A[j]$ is denoted as $A[i, j]$. A special case of the problem is known as range median query, which arises when k is fixed to $\lfloor (j - i + 1)/2 \rfloor$. The prefix selection query is another special case of the problem, which arises when i is fixed to 0. These problems have many important applications in statistical analysis, and have been studied extensively in the last few years, see e.g. [6, 10, 12].

If we want to support fast range median queries, then we can use a data structure supporting range median queries in constant time using $O(n^2 \log\log n/ \log n)$ words of space presented in [2]. The space bound of this data structure was later improved to $O(n^2 \log^k n/ \log n)$, for any constant k [3]. A data structure that supports range median queries in expected constant time with expected $O(n^{3/2})$ space, under the assumption that all inputs are equally likely can also be found in [1]. A linear space data structure capable of answering range median queries in $O(n^\epsilon)$ time for any constant $\epsilon > 0$ was also available. These results can be improved to linear space and $O(\log n)$ query time by data structures, which also supports range selection queries in the same bound. Finally, a linear space data structure with $O(\log n/ \log\log n)$ query time for range selection was developed [11], where it was also shown how to augment the range selection data structure to support 2d dominance counting queries in the same query bound.

Received July 15, 2013; Revised September 23, 2013; Accepted October 12, 2013

The wavelet tree is a data structure to represent a sequence and answer some queries on it. This data structure was invented in 2003 by Grossi, Gupta, and Vitter [8]. The wavelet tree is a versatile data structure. It serves a number of purposes such as string processing and geometry. The wavelet tree can be regarded as a device that represents a sequence, a reordering, or a grid of points. Many competitive solutions to a number of problems are based on wavelet trees such as basic and weighted point grids, sets of rectangles, strings, permutations, binary relations, graphs, inverted indexes, document retrieval indexes, full-text indexes, XML indexes, and general numeric sequences. One of the efficient application of the wavelet tree was in solving the range quantile query problem[5, 6, 7]. The range quantile query problem can be solved by using the wavelet tree structure in $O(\log\sigma)$ time and just $o(n)$ extra bits. This is close to $O(\log n/\log\log n)$.

Previously, the best linear space data structure supported range selection queries in $O(\log n)$ time. In the dynamic case the only known data structure uses $O(n\log n)$ space and supports updates and queries in $O(n\log^2 n)$ time. For dominance queries, linear space data structures supporting queries in $O(\log n/\log\log n)$ time is known. In the dynamic case [4] describes an $O(n)$ space data structure that supports dominance queries in $O((\log n/\log\log n)^2)$ time and updates in $O(\log^{9/2} n/(\log\log n)^2)$ time.

We present a practical study on data structures for sequences supporting range selection queries. While there are several theoretical solutions to the problem, only a few have been tried out, and there is little idea on how the others would perform [1, 8, 9, 11]. The computation model used in this paper is the RAM model with word-size $\Theta(\log n)$. The data structure presented in this paper has the same basic approach as in [4]. We design a practical linear space data structure that supports range selection queries in $O(\log n)$ time.

The organization of the paper is as follows. In the following 3 sections we describe our general data structure design paradigm.

In section 2 we give an extremely simple data structure for answering range selection queries with $O(n\log n)$ time and space. Then we reduce total space costs since only the array stored in the left subtree is used. By using bit-vectors, the query time of the algorithm can also be reduced further. Finally we obtained a data structure for solving range selection problem with preprocessing time $O(n\log n)$ using $O(n)$ space and query time $O(\log n)$. This improves the space consumption compared to [4] by a factor $O(\log^2 n/\log\log n)$.

In section 3 we give a computational study of the presented data structure which demonstrates that the achieved results are not only of theoretical interest, but also that the techniques developed may actually lead to a practical data structure for general range selection algorithms.

Some concluding remarks are in section 4.

2. The Design of Data Structure

2.1. A Simple Data Structure

Let $A = (A[0], A[1], \cdots, A[n-1])$ be the input array. Our data structure is a complete binary tree. We sort the array A and build a corresponding complete binary tree T that stores the n elements in the leaves in sorted order.

For a node v in the tree T, the subtree rooted at node v is denoted as T_v, and the number of leaves in T_v is denoted as $|T_v|$. In each node v of T, the $|T_v|$ elements in the leaves of subtree rooted at node v are stored in an array A_v of size $|T_v|$ sorted by their index in A.

The construction of the basic data structure can be described as follows.

```
Algorithm 2..1: PREPROCESS($A, n$)

  comment: Build range select data structure

  $y \leftarrow$ INDEXSORT($A$)
  $v \leftarrow$ BUILD($y, 0, n-1$)
  return ($v$)
```

In the above algorithm, we first sort the input array A into an index array y by the algorithm INDEXSORT(A) such that the sequence $A[y[0]], A[y[1]], \cdots, A[y[n-1]]$ is exactly the n elements $A[0], A[1], \cdots, A[n-1]$ of the input array in sorted order. Then the actual construction of the tree T is performed by the recursive algorithm BUILD$(y, first, last)$ as follows.

```
Algorithm 2..2: BUILD($y, first, last$)

  comment: Construct the tree recursively
  $mid \leftarrow (first + last)/2$
  if $last > first$
    then { $left \leftarrow$ BUILD($y, first, mid$)
         { $right \leftarrow$ BUILD($y, mid + 1, last$)
  $v \leftarrow$ MAKE-TREE($left, right$)
  $A_v \leftarrow$ MERGE($left.A_v, right.A_v$)
  return ($v$)
```

In the algorithm BUILD$(y, first, last)$, the parameters $first$ and $last$ indicate the begin and end positions of array A_v in array y, which is the index array computed by INDEXSORT(A). The tree T is built recursively in a bottom up manner. The sub-algorithm MERGE$(left.A_v, right.A_v)$ merges the two arrays in the subtrees of v into an array A_v of size $|T_v| = last - first + 1$ sorted by their index in A. This merge procedure is exactly the same as the merge procedure of merge sort algorithm.

Since our algorithm for constructing the basic data structure is essentially a sorting process, we can readily conclude that our algorithm uses $O(n \log n)$ time and $O(n \log n)$ words of space in the worst case.

If we visit the nodes of the complete binary tree T by an in-order traversal we can see that, at any node v of T, the elements in the array A_v are subdivided into two parts of (almost) equal size and stored in the left child and the right child nodes of v. The two parts in the subtrees are recursively subdivided further.

To answer a range selection query, we first visit the root of T and determine in which of its two subtrees the element of the desired rank lies. Once this is known, the search continues recursively in the appropriate subtree until a trivial problem of constant size is encountered.

The algorithm for answering the range selection queries (i, j, k) is described as follows.

```
Algorithm 2..3: QUERY($v, first, last, i, j, k$)

  comment: Answer range selection queries $(i, j, k)$
  $mid \leftarrow (first + last)/2$
  if $last = first$
    then return ($A[y[first]]$)
  $l \leftarrow$ RANK($v.left, i - 1$)
  $r \leftarrow$ RANK($v.left, j$)
  $s \leftarrow r - l$
  if $k \leq s$
    then return (QUERY($v.left, first, mid, l, r - 1, k$))
    else return (QUERY($v.right, mid + 1, last, i - l, j - r, k - s$))
```

In the above algorithm QUERY$(v, first, last, i, j, k)$, we want to find the element of rank k in the subarray $A[i, j]$ from node v of T, where array A_v begins at position $first$ and ends at position $last$ of array y.

The number l is the rank of $i-1$ in the array A_v of the left subtree $v.left$. It means that there are l elements in the left subtree $v.left$ are to the left of i in the subarray $A[i,j]$. The number r is the rank of j in the array A_v of the left subtree $v.left$. It means that there are r elements in the left subtree $v.left$ are to the left of and up to j in the subarray $A[i,j]$. Thus the number s is the number of elements in the array A_v of the left subtree $v.left$ contained in in the subarray $A[i,j]$.

If $k \leq s$ then the element of rank k in $A[i,j]$ is the element of rank k in the subarray $A_v[l, r-1]$ of the left subtree $v.left$. Otherwise, the element of rank k is the element of rank $k-s$ in the subarray $A_v[i-l, j-r]$ of the right subtree $v.right$.

Thus the algorithm reduces the problem of finding an element of a given rank in the subarray $A[i,j]$ to the same problem, but on a smaller array. This reduction is applied recursively by the algorithm.

The number l and r can be found in $O(\log((last-first+1)/2))$ time by a binary search. The query descends $\log n$ levels of recursion, so the algorithm $Query$ would get a total execution time of up to $\sum_{i=1}^{\log n} O(\log n/2^i) = O(\log^2 n)$.

2.2. An Improved Data Structure

In the algorithm Query$(v, first, last, i, j, k)$ we can see that to find out in which subtrees the element of rank k lies, only the array stored in the left subtree is used. Therefor, we can lift it to the node v and thus save half of the total space costs. Furthermore, we note that to compute the numbers l and r, the information available in the arrays stored at the interior nodes of our data structure can be reduced further. We can use a bit-vector to store the information we need, where a 1-bit indicates whether an element of the original array is in A_v. Since we have n positions one very level, a total of $O(n\log n)$ bits are used in our data structure.

With these bit-vectors, the execution time of the algorithm $Query$ can also be reduced.

In order to compute the number l and r in the query algorithm efficiently, we can store a table with ranks for indices that are a multiple of the size of machine word w. General ranks are then the sum of the next smaller table entry and the number of 1-bits in the bit-vector between this rounded position and the query position. In this way, the number l and r can be computed in $O(1)$ time. Therefore the execution time of the algorithm $Query$ can be reduced to $O(\log n)$.

Summing up, we have obtained a data structure for solving range selection problem with preprocessing time $O(n\log n)$ using $O(n)$ space and query time $O(\log n)$. This improves the space consumption compared to [16] by a factor $O(\log^2 n/\log\log n)$.

3. The Implementation and Experiments

In this section we will describe the implementation of the data structure presented in last section. Based on the discussion above, we can design an node class for the nodes of the complete binary tree of our new data structure.

```
class node –
public:
  node()–˝ // constructor
  void init(vector¡int¿& y,int first ,int last);
  int rank(int k);
private:
  bitvector low; // elements in the left subtree
  vector¡int¿ t; // a table of ranks
˝;
```

In the node class we store a bitvector low to indicate the elements in the left subtree of current node sorted by their index in A.

The vector t is a table with ranks for indices that are a multiple of the size of machine word w. With the vector t the rank of an index in the bitvector low can be computed in $O(1)$ time as follows.

```
int node::rank(int k)
–
   if(k¿low.size()) return 0;
   int a=0,b=k/w;
   for(int i=w*b;i¡=k;i++)a+=low[i];
   if(b¿0)a+=t[b-1];
   return a;
˝
```

the complete binary tree of our new data structure is built recursively in a bottom up manner. For the current node, the two arrays of its subtrees are merged into one array and then the bitvector low is formed. According to the information of low, the vector t can then be constructed readily. The merge procedure is exactly the same as the merge procedure of merge sort algorithm. The following algorithm $init$ performs these tasks.

```
void node::init(vector¡int¿& y,int first ,int last)
–
   deque¡int¿ dq; // a dequeue for merging two sorted arrays
   int k=first , j=(last-first)/2,
   mid=(first+last)/2, a=0, b=(last-first+1)/w;
   low.resize(last-first+1);
   t.resize(b);
   if(first==last)–low[0]=true; return;˝
   // push˙back y[first,mid] from left to right
   for(int i=first;i¡=mid;i++)dq.push˙back(y[i]);
   // push˙back y[mid+1,last] from right to left
   for(int i=last;i¿mid;i--)dq.push˙back(y[i]);
   while(!dq.empty())
      if(dq.front()¿dq.back())–
         y[k++]=dq.back();dq.pop˙back(); // pop back
      ˝

      else–
         if(j¿=0)–low[k-first]=true;j--;˝
         y[k++]=dq.front();dq.pop˙front(); // pop front
      ˝

   j=k=0;
   while(b¿0)–
      // preprocessing t
      for(int i=0;i¡w;i++)j+=low[k+i];
      t[a++]=j;k+=w;b--;
   ˝
˝
```

In the algorithm $init$, the parameters $first$ and $last$ indicate the begin and end positions of array A_v in array y, which is the index array computed by the $Indexsort$ algorithm. The size of bitvector low must be $last - first + 1$ bits.

A dequeue is used for merging two sorted arrays $y[first, mid]$ and $y[mid + 1, last]$. In the merge process, if the next element comes from the first array $y[first, mid]$, then the corresponding bit of bitvector low is marked true. When the two arrays are sorted, the bitvector low is built. The table t can then be constructed easily from the bitvector low.

With the class $node$, we can design a new class $rmedian$ for our new data structure for general range selection query as follows. In the class $rmedian$, array $data$ is used to store the input sequence. The arrays y and z are used to store the sorted index arrays of the input sequence.

```
class rmedian–
public:
  rmedian()–˝ // constructor
  rmedian(vector¡int¿& dt)–init(dt);˝ // constructor
  int median(int left,int right);
private:
  vector¡int¿ data,y,z;
  vector¡node¿ bt; // a complete binary tree
  void init(vector¡int¿& dt);
  void build(int ind,int first,int last);
  int query(int ind,int first,int last,
            int left,int right,int rank);
˝;
```

The main part of the class is a vector bt of element type $node$. This vector is used to store the complete binary tree T that the n elements of the input sequence are stored in its leaves in sorted order. The vector bt is in fact an array indexed binary tree. The root of the tree is $bt[0]$. For a given index i of a node, the left and right child node of $bt[i]$ are $bt[2*i+1]$ and $bt[2*i+2]$ respectively. The parent node of of $bt[i]$ is $bt[(i-1)/2]$.

The vector bt can be built recursively in a bottom up manner as follows.

```
void rmedian::build(int ind,int first,int last)
–
    int mid=(first+last)/2;
    if(last¿first)–
        build(2*ind+1,first,mid);
        build(2*ind+2,mid+1,last);
    ˝
    bt[ind].init(y,first,last);
˝
```

Once an instance of $rmedian$ is built, we can then answer any range selection query in $O(\log n)$ time as follows.

For any range selection query QUERY$(ind, first, last, left, right, rank)$, we want to find the element of rank $rank$ in the subarray $A[left, right]$ from node $v = bt[ind]$ of bt, where array A_v begins at position $first$ and ends at position $last$ of array z.

We first find the number l and r, which are the ranks of $left-1$ and $right$ in the array A_v of the left subtree of node v respectively.

Thus the number $length = r-l$ is computed, which is the number of elements in the array A_v of the left subtree of v contained in in the subarray $A[left, right]$.

If $rank \leq length$ then the element of rank $rank$ in $A[left, right]$ is the element of rank $rank$ in the subarray $A_v[l, r-1]$ of the left subtree of v. Otherwise, the element of rank $rank$ is the element of rank $rank - length$ in the subarray $A_v[left-l, right-r]$ of the right subtree of v.

The algorithm reduces the problem of finding an element of a given rank in the subarray $A[left, right]$ to the same problem, but on a smaller array. This reduction is applied recursively by the algorithm as follows.

```
int rmedian::query(int ind,int first,int last,
                   int left,int right,int rank)
–
    int mid=(first+last)/2;
    node m=bt[ind];
```

Table 1. The bitvectors of the complete binary tree bt

0111010011101000							
10001101	00111100						
1100	0011	1010	0110				
01	10	01	01	01	01	01	10

```
    if(first==last)return data[z[first]];
    int l=m.rank(left-1), r=m.rank(right);
    int length=r-l;
    if(rank¡=length)
      return query(2*ind+1,first,mid,l,r-1,rank);
    else
      return query(2*ind+2,mid+1,
        last,left-l,right-r,rank-length);
˝
```

We can explain the implementation of our data structure by an example with the input array $A = [14, 1, 7, 6, 13, 5, 9, 11, 0, 2, 4, 8, 3, 10, 12, 15]$ of 16 elements as follows. In the first step, the input array A is sorted into an index array

$y = [8, 1, 9, 12, 10, 5, 3, 2, 11, 6, 13, 7, 14, 4, 0, 15]$ by the algorithm INDEXSORT(A) such that the sequence $A[y[0]], A[y[1]], \cdots, A[y[15]]$ is exactly the 16 elements of the input array in sorted order.

Then the complete binary tree bt that stores the 16 elements in the leaves in sorted order is built according to the index array y. The most important data stored in the nodes of bt is the corresponding bitvector low. The following Table 1 shows the bitvector low in the nodes of bt by a level-order traversal.

With this basic data structure we can answer any range selection query in $O(\log n)$ time. For example, if we want to find the median of the sub-array $A[4, 10] = [13, 5, 9, 11, 0, 2, 4]$, then a function call of $query(0, 0, 15, 4, 10, 4)$ will give the result.

The algorithm visit the root of bt first.

According to the bitvector $low = [0111010011101000]$ of the root, the elements of the left subtree of the root sorted by their indices in A are

$$A[1], A[2], A[3], A[5], A[8], A[9], A[10], A[12].$$

The rank of index 3 and index 10 in this sequence can be computed as $l = 3$ and $r = 7$ and thus $length = 4$. The median is contained in the left subtree. The function call of $query(1, 0, 7, 3, 6, 4)$ is then applied.

In the node $bt[1]$, according to its bitvector $low = [10001101]$, l ,r and $length$ can be computed as $l = 1$, $r = 3$ and $length = 2$ and thus the recursive call $query(4, 4, 7, 2, 3, 2)$ is applied.

Similarly, the next recursive call is $query(9, 4, 5, 0, 1, 2)$.

Finally, the function call $query(20, 5, 5, 0, 0, 1)$ gives the query answer $A[5] = 5$.

We implemented our data structure in C++ and tested them on a personal computer with Pentium(R) Dual Core CPU 2.10 GHz and 2.0 Gb RAM, using the Microsoft Visual C++ version 8.0 compilers. The word size of the processor is $w = 32$.

The experiment results show that our data structure is very practical for solving the range selection query problem.

We also performed some limited experiments on the relative performance of our data structure. The new data structure has similar or better speed than existing data structures but uses less space in the worst case.

4. Conclusion

We have presented new data structure for solving the range selection query problem. While there are several theoretical solutions to the problem, only a few have been tried out, and there is little idea on how the others would perform. The computation model used in this paper is the RAM model with word-size $\Theta(\log n)$. Our data structure is a practical linear space data structure that supports range selection queries in $O(\log n)$ time with $O(n \log n)$ preprocessing time.

The computational experiments in Section 3 demonstrate that the achieved results are not only of theoretical interest, but also that the techniques developed may actually lead to considerably faster algorithms.

Acknowledgement

The authors acknowledge the financial support of Natural Science Foundation of Fujian under Grant No.2013J01247, and the Haixi Project of Fujian under Grant No.A099.

References

[1] M. J. Atallah and H. Yuan, Data structures for range minimum queries in multidimensional arrays, In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 150-160, 2010.

[2] G. S. Brodal and A. G. Jorgensen, Data structures for range median queries, In *Proceedings of the 20th International Symposium on Algorithms and Computation*, 822-831, 2009.

[3] T. M. Chan, Persistent predecessor search and orthogonal point location in the word RAM, In *Proceedings of the 22nd ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 1131-1145, 2011.

[4] G. S. Brodal, B. Gfeller, A. G. Jargensen, P. Sanders, Towards optimal range medians, *Theoretical Computer Science*, 412(1):2588-2601, 2011.

[5] T. Gagie, J. Karkkainen, G. Navarro, S. J. Puglisi, Colored range queries and document retrieval, *Theoretical Computer Science*, 483(3):36-50, 2013.

[6] T. Gagie, G. Navarro, S.J. Puglisi, New algorithms on wavelet trees and applications to information retrieval. *Theoretical Computer Science*, 426-427:25-41, 2012.

[7] T. Gagie, S.J. Puglisi, A. Turpin, Range Quantile Queries: Another Virtue of Wavelet Trees. , *Proc. 16th International Symposium on String Processing and Information Retrieval, SPIRE 2009*, LNCS 5721:1-6, 2009.

[8] R. Grossi, A. Gupta, J.S. Vitter, High-order entropy-compressed text indexes. *Proceedings of the 14th Symposium on Discrete Algorithms*, 841C850, 2003.

[9] A.G. Jorgensen, K.D. Larsen, Range selection and median: Tight cell probe lower bounds and adaptive data structures, *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 805-813, 2011.

[10] D. Krizanc, P. Morin, and M. H. M. Smid, Range mode and range median queries on lists and trees. *Nordic Journal of Computing*, 12(1):1-17, 2005.

[11] Kasper Green Larsen, The cell probe complexity of dynamic range counting, In *Proceedings 44th ACM Symposium on Theory of Computing (STOC)*, 2012.

[12] H. Petersen and S. Grabowski, Range mode and range median queries in constant time and sub-quadratic space, *Information Processing Letters*, 109(4):225-228, 2008.

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