

Chapter 31 – Searching a Sorted List

1. Using a Sorted List

If a list is to be used frequently, placing its elements in order can greatly improve the efficiency of the algorithms that are to use the list.

Once a list is sorted, finding the minimum and maximum values in the list becomes a trivial operation. For example, if the array A contains a list of doubles that has been sorted into ascending order, the smallest value in the list is located at $A[0]$ and the largest value in the list is located at $A[\text{listlength} - 1]$.

Finding an element in a sorted list becomes more efficient. A sorted list does not have to be searched from end to end in order to determine that an element is not present. Instead, the list need only be searched until the place the value would have occupied is passed.

Finding an element in a sorted list stored in an array can become even more efficient by use of a binary search (covered in the next chapter). Binary search is much like the game where a number is picked within a certain range. The player makes a guess and the person who made up the number responds with information as to whether the user's guess is high or low. Since numbers are always sorted, the question becomes how many moves will the player take to find the correct value, not if it will be found.

As seen in the previous chapter, insertion into a sorted list is more complex. No longer can a new element just be tacked onto the back of the list. If the list is stored in an array, the insertion algorithm from the insertion sort must be used to find the correct location for the new element, all the while moving elements out of the way.

2. Search of a Sorted List by Traversal

To search a sorted list that is stored in an array, the algorithm of searching an unsorted list is changed slightly. *Note: For the purposes of this discussion, it is assumed that the list is sorted into ascending order.* First, the while statement runs until the list is exhausted or the item to be found is no longer greater than the value found in the current element of the array. If the item to be found is no longer greater than the value found in the current element for the array, the item may or may not be in the list. While the final question has not been answered, the rest of the list does not have to be traversed. Instead, an *if* test can determine if the item is in the list or if the function should return a value that will indicate that the element has not been found.

```
int search(elementtype array[ ], int listlen, elementtype item) {
    int index = 0;
    while (index < listlen && item > array[index]) index++;
    if (index < listlen)
        return index; // return index of located item
    else
        return -1; // return impossible index to indicate item not found
}
```

A slight variation on this algorithm removes one of the tests in the *while* statement, making the Boolean expression of the *while* statement much more efficient. However, it depends on the array being at least one element longer than the list. In this variation, the value sought is copied to the element of the array at the location *listlen*, which is one index past the last element of the list stored in the array. Since the value sought will be found, all that is necessary to test for is the presence of the value sought which halves the length of the Boolean expression. When the loop finishes, the same if test can be used to determine if the element was found or not.

```
int search(elementtype array[ ], int listlen, elementtype item) {
    int index = 0;
    array[listlen] = item;
    while (item > array[index]) index++;
    if (index < listlen)
        return index; // return index of located item
    else
        return -1;    // return impossible index to indicate item not found
}
```

Analysis on Time: In either variation, the search of a sorted list will traverse on average $\frac{1}{2}$ the list or $\frac{1}{2}n$ elements, whether the item sought is found or not. If the list is searched enough times, this will make the algorithm run $\frac{1}{2}n^2$ times. Since the multiplier $\frac{1}{2}$ is a constant and can be discarded, the worst case growth in time of the traversal search of an ordered list algorithm is described by the curve of n^2 or $O(n^2)$.

Programming Assignment 31.1

When a list consists of elements that are compound (that is each element contains more than one data field such as in the case of a list stored in an array of structs), the list is usually sorted and searched on one or more of the data fields, but not all of the data fields. The data fields chosen for sorting or searching are referred to as the *key*.

Insert users into the *phonebook* class (from chapter 26.2 and which was later modified in chapter 29.3) via the insertion algorithm so that the list is sorted on a *key* of last name and first name. Alter all the search routines so that search of a sorted list by traversal is used.

*Hint: You cannot concatenate the last name and first name directly together to make one key field to sort on. This could result in Xavier Provos being placed in the list after Aphonse Provost, which would be the wrong way around. What you can do is use the *strlen* function (for C-strings) or the *length* member function (for strings) to find the number of characters in the last name string, then concatenate the appropriate number of blanks to completely fill the last name string (a pre-determined maximum in the case of a string type), then concatenate the first name string to the last name string to make one key variable. If the last name was to always take up 15 spaces in the key and dashes indicate spaces, here is an example:*

*Firstname is Fred, last name is Jones. The key
would be Jones- - - - - Fred*