CS 1723, Data Structures
Fall Semester, 1998
Final Examination

1. Give C code for the function **`strlen`** that will return the length of its character string input parameter. *Your function must not use brackets [].* (**`strlen`** is a C library function in **`<string.h>`**).

2. (a) Regard the following array as a heap. Draw a tree diagram for this heap, and determine that the heap property fails at location 2. Show step-by-step how the heapify function will restore the heap property. (This example uses arrays based at 1, as we have done with examples in class.)

Array index:	1	2	3	4	5	6	7	8	9	10
Array element:	43	5	28	18	13	22	14	12	14	11

   (b) The following array satisfies the heap property. Show the first two steps of the heapsort algorithm. These steps should place two array elements in their proper locations and should restore the heap property for the remaining elements. Show the final result of these first two steps.

Array index:	1	2	3	4	5	6	7	8	9	10	11	12
Array element:	57	31	45	27	24	36	16	14	12	8	19	21

3. Suppose we have a linked list of single characters, using the struct:

```
struct lnode {
   char data;
   struct lnode *next;
};
```

   (a) Write a function **`listlength`** that will find the length of this list by chasing down it and counting the nodes. **`listlength`** should return the length. Use the prototype:

```
int listlength(struct lnode *list);
```

   (b) Write a function **`makelist`** that will take an input character string and will create a linked list with each character of the string in a separate node of the list. (The nodes can be in backwards order.) The function should return a pointer to the start of the list.) You should use a prototype:

```
struct lnode *makelist(char *s);
```

4. Recall that when C processes a command line, like say: **`runner% cc -o prog prog.c`**, C sets up variables **`argc`** and **`argv`** as if they were initialized as follows:

```
int argc = 4;
char *argv[] = {"cc", "-o", "prog", "prog.c", NULL};
```

   By any method, give a loop that will print each string in the array **`argv`**, one to a line.

5. Consider the following portion of the binary tree program discussed in class. This tree is keeping a single character at each node. The **addtree** function will add nodes so that alphabetic order is the natural order of the tree.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

struct tnode {
   char ch;
   struct tnode *left;
   struct tnode *right;
};
void          addtree(struct tnode *, char );
struct tnode *newnode(char );
void          treeprint(struct tnode *);
struct tnode *lookup(struct tnode *, char );

void main(void)
{
   struct tnode *root;
   struct tnode *p;
   char c;
   root = NULL;
   while ((c = getchar()) != '\n') {
      if (root == NULL) root = newnode(c);
      else addtree(root, c);
   }
   treeprint(root);
   printf("\n");
   p = lookup(root, 'x');
   if (p == NULL) printf("Not found\n");
   else printf("Here it is: %c\n", p -> ch);
}
```

```
runner%  exam2_3
The quick brown fox jumps over the lazy dog.
 .Tabcdefghijklmnopqrstuvwxyz
Here it is: x
runner%  exam2_3
The quick brown pig jumps up in the air.
 .Tabceghijkmnopqrstuw
Not found
```

(a) Supply code for a function **treeprint** that prints the nodes in alphabetic order.

(b) Supply code for a **lookup** function that will look up the character that is its second argument. If it finds the character, it returns a pointer to the node containing the character, and otherwise it returns **NULL**. (**lookup** is similar in logic to the **addtree** function, but simpler because there is no need to add a node.)

6. Write C code for a recursive binary search function. This function will look up an element **x** in a sorted array **a**. The function should use a divide-and-conquer strategy similar to quicksort. (This function does not use a binary tree, but just repeatedly divides the input array in half.) The function should have the following prototype:

```
int bin_search(int a[M], int p, int r, int x);
```

bin_search should search for the element **x** in the array **a**, in positions **a[p]**, ... , **a[r]**. The array **a** is assumed to be sorted into increasing order. The function should use the following strategy: check the array element in the middle between elements with index **p** and **r** — call this middle index **q**. In case **x == a[q]**, return **q**. In case **x < a[q]**, let **bin_search** work recursively on the positions from **p** to **q-1** and in case **x > a[q]**, let **bin_search** work recursively on the positions from **q+1** to **r**. Your code must also decide how to terminate the recursion. In case **x** is not in the array, return **-1**.

This **bin_search** function might be called from a C main function as follows:

```
int a[M];
/* obtain values for a[0], . . . , a[M-1] */
/* sort a[0], . . . , a[M-1] into increasing order */
/* obtain a value x to search for in the array a */
index = bin_search(a, 0, M-1, x);
if (index == -1) printf("Element %d not present\n", x);
else printf("Element %d present at location %d\n", x, index);
```