Study on Sorting and Dictionaries

Sujeet Kumar & Ashish Gupta

Department of Information and technology Dronacharya College of Engineering, Gurgaon-122001, India
Email:sujeet.16939@ggnindia.dronacharya.info ; Email:ashish.16907@ggnindia.dronacharya.info

Abstract-

Arrays and linked lists are two basic data structures used to store information. We may wish to search, insert or delete records in a database based on a key value. This section examines the performance of these operations on arrays and linked lists. In this research paper we studied about the sorting and types of sorting. This is followed by techniques for implementing dictionaries, structures that allow efficient search, insert, and delete operations. We have also illustrated algorithms that sort data and implement dictionaries for very large files

Keywords-

insertion sort,quick sort, sort,binary search tree,skip list.

1. INTRODUCTION

Arrays and linked lists are two basic data structures used to store information. We may wish to search, insert or delete records in a database based on a key value. This section examines the performance of these operations on arrays and linked lists.

1.1 ARRAYS

Figure 1-1 shows an array, seven elements long, containing numeric values. To search the array sequentially, we may use the algorithm in Figure 1-2. The maximum number of comparisons is 7, and occurs when the key we are searching for is in A[6].

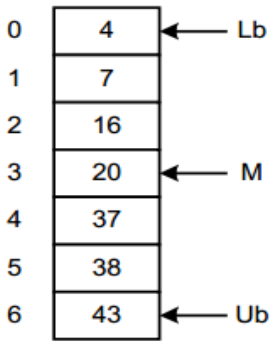

Figure 1-1: An Array

```
int function
SequentialSearch(Array A , int
Lb, int Ub, int Key );
    begin
    for i= Lbto Ub do
        if A [ i ]= Key then
            return i ;
        return –1;
    end;
```

Figure 1-2: Sequential Search

If the data is sorted, a binary search may be done (Figure 1-3). Variables Lb and Ub keep track of the lower bound and upper bound of the array. We begin from the middle element of the array. If the key we are searching for is less than the middle element, then it must reside in the top half of the array. Thus, we set Ub to (M –1). It restricts our next iteration through the loop to the top half of the array. Each iteration halves the size of the array to be searched. After the second iteration, there will be one item left to test. Therefore it takes only three iterations to find any number.

```
int function BinarySearch(Array
b, int Ub, int Key);
        begin
        do forever
            M = ( Lb + Ub )/2;
if ( Key< A[M])then
                Ub = M – 1;
            else if (Key >
A[M])then
                Lb = M + 1;
            else
                return M;
            if (Lb > Ub)then
                return –1;
        end;
```

Figure 1-3: Binary Search

In searching, we may wish to insert or delete data.we know, an array is not a good arrangement for these operations. For example, to insert the number 18 in Figure 1-1, we would need to shift A[3]…A[6] down by one slot. Then we could copy number 18 into A[3]. A similar problem arises when deleting numbers. To improve the efficiency of insert and delete operations, linked lists may be used.

1.2 LINKEDLISTS

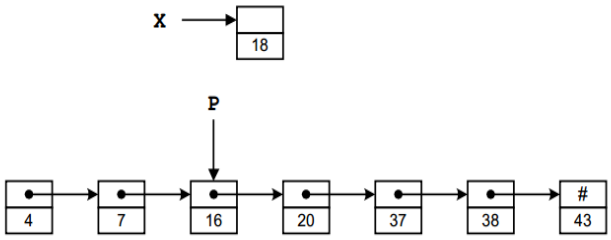

Figure 1-4: A Linked List

In Figure 1-4 we have the same values stored in a linked list. Assuming pointers Xand P, as shown in the figure, values may be inserted as follows:

X->Next = P->Next;

P->Next = X;

2.1 INSERTION SORT

It is the simplest methods to sort an array is an insertion sort. An example of an insertion sort occurs in everyday life while playing cards.

2.1.1 THEORY

Starting near the top of the array in Figure 2-1(a), we extract the 3. Then the above elements are shifted down until we find the correct place to insert the 3. This process repeats in Figure 2-1(b) with the next number. Finally, in Figure 2-1(c), we complete the sort by inserting 2 in the correct place.

Assuming there are nelements in the array, we must index through n – 1 entries. For each entry, we may need to examine and shift up to n –1 other entries. No extra memory is required. The insertion sort is also a stablesort. Stable sorts retain the original ordering of keys when identical keys are present in the input data.

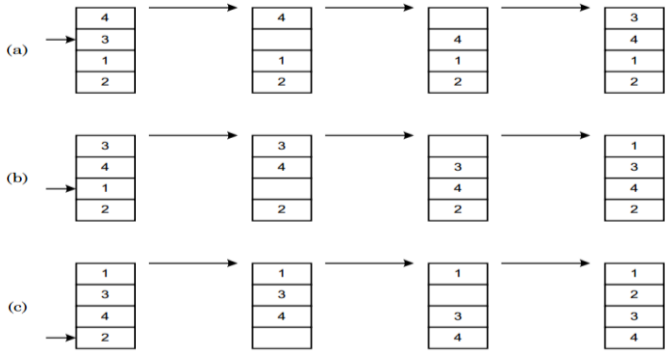

Figure 2-1: Insertion Sort

2.2 SHELLSORT

Shell sort, developed by Donald L. Shell, is a non-stable in-place sort. Shell sort improves the efficiency of insertion sort by quickly shifting values to their destination. Average sort time is O($n^{1.25}$), while worst-case time is O($n^{1.5}$).

2.2.1 THEORY

In Figure 2.2(a) we have an example of sorting by insertion. First we extract 1, shift 3 and down one slot, and then insert the 1, for a count of 2 shifts. In the next frame, two shifts are required before we can insert the 2. The process continues until the last frame, where a total of 2+ 2 + 1 = 5 shifts have been made.

Example of shell sort is illustrated in fig 2.2(b). We begin by doing an insertion sort using a spacingof two. In the first frame we examine numbers 3-1. Extracting 1, we shift 3 down one slot for a shift count of 1. Next we examine numbers 5-2. We extract 2, shift 5 down,and then insert 2. After sorting with a spacing of two, a final pass is made with a spacing of one. This is simply the traditional insertion sort. The total shift count using shell sort is 1+1+1 = 3.By using an initial spacing larger than one, we were able to quickly shift values to their proper destination.

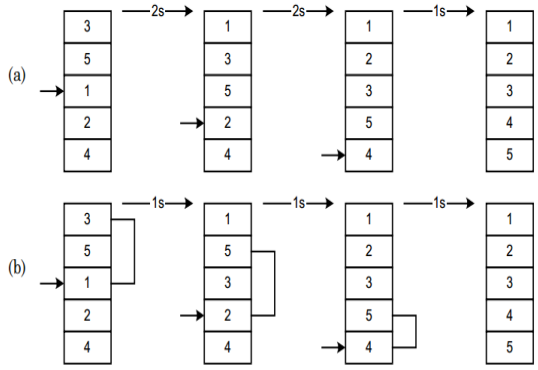

Figure 2-2: Shell Sort

2.3 QUICKSORT

The shell sort algorithm is significantly better than insertion sort, there is still room for improvement.Quicksort executes in O(n log n) on average, and $O(n^2)$ in the worst-case.Quicksort is a non-stable sort.

2.3.1 THEORY

The quicksort algorithm works by partitioning the array to be sorted, then recursively sorting each partition. In Partition(Figure 2-3), one of the array elements is selected as a pivot value. Values smaller than the pivot value are placed to the left of the pivot, while larger values are placed to the right.

```
int function Partition(Array A, int Lb, int Ub);
    begin
   select apivot fromA[Lb]…A[Ub];
   reorder A[Lb]…A[Ub] such that:
        all values to the left of thepivot are ≤pivot
       all values to the right of thepivot are ≥pivot
    return pivot position;
   end;
procedure QuickSort(Array A, int Lb, int Ub);
  begin
  if Lb< Ubthen
       M = Partition (A, Lb, Ub);
       QuickSort (A, Lb, M – 1);
QuickSort (A, M + 1, Ub);
  end;
```

Figure 2.3

2.3.2 IMPLEMENTATION

Several enhancements have been made to the basic quicksort algorithm:

1. The center element is selected as a pivot in partition. If the list is partially ordered,this will be a good choice. Worst-case behavior occurs when the center element happens to be the largest or smallest element each time partitionis invoked.

2. For short arrays, insertSortis called. Due to recursion and other overhead, quicksort is not an efficient algorithm to use on small arrays. Consequently, any array with fewer than 12 elements is sorted using an insertion sort. The optimal cutoff value is not critical and varies based on the quality of generated code.

3.Tail recursion occurs when the last statement in a function is a call to the function itself. Tail recursion may be replaced by iteration, resulting in a better utilization of stack space. This has been done with the second call to QuickSorting Figure 2-3.

4. After an array is partitioned, the smallest partition is sorted first. This results in a better utilization of stack space, as short partitions are quickly sorted and dispensed with.An ANSI-C standard library function usually implemented with quicksort. Recursive calls were replaced by explicit stack operations. Table 2.1 shows timing statistics and stack utilization before and after the enhancements were applied.

	time (μs)		*stacksize*	
count	*before*	*after*	*before*	*after*
16	103	51	540	28
256	1,630	911	912	112
4,096	34,183	20,016	1,908	168
65,536	658,003	470,737	2,436	252

Table 2-1: Effect of Enhancements on Speed and Stack Utilization

3. DICTIONARIES

Dictionariesare data structures that support search, insert, and deleteoperations. One of the most effective representations is a hash table. A simple function is applied to the key to determine its place in the dictionary. Also included are binary treesand red-black trees. Both treemethods use a technique similar to the binary search algorithm to minimize the number of comparisons during search and update operations on the dictionary. Finally, skip lists illustrate a simple approach that utilizes random numbers to construct a dictionary.

3.1 HASHTABLES

Hash tables are a simple and effective method to implement dictionaries. Average time to search for an element is O(1), while worst-case time is O(n).

3.1.1 THEORY

A hash table is simply an array that is addressed via a hash function. See Figure 3.1 is HashTableis an array with 8 elements. Each element is a pointer to a linked list of numeric data. The hash function for this example simply divides the data key by 8,

and uses the remainder as an index into the table. This yields a number from 0 to 7. Since the range of indices for Hash Table is 0 to 7, we are sure that the index is valid.

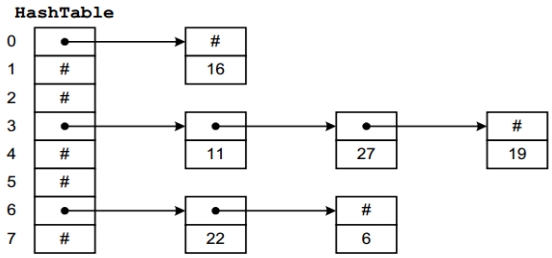

Figure 3.1

3.2 BINARYSEARCHTREES

We used the binary search algorithm to find data stored in an array. This method is very effective, each iteration reduced the number of items to search by one-half. Binary search trees store data in nodes that are linked in a tree-like fashion.[2]

3.2.1THEORY

A binary search tree is a tree where each node has a left and right child.Assuming k represents the value of a given node, then a binary search tree also has the following property: all children to the left of the node have values smaller than k, and all children to the right of the node have values larger than k. The top of a tree is known as the root, and the exposed nodes at the bottom are known as leaves. In Figure 3-2, the root is node 20 and the leaves are nodes 4, 16, 37, and 43. The heightof a tree is the length of the longest path from root to leaf. For this example the tree height is 2.

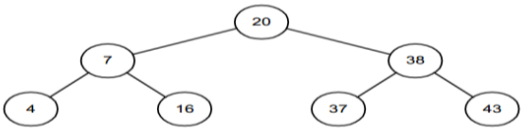

Figure 3-2: A Binary Search Tree

3.2.2 IMPLEMENTATION

Each Node consists of left, right, and parent pointers designating each child and the parent. Data is stored in the data field. The tree is based at root, and is initially NULL. Function insert Node allocates a new node and inserts it in the tree. Function delete Node deletes and frees a node from the tree. Function find Node searches the tree for a particular value.

3.3SKIPLISTS

Skip lists are linked lists that allow you to skipto the correct node. The performance bottleneck inherent in a sequential scan is avoided, while insertion and deletion remain relatively efficient. Average search time is O(lg n). Worst-case search time is O(n), but is extremely unlikely. An excellent reference for skip lists is [5]Pugh [1990].

3.3.1THEORY

The indexing scheme employed in skip lists is similar in nature to the method used to lookup names in an address book. To lookup a name, you index to the tab representing the first character of the desired entry. In Figure 3.8, for example, the top-most list represents a simple linked list with no tabs. Adding tabs (middle figure) facilitates the search. In this case, level-1 pointers are traversed. Once the correct segment of the list is found, level-0 pointers are traversed to find the specific entry.

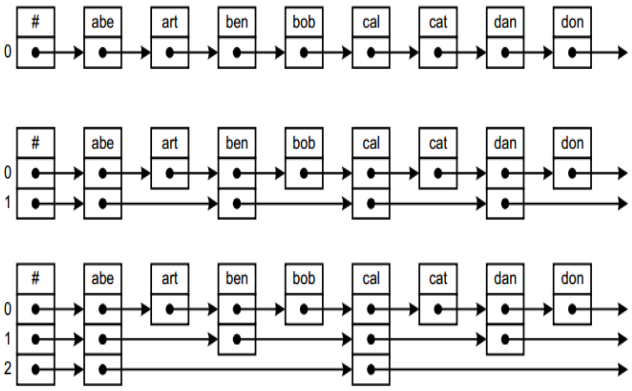

Figure 3-3: Skip List Construction

The previous algorithms have assumed that all data reside in memory. However, there may be times when the dataset is too large, and alternative methods are required. This includes techniques for sorting (external sorts) and implementing dictionaries (B-trees) for very large files. For external sorting use reference [1]Knuth[1998].and for B-tree [2]Cormen[1998] and [3]Aho[1983].

4.CONCLUSION

The research paper concludes on searching algorithms and dictionaries.Descriptions are brief and intuitive, the first section introduces basic data structures and notation. The next section presents several sortingalgorithms. This is followed by techniques for implementing dictionaries, structures that allow efficient search, insert, and deleteoperations. The last section illustrates algorithms that sort data and implement dictionaries.

5.REFERENCES

1.Aho, Alfred V. and Jeffrey D. Ullman [1983]. Data Structures and Algorithms. AddisonWesley, Reading,Massachusetts.

2.Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest [1990]. Introduction toAlgorithms. McGraw-Hill, New York.

3.Knuth, Donald. E. [1998]. The Art of Computer Programming, Volume 3, Sorting andSearching. Addison-Wesley, Reading, Massachusetts.

4.Pearson, Peter K [1990]. Fast Hashing of Variable-Length Text Strings. Communications ofthe ACM, 33(6):677-680, June 1990.

5.Pugh, William [1990]. Skip lists: A Probabilistic Alternative To Balanced Trees.Communications of the ACM, 33(6):668-676, June 1990.