Unit 14 □ Search, Sorting Algorithm and Data Structure

Structure

14.0 Objectives

The objectives of the Unit are to :

- Present an overview of data structures
- Explain the concept behind sorting algorithms
- Identify the basic classes of sorting algorithms
- Present an overview of certain sorting algorithms
- Discuss the impacts of memory on the sorting algorithms

14.1 Introduction

Algorithms and data structures are the building blocks of computer programmes. Common algorithms include searching a collection of data, sorting data, and numerical operations such as matrix multiplication. Data structures are patterns for organizing information, and often represent relationships between data values. Some common data structures are called lists, arrays, records, stacks, queues, and trees.

One of the fundamental problems is ordering a list of items. There's a plethora of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightening-fast results.

14.2 Data Structures

A major area of study in computer science has been the storage of data for efficient search and retrieval. The main memory of a computer is linear, consisting of a sequence of memory cells that are numbered 0, 1, 2, in order.

Similarly, the simplest data structure is the one-dimensional, or linear, array, in which array elements are numbered with consecutive integers and the element numbers may access array contents. Data items (a list of names, for example) are often stored in arrays, and efficient methods are sought to handle the array data.

A data structure is a way of storing data in a computer so that it can be used efficiently. A well-designed data structure allows a variety of critical operations to be performed on using as little resources, both execution time and memory space, as possible. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. Some common data structures are called lists, arrays, records, stacks, queues, and trees. For example, B-trees are particularly well-suited for implementation of databases, while routing tables rely on networks of machines to function.

In the design of many types of programmes, the choice of data structures is a primary design consideration,as experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious. Sometimes things work in the opposite direction-data structures are chosen because certain key tasks have algorithms that work best with particular data structuers. In either case, the choice of appropriate data structures is crucial.

This insight has given rise to many formalized design methods and programming languages in which data structures, rather than algorithms, are the key organizing factor. Most languages feature some sort of module system, allowing data structures to be safely reused in different applications by hiding their verified implementation details behind controlled interfaces. Object-oriented programming languages such as C++ and Java in particular use objects for this purpose.

The fundamental building blocks of most data structures are arrays, records, discriminated unions, and references. There is some debate about whether data structures represent implementations or interfaces. How they are seen may be a matter of perspective. A data structure can be viewed as an interface between two functions or as an implementation of methods to access storage that is organized according to the associated data type.

14.3 Search and sorting algorithm

Search techniques must address, for example, how a particular name is to be found. One possibility is to examine the contents of each element in turn. If the list is long, it is important to sort the data first-in the case of names, to alphabetize them. Just as the alphabetizing of names in a telephone book greatly facilitates their retrieval by a user, the sorting of list elements significantly reduces the search time required by a computer algorithm as compared to a search on an unsorted list. Many algorithms have been developed for sorting data efficiently. These algorithms have application not only to data structures residing in main memory but even more importantly to the files that consititute information systems and databases.

In computer science a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order (alphabetic order). Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output.

14.4 Classification

The common sorting algorithms can be divided into two classes by the complexity of their algorithms. There's a direct correlation between the complexity of an algorithm and its relative efficiency. Algorithmic complexity is generally written in a form known as Big-O notation, where the O represents the complexity of the algorithm and a value n represents the size of the set the algorithm is run against.

For example, O(n) means that an algorithm has a linear complexity. In other words, it takes ten times longer to operate on a set of 100 items than it does on a set of 10 items (10 * 10 = 100). If the complexity was O(n2) (quadratic complexity), then it would take 100 times longer to operate on a set of 100 items than it does on a set of 10 items.

The two classes of sorting algorithms are O(n2), which includes the bubble, insertion, selection, and shell sorts; and O(n log n) which includes the heap, merge, and quick sorts.

In addition to algorithmic complexity, the speed of the various sorts can be compared with empirical data. Since the speed of a sort can vary greatly depending on what data set it sorts, accurate empirical results require several runs of the sort be made and the results averaged together. The run times also depend on system configurations.

14.5 Summaries of the seven most popular sorting algorithms

One of the fundamental problems of computer science is ordering a list of items. There's a plethora of solutions to this problem, known as sorting algorithms. Some sorting algorithms are simple and intuitive, such as the bubble sort. Others, such as the quick sort are extremely complicated, but produce lightening-fast results. Below are descriptions to algorithms for seven of the most common sorting algorithms.

- Bubble sort
- Heap sort
- Insertion sort
- Merge sort
- Quick sort
- Selection sort
- Shell sort

14.5.1 Bubble Sort

The bubble sort is the oldest and simplest sort in use. Bubble sort is the most straightforward and simplistic method of sorting data that could actually be considered for real world use. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm, however, is vastly inefficient, and is rarely used except in education (i.e., beginning programming classes). A slightly better variant is generally called cocktail sort, and works by inverting the ordering criteria and the pass direction on alternating passes. The bubble sort is generally considered to be the most inefficient sorting algorithm in common usage.

A fair number of algorithm purists claim that the bubble sort should never be used for any reason. Realistically, there isn't a noticeable performance difference between the various sorts for 100 items or less, and the simplicity of the bubble sort makes it attractive. The bubble sort shouldn't be used for repetitive sorts or sorts of more than a couple hundred items.

14.5.2 Heap Sort

Heap sort is a member of the family of selection sorts. This family of algorithms works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list. Straight selection sort runs in O(n2) time, but Heap sort accomplishes its task efficiently by using a data structure called a heap, which is a binary tree where each parent is larger than either of its children. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. It is removed and placed at the end of the list, and then the remaining list is "heapified" again. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays-one to hold the heap and the other to hold the sorted elements.

To do an in-place sort and save the space the second array would require, the algorithm below "cheats" by using the same array to store both the heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

The heap sort is the slowest of the O(n log n) sorting algorithms, but unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. This makes it the most attractive option for very large data sets of millions of items.

As mentioned above, the heap sort is slower than the merge and quick sorts but doesn't use multiple arrays or massive recursion like they do. This makes it a good choice for really large sets, but most modern computers have enough memory and processing power to handle the faster sorts unless over a million items are being sorted.

The "million item rule" is just a rule of thumb for common applications-high-end servers and workstations can probably safely handle sorting tens of millions of items with the quick or merge sorts.

14.5.3 Insertion Sort

Insertion sort is similar to bubble sort, but is more efficient as it reduces element comparisons somewhat with each pass. An element is compared to all the prior elements until a lesser element is found. In other words, if an element contains a value less than all the previous elements, it compares the element to all the previous elements before going on the the next comparison. Although this algorithm is more efficient than the Bubble sort, it is still inefficient compared to many other sort algorithms since it, and bubble sort, move elements only one position at a time. However, insertion sort is a good choice for small lists (about 30 elements or fewer), and for nearly-sorted lists. These observations can be combined to create a variant of insertion sort which works efficiently for larger lists. This variant is called shell sort (see below).

The insertion sort works just like its name suggets-it inserts each item into its porper place in the final list. The simplest implementation of this requires two list structures-the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place.

Like the bubble sort, the insertion sort has a complexity of O(n2). Although it has the same complexity, the insertion sort is a little over twice as efficient as the bubble sort.

The insertion sort is a good middle-of-the road choice for sorting lists of a few thousand items or less. The algorithm is significantly simpler than the shell sort, with only a small trade-off in efficiency. At the same time, the insertion sort is over twice as fast as the bubble sort and almost 40% faster than the selection sort. The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items.

14.5.4 Merge Sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e. 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list.

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. Each array is recursively sorted, and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of O(n log n).

Elementary implementations of the merge sort make use of three arrays-one for each half of the data set and one to store the sorted list in. The below algorithm merges the arrays in-place, so only two arrays are required. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines.

The merge sort is slightly faster than the heap sort for larger sets, but it requires twice the memory of the heap sort because of the second array. This additional memory requirement makes it unattractive for most purposes-the quick sort is a better choice most of the time and the heap sort is a better choice for very large sets. Like the quick sort, the merge sort is recursive which can make it a bad choice for applications that run on machines with limited memory.

14.5.5 Quick Sort

Quicksort takes an element from an array and places it in its final position whilst at the same time partitioning the array so that all elements above the partition element are larger and all elements below are smaller. It then sorts each sub-array (partition) recursively. The sort is usually implemented by scanning p from the bottom of an array until an element larger than the partition element is found, then scanning down from the top for an element smaller than the partition element; these two elements are then swapped. When the scans from bottom and top meet the partition element is swapped into its final position. Quicksort is an in place sort so it has modest memory requirements and does not involve copying, if implemented carefully bad / worst case performance is extermely unlikely, and quicksort is probably the fastest sorting method. Quicksort runs in O(n log n) time.

As mentioned earlier, it's massively recursive (which means that for very large sorts) you can run the system out of stack space pretty easily. It's also a complex algorithm-a little too complex to make it practical for a one-time sort of 25 items, for example. Ironically, the quick sort has horrible efficiency when operating on lists that are mostly sorted in either forward or reverse order-avoid it in those situations.

14.5.6 Selection Sort

The selection sort works by selecting the smallest unsorted item remaining in the list, and then swapping it with the item in the next position to be filled. The selection sort has a complexity of O(n2).

The selection sort is the unwanted stepchild of the n2 sorts. It yields a 60% performance improvement over the bubble sort, but the insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort. In short, there really isn't any reason to use the selection sort-use the insertion sort instead.

If you really want to use the selection sort for some reason, try to avoid sorting lists of more than a 1000 items with it or repetitively sorting lists of more than a couple hundred items.

14.5.7 Shell Sort

Donald Shell invented shell sort in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two dimensional array (in reality, the array is an appropriately indexed one dimensional array) and then sorting the columns of the array using the Insertion sort method. Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements (sets with less than 1000 or so elements). Another advantage of this algorithm is that it requires relatively small amounts of memory. The shell sort is the most efficient of the O(n2) class of sorting algorithms. Of course, the shell sort is also the most complex of the O(n2) algorithms.

The algorithm makes multiple passes through the list, and each time sorts a number of equally sized sets using the insertion sort. The size of the set to be sorted gets larger with each pass through the list, until the set consists of the entire list. (Note that as the size of the set increases, the number of sets to be sorted decreases). This sets the insertion sort up for an almost-best case run each iteration with a complexity that approaches O(n).

The items contained in each set are not contiguous-rather, if there are i sets then a set is composed of every i-th element. For example, if there are 3 sets then the first set would contain the elements located at positions 1, 4, 7 and so on. The second set would contain the elements located at positions 2, 5, 8, and so on; while the third set would contain the items located at positions 3, 6, 9, and so on.

The shell sort is by far the fastest of the N2 class of sorting algorithms. It's more than 5 times faster than the bubble sort and a little over twice as fast as the insertion sort, its closest competitor.

The shell sort is still significantly slower than the merge, heap, and quick sorts, but its relatively simple algorithm makes it a good choice for sorting lists of less than 5000 items unless speed is hypercritical. It's also an excellent choice for repetitive sorting of smaller lists.

14.5.8 Comparative Chart

The following table presents an overview of certain parameters of the above-discussed sort algorithms.

Sorting Algorithms	Class	Advantages	Limitations
Bubble Sort	O(n2)	Simplicity and ease of implementation.	Horribly inefficient.
Heap Sort	O(n log n)	In-place and non-recursive, making it a good choice for extremely large data sets.	Slower than the merge and quick sorts.
Insertion Sort	O(n2)	Relatively simple and easy to implement.	Inefficient for large lists.
Merge Sort	O(n log n)	Marginally faster than the heap sort for larger sets.	At least twice the memory requirements of the other sorts; recursive.
Quick Sort	O(n log n)	Extremely fast.	Very complex algorithm, massively recursive.
Selection Sort	O(n2)	Simple and easy to implement.	Inefficient for large lists, so similar to the more efficient insertion sort that the insertion sort should be used in its place.
Shell Sort	O (n2)	Efficient for medium size lists.	Somewhat complex algorithm, not nearly as efficient as the merge, heap, and quick sorts.

14.6 Memory Usage Patterns and Index Sorting

When the size of the array to be sorted approaches or exceeds the available primary memory, so that disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm.

For example, the popular recursive quicksort algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM.

References and Further Readings

1	2005	Algorithm and data structure. (http://linux.wku.edu/~lamonml/algor). Visited last : 26/10/2005.
2	1996	Majumder (Arun K) and Bhattacharyya (Pritimoy). database management systems. New Delhi : Tata McGraw-Hill, 1996.
3	1994	Elmasri (Ramez) and Navathe (Shamkant B). Fundamentals of database systems. California : Benjamin/Cummings, 1994.
4	1985	Data (CJ). An Introduction to database systems. 3rd ed. New Delhi : Narosa, 1985.
5	1983	Date (CJ). Database : a primer. Reading : Addison-Wesley, 1983.

14.7 Exercise

1. Discuss different search algorithms.
2. What is an algorithm ?