Data Structures and Algorithms – Problems from a Previous Semester

The midterm for the Spring 2003 semester will consist of about 5 or 6 smaller problems. The problems below are for your reference only.

Problem 1

The median is the middle element of an odd-sized array. If the array, on the other hand, has an even number of elements, the median is the average of the middle two elements. Suppose an approximation of this selects one of the two middle elements if the size of the array (or list) is even. Suppose the array has *n* elements. Implement a "median heap", similar to the min and max-heap. The root node would be the median of the entire array. Its left child is the median of the array that excludes the element that was picked as the median (size of the array is *n*-1). The right child is the median of the array excluding the two medians that were identified (size of the array is *n*-2). This process continues until all the array elements are exhausted. Allow the user to enter an array of size 11, which you will use to build the median heap. Print the original array as well as the content of the heap in preorder traversal.

Problem 2

Implement an array of linked lists, which will be used to count the number of identical characters in a string (size of the array is the number of characters in the Alphabet). All identical characters would go into the same linked list. Provide an interface to allow the user to enter a string of up to 70 characters (use the *cin.getline*() function). A function, which will be used by your program, would traverse the each list to count the corresponding number of occurrences for each character and print it out.

Problem 3

```
Void bubsort(Elem A[], int n)
{
   for (int I=0; I<n-1; I++)
       for (int j=n-1; j>I; j--)
           if (Comp::lt(A[j], A[j-1]))
              swap(A, j, j-1);
}
```

The **bubblesort** algorithm, as defined above, can be improved to take less time in certain cases (if the data is sorted or partially sorted). Identify this improvement and make the change. Test your changed function against the old one using the array:

```
A[]={5, 12, 0, 3, 6, 4, 8, 3, 2, 1, 7, 14, 15, 18, 20};
```

Display the number of passes and number of swaps in each case.

Problem 4

We want to sort the data so the maximum element is in the middle (size is odd) or near the middle (size is even). The next maximum element is to the left of the maximum and the next-next maximum is to the right of the maximum, and so on. For example, if A is the original array and B is the sorted array, then we have:

```
A[]={5, 12, 0, 3, 6, 4, 8, 3, 2, 1, 7, 14, 15, 18, 20};
B[]={1, 2, 3, 4, 6, 8, 14, 18, 20, 15, 12, 7, 5, 3, 0};
```

Design what we will call *MountainSort* using any of the 9 sorting algorithms. Generate 20 random values between 0 and 20 and store in an array. Sort this array using *MountainSort* and display the results.

Problem 5

Create 7 student records. Each record consists of first name (up to 16 characters), last name (up to 16 characters), ID (long int), rank (int – specifying 1st year, 2nd year, etc.), and major (up to 24 characters). Design the user interface, which allows the user to enter the data for the seven students. Suppose we want to use the last name as the key for sorting. Form an **index** array, which will have the sorted last names and pointers to the corresponding records. Therefore, you do not sort the records, but **only** the last names along with the pointers that point to the corresponding records. After sorting the last names, display the records in sorted order. You may use any sorting algorithm to sort the names and you can use structures to represent records.

Problem 6

Declare an array of 100 characters, which will be used to store 10 names. Each name consists of a maximum of 10 characters and is terminated by a NULL character. To locate the individual names, an integer (position) array of 10 elements is used, where each value in this array indicates the start of the corresponding name (see example below)

'F','a','d','i','\0','A','b','b','a','s','\0','B','a','d','e','r','\0','M','o','u','e','e','n','\0','K','a','r','i','m','\0','A','l','l','\0', ...

0, 5, 11,17, 24, 30, …

To sort the names, we simply move the values in the position array to the corresponding location. In other words, instead of swapping strings, we just swap positions (values in the position array). For the above example, we would have the following sorted position array:

5, 30, 11, 0, 24, 17, …

Implement the above algorithm and prove it using an example.

Remember that an array can be considered a pointer. For example,

if

```
char c[5]={'a','b','c','\0','d'}; char *p; p=c;
```

then

`*p` is `'a'`, `*(p+1)` is `'b'`, `*(p+2)` is `'c'`, **and** `p` is `"abc"`.

Problem 7

Suppose we have three arrays:
Array1 has 10 values that are random integers (use the *rand*() function) between 1 and 20
Array2 has 6 values that are a random integers between 1 and 12
Array3 has 4 values that are random integers between 13 and 20

We need to sort the values across the arrays **WITHOUT** using temporary arrays. That is, Array1 will have the smallest 10 values, in sorted order, of all 20 values in the three arrays. Array2 will have the next 6 values in sorted order. And, array 3 will have the largest 4 values in sorted order. Your code must display the contents of each array before and after sorting. Include the function call *srand*(*time*(0)) at the top of your program so the randomly-generated numbers are different every time you run your program.

Problem 8

Suppose we have the following array

```
5, 3.1, 10.4, 0.99, 2.4, 1.72, 7.7, 2.11, 4.45, 2.08, 8.248, 6, 4.5, 9
```

Sort it using the Radix Sort algorithm