CENTRO *UNIVERSITÁRIO* DE BARRA MANSA
ACADEMIC PRO-RECTORY

COMPUTER ENGINEERING COURSE

QUICKSORT AND BINARY SEARCH ALGORITHMS

By:

Leniel Braz de Oliveira Macaferi
Wellington Magalhães Leite

Barra Mansa
June, 2005

CENTRO *UNIVERSITÁRIO* DE BARRA MANSA
ACADEMIC PRO-RECTORY

COMPUTER ENGINEERING COURSE

QUICKSORT AND BINARY SEARCH ALGORITHMS

By:

Leniel Braz de Oliveira Macaferi
Wellington Magalhães Leite

Paper presented to the Computer Engineering course at Centro Universitário de Barra Mansa, as a partial requisite to the obtention of the second grade for the Programming Languages discipline, under the supervision of prof. Marcus Vinicius Carvalho Guelpeli.

Barra Mansa
June, 2005

ABSTRACT

Sorting and searching algorithms are a core part of the computer science area. They are used throughout the programming work when you need to sort a set of data and when you need to search for a specific record (key) present in such set of data.

Quicksort is one of the fastest (quick) sorting algorithms and is most used in huge sets of data. It performs really well in such situations.

Binary search tree is one of the fastest searching algorithms and is applied in a sorted set of data. It reduces the search space by 2 in each iteration, hence its name (binary).

In this paper we present the intrinsic nature of each algorithm as well as a functional implementation of such algorithms in the C++ programming language.

Keywords: quicksort, binary search, sorting algorithms, searching algorithms, c++ programming language

LIST OF TABLES

CONTENTS

Page

1 INTRODUCTION

1.1 Objective

Our objective is to present and implement the Quicksort and Binary Search algorithms.

1.2 Definition

1.2.1 Sorting algorithms

In our day to day it's common to see the advantages of manipulating a set of data previously sorted, for example, when we need to find a phone number of a given person in the yellow pages. In this case we perceive that this simple searching task could be extremely exhaustive if the yellow pages weren't sorted by the subscribers' name in the alphabetic order.

We can observe that sorting the data can facilitate and augment the efficiency of the searching operations over a set of data. The sorting can be done in a crescent or decrescent order.

If we consider a set of input data in any fashion of n numbers $\{S_1, S_2, ..., S_n\}$, after classifying this input the output will be the sequence of n input numbers reordered. We'll verify that the output is such that: $S_1 \leq S_2, ..., \leq S_n$ (for the crescent sorting) or $S_1 \geq S_2, ..., \geq S_n$ (for the case of decrescent sorting).

Normally the input sequence is a vector with n records but there are other possible data structures as for example a linked list. In a practical form the records to be sorted are manipulated as a set of data denominated record instead of being manipulated in an isolated form. This way the set of records form a table. Each record has a key that is the value to be sorted and additional values that always follow the key. This means that when we start a sorting process, if there is the need of changing the position of a key, all the record's data will follow the key.

Aiming at the minimization of the computational task of moving the data during a sorting process, it's observed that in practice when the records are big, that is, have a great amount of data (beyond the key), the sorting process can be done using a vector or a list of pointers for the records.

In this paper we're going to approach one of the main sorting algorithms: sort by swapping, in our case the swapping and partitioning method (Quicksort).

1.2.2 Searching algorithms

In our day to day we do lots of searches in yellow pages, words in a dictionary, books in a library, clients in records, passenger tickets in a flight list, etc.

It appears that the location of a given record in a set of records is something simple but the development of efficient searching algorithms has been an arduous task for the computer scientists.

We’re going to approach one of the main searching methods applied in table (vectors), the binary search. We’ll discuss about its working mechanism and some of its efficiency aspects.

2 DEVELOPMENT

2.1 Quicksort algorithm (swapping and partitioning)

The quicksort method [1] was first proposed in 1962 by C. A. Hoare [2], a British mathematician. This method is considered the fastest in relation to methods as direct insertion, shellsort and bubble sort. This is the principal reason to its widespread use when it's necessary to sort a set of data. It adopts the principle that it is faster to classify two vectors with n / 2 records each one instead of n records, that is, it follows the lemma of divide and conquer (dividing a big problem into two small ones so that it's possible to have the complexity domain of the original problem).

2.1.1 Detailed steps

The most critical point in this method is the decision that must be taken of how to partition the vector that has the data to be sorted. To understand the importance of this decision, let's discuss a little bit about how the partitioning process is done.

Suppose that the vector is partitioned in three segments:

v[1], ..., v[i - 1]	v[i]	v[i + 1], ..., v[n]
| segment 1 |	| segment 2 |	| segment 3|

The selection of the record v[i] (called pivot record) that divides the vector in three segments can be done arbitrarily but normally it's chosen the record of the central part what makes the execution of the quicksort algorithm more efficient. After that, a record swapping is done so that the first segment has the records that are smaller than v[i] and the third segment has the records greater than v[i]. Later on we apply the same process over the first and third segments and so on until the original vector is completely sorted (classified).

A more detailed description of the quicksort algorithm can be written in the following way:

A. It's chosen arbitrarily a record of the vector (normally in the middle). After that the record is stored in an auxiliary variable aux;
B. Two pointers are created: i and j (i = 1 and j = n);
C. The vector is traversed starting on the left side until a value v[i] >= aux is found (incrementing the value of i);
D. The vector is traversed starting on the right side until a value v[i] <= aux is found

(decrementing the value of j);

E. It's effectuated the swapping of the records v[i] and v[j] (they're out of place). After that i = i + 1 and j = j – 1;
F. This process is repeated until the instant that i and j cross each other in some point of the vector;
G. In the moment that we get the two segments of the vector by means of the partitioning process, we do the sorting of each vector in a recursive way.

2.2 Binary search algorithm

Searching algorithms applied over tables, enhanced search in table and search with sentinel take into account that the search will be done in the sequential order and have the characteristic of being simple, however they can demand the inspection of all the records in the case of the searched record doesn't exist in the table.

The binary search [3] is a more efficient alternative if compared to the sequential search and demands that the records over which the search will be done be previously sorted.

Using the binary search we consider primarily that the record can be found in the middle of the table. If this record is greater than the record that we're after (for greater we mean "appears after"), we can guarantee that the record that we're after can't be found in the second part of the table.

We repeat the binary searching process for the first part of the table. If the record in the middle of the table is smaller than the record that we're after (for smaller we mean "appears before"), we can then guarantee that the record that we're after can't be found in the first half of the table. If the record in the middle of the table is equal the record we're after, the search ends.

In each step, the binary search reduces the number of records to be searched by n / 2 (so this is the meaning of its name - binary).

2.3 Studying the efficiency of the methods

2.3.1 The big O notation

One of the things to consider is the efficiency of the sorting and searching methods when we work with a considerable big number of records. If there is a file with only five records even the least efficient algorithm can solve the sorting problem. However as the

number of records augments the necessary effort starts to make a difference in a considerable way from algorithm to algorithm. To illustrate this possibility, let's consider the following example: in a file with 1.000.000.000 records (1 billion records) we find that a sequential search demands an average of 100.000.000 comparisons while the binary search does the sorting with a maximum of 61 comparisons. It's a significative difference!

It's possible to express an approximation to the relation between amount of work necessary and the number of records considered using a mathematical notation known as big O notation (we read it big Oh) [4].

2.3.2 Quicksort efficiency

It's worth to remember that each iteration of the quicksort algorithm divides the set of data in three parts. One part has the records that are smaller than the record considered as the pivot of this division; the other with the pivot record and the third part with the records greater than the pivot record. As already mentioned, the pivot normally is localized in the middle of the vector.

Assuming that this record is localized the nearest possible of the center of the vector, we can see that we divide the vector in two parts. This causes that the number of repetitions be logarithmic. The total effort of the sorting process is the result of the multiplications of the first repetition by the number of repetitions that are called in a recursive fashion or $n \cdot \log n$. We perceive that the computation effort in relation to the number of records n present in a set of data that will be sorted, grows in a logarithmic base and this is why the quicksort algorithm is one of the few sorting algorithm to play so well.

2.3.2.1 The best and the worst case

You can select the value of the pivot record by means of two forms. You can select it randomly or select it averaging a small set of data gotten from the vector. For the best sorting operation, you should select the value that is precisely in the middle of the range of values. Nevertheless, this isn't that easy in the majority data sets. In the worst case the value selected to be the pivot is in one of the extremities and even in this case, quicksort shows an outstanding efficiency.

2.3.2.2 Comparison with other sorting algorithms

n	Quick		Heap		Insert	
	Comp	Exch	Comp	Exch	Comp	Exch
100	712	148	2842	581	2595	899
200	1682	328	9736	9736	10307	3503
500	5102	919	53113	4042	62746	21083

Table 1 - Comparison between sorting methods

n	Quick			Heap			Insert		nlogn	n^2
	Comp	Exch	Norm	Comp	Exch	Norm	Comp	Exch	Norm	Norm
100	712	148	0,74	2842	581	2,91	2595	899	4,50	0,09
200	1682	328	0,71	9736	1366	2,97	10307	3503	7,61	0,09
500	5102	919	0,68	53113	4042	3,00	62746	21083	15,62	0,08

Divide by $n \log n$

Divide by n^2

Table 2 - Normalized values

Algorithm	Time	Notes
selection-sort	$O(n^2)$	• in-place • slow (good for small inputs)
insertion-sort	$O(n^2)$	• in-place • slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	• in-place, randomized • fastest (good for large inputs)
heap-sort	$O(n \log n)$	• in-place • fast (good for large inputs)
merge-sort	$O(n \log n)$	• sequential data access • fast (good for huge inputs)

Table 3 - Convergence time of some of the sorting algorithms

2.3.3 Binary search efficiency

With the binary search it's possible to reduce by half the number of records being considered in each iteration, that is, always that a comparison takes place. Suppose that totalRecords variable represents the total number of records upon which will be executed a search. The search starts over a file with totalRecords; after one iteration, the number of records will be totalRecords / 2, after that totalRecords / 4 and so forth. In the general case, the number of records to be considered after n iterations will be totalRecords / 2n.

3 APPLICATION

Here we present the C++ code of the quicksort and binary search algorithms.

3.1 Quicksort implementation

```
// QuickSort implementation
void QuickSort(char** szArray, int nLower, int nUpper)
{
  // Check for non-base case
  if(nLower < nUpper)
  {
    // Split and sort partitions
    int nSplit = Partition(szArray, nLower, nUpper);
    QuickSort(szArray, nLower, nSplit - 1);
    QuickSort(szArray, nSplit + 1, nUpper);
  }
}

// QuickSort partition implementation
int Partition (char** szArray, int nLower, int nUpper)
{
  // Pivot with first element
  int nLeft = nLower + 1;
  char* szPivot = szArray[nLower];
  int nRight = nUpper;

  // Partition array elements
  char* szSwap;
  while(nLeft <= nRight)
  {
    // Find item out of place
    while(nLeft <= nRight && strcmp (szArray[nLeft], szPivot) <= 0)
      nLeft = nLeft + 1;
    while (nLeft <= nRight && strcmp (szArray[nRight], szPivot) > 0)
      nRight = nRight - 1;

    // Swap values if necessary
    if(nLeft < nRight)
    {
      szSwap = szArray[nLeft];
      szArray[nLeft] = szArray[nRight];
      szArray[nRight] = szSwap;
      nLeft = nLeft + 1;
      nRight = nRight - 1;
    }
  }

  // Move pivot element
  szSwap = szArray[nLower];
  szArray[nLower] = szArray[nRight];
  szArray[nRight] = szSwap;
  return nRight;
}
```

3.2 Binary search implementation

```
int BinarySearch(char** szArray, char key[], int nLower, int nUpper)
{
  // Termination case
  if(nLower > nUpper)
    return 0;

  int middle = (nLower + nUpper) / 2;

  if(strcmp(szArray[middle], key) == 0)
    return middle;
  else
  {
    if(strcmp(szArray[middle], key) > 0)
      // Search left
      return BinarySearch(szArray, key, nLower, middle - 1);
    // Search right
    return BinarySearch(szArray, key, middle + 1, nUpper);
  }
}
```

4 CONCLUSION

Through the development of this work we had the opportunity of stretching our knowledge regarding the sorting and searching techniques. The subject in question has great value because of the fact that such algorithms make it possible to manipulate data (even a great amount of it) in an efficient way.

We’ve chosen to approach the quicksort sorting method and the binary search searching method because right in the moment that we made an initial analysis we found that they have a great acceptance in the software field. This acceptance is because of the efficiency of such methods.

With the adopted methodology it was possible to see the principle of each method, the algorithm and we got our hands dirty while implementing such algorithms.

It was really an excellent opportunity we had to master our programming skills and at the same time we learned about such mainstays of computer science.

5 REFERENCES

[1] Wikipedia. **Quicksort algorithm**. Available at <http://en.wikipedia.org/wiki/Quicksort>. Accessed May 20, 2008.

[2] —. **C. A . R. Hoare**. Available at <http://en.wikipedia.org/wiki/C._A._R._Hoare>. Acesso em 20 de May de 2008.

[3] —. **Binary search algorithm**. Available at <http://en.wikipedia.org/wiki/Binary_search>. Accessed May 20, 2008.

[4] —. **Big O notation**. Available at <http://en.wikipedia.org/wiki/Big_O_notation>. Accessed May 20, 2008.

[5] Moraes, Celso Roberto. **Estruturas de Dados e Algoritmos - Uma Abordagem Didática**. 5ª ed. São Paulo : Berkeley, 2001.

[6] Schildt, Herbert. **C, Completo e Total**. 3ª ed. São Paulo : Makron Books, 1997.

[7] sengpielaudio. **Time calculator**. Available at <http://www.sengpielaudio.com/calculator-millisecond.htm>. Accessed May 20, 2008.