Physics PHZ-5156-C, 23 August 2004

Week 1: Histograms and an Introduction to C

Copy the question files and suggested program templates from /home/5156/assignments/week1, for example using the following commands:

```
cd
cp -r /home/5156/assignments/week1 .
cd week1
```

start at your home directory

When you have completed everything, the `make` command should compile and demonstrate your programs. Feel free to modify the `Makefile` if you wish. The grading for this week is S/U, with “S” allowed for any substantial effort; we expect that students with more programming experience will help those with less. Type `make submit` when you are ready to submit your work.

Often one measures something that should have a single value but finds a distribution of values. Standard courses in statistical techniques tell us how to estimate the true underlying value (often the mean is best justified) and variance. This week, however, we’d just like to draw a histogram as an approximation to the shape of the distribution of fluctuations. If we see two separated peaks in the histogram, we might begin to wonder whether in fact the thing we are measuring can take two different true values.

The next page of notes gives you your assignment. To help any students with less experience in the C programming language, I’ve provided partly-completed template files. I shall be available several evenings this week in room 102 to work individually or in a group with students requiring a review.

The rest of the notes are essentially props to help with the C language. These include some comparisons of C to other languages. The comparison of quick-sort routines demonstrates how it is possible to write very concise, easy-to-read code. The following example shows how the opposite is also possible. Some people count the mere possibility of writing obfuscated code against C, but then the same criticism could apply to English or French. Just as everyone writes text differently, so everyone writes C differently. You will be exposed to at least three programming styles. Of these, that in Numerical Recipes is probably the worst: the authors are FORTRAN programmers who haven’t mastered the C idiom. In all cases, your goal should be code that you or someone else will be able to pick up in five years and still understand completely. Short subroutines and copious comments are the most important keys to achieving this. In this course, we are not going to be concerned with how fast your program runs. Once you start writing programs for research, you will find that typical programs spend nearly all their time in a few very well isolated pieces of code; it therefore makes better sense to worry about optimizing these sections only after everything works than to take ten times as long to write the program by squeezing every possible optimization out of the irrelevant parts. We distinguish, however, algorithms from code optimization: often you will achieve an enormous savings in time (both execution and programming) by picking the right algorithm from the start.

Continuing in the packet, we offer some elementary exercises in preparation for the week’s assignment, information on how to compile and debug programs, pictures modeling how C treats pointers and argument-passing internally, a binary-tree sorting program (to demonstrate the use of `structs`), and a list of common programming mistakes. It will help if you bring your copy of these notes to the laboratory so we can refer to them in explaining some point or another.
We shall write three programs useful for the manipulation of data. Even before you have written them yourself, you will be able to test working versions. These commands extract columns of data from a file, collect the data into histogram bins, and then convert the histogram bins for graphing by the `axis` program. Using the working versions, first try them out. You will already have copied the `data` file into your directory; it contains three columns of manufactured data. The following command will plot a histogram of the second column:

```
extract 2 < data | histbin | hist2axis | axis | xplot
```

The data in column 2, all of which here happen to lie between 0 and 1, have been grouped into ten bins, with the number in each bin plotted vertically: so the first bin, for data greater than or equal to 0 but less than 0.1, contains 50 data, the second bin, between 0.1 and 0.2, 51 counts, the third 45, etc.

Histograms find use wherever measurements include noise or a fundamentally stochastic element. For example, in Chapter 4, we'll use the histogram program written here to analyze data from a cat's brain stem; one could apply the same program to a distribution of scattering angles in particle physics or to the spacings between energy levels of electrons in a ring.

`extract` While in the example above, we wished to extract only a single column, in general, the `extract` program should know how to extract an arbitrary number of columns. For example, to plot column two against column three, we could give the command

```
extract 2 3 < data | sort -n | axis | xplot
```

You will already have copied an outline `extract.c` that includes comments and templates for writing the various functions needed as well as a complete `util.c` containing some miscellaneous utilities needed for the project and a `Makefile`. Once you have completed `extract.c`, the command `make extract` should compile and link the program.

The top of `extract.c` contains a detailed description of what the program should do. Once the program has compiled, you should test it with your own data set or with the `data` file provided, making sure it does what you expect. (The template also suggests an optional feature, the processing of comment lines embedded in the data. I like to use such lines to track when and how data were created and later processed.)

`histbin` This program reads in a single column of numbers, determines the range of the data (the extremal numbers all the data lie between), puts them into bins, and prints out two columns. The first column is the midpoint of a bin, the second the number of counts in that bin. If an argument is given to `histbin` on the command line, it specifies how many equal-width bins `histbin` should create. Otherwise, the program makes ten. I have provided in `histread.c` a `histread()` function to read all the lines of the file into an array; internally, it uses a singly-linked list before converting the list to an array for your use. Again, the top of `histbin.c` describes in greater detail what the program should do.

`hist2axis` The public-domain `axis` program prints a graph of x and y values in the first and second columns; in its simplest form, `axis` merely connects the dots with lines and draws a graduated frame. The output of `histbin` above can be piped to `axis`,

```
extract 2 < data | histbin | axis | xplot
```

but the result doesn’t look very good. We prefer to represent the number of counts in each histogram bin as a box of some height rather than as a point in an x-y plot. The `hist2axis` program accomplishes this by writing the x and y values of the box’s corners on successive lines, which `axis` then connects with lines.
1. Histograms and an Introduction to C

Comparison of C and FORTRAN

Quick-sort Algorithm

/* qsort.c */
#define MINFORQ 7 /* if fewer than MINFORQ elements, use insertion sort */

void swap(int *v, int i, int j) {
    int temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

/* Insertion sort: good when there aren't too many to sort (n=length). */
void insertionsort(int *vec, int n) {
    int *vleft, *vright = vec + 1, nright = n, nleft = 1, ninner, a, b;
    while (--nright > 0) {
        a = *vright;
        for (ninner = nleft++, vleft = vright++ - 1; ninner--; vleft--)
            if ( (b = *vleft) < a) break; /* found place */
        vleft[1] = b; /* shuffle */
        vleft[1] = a;
    }
}

void qsort(int *vec, int left, int right) {
    int i, last;
    if((i = right - left + 1) < MINFORQ) { /* more efficient */
        insertionsort(vec + left, i); /* to call this alg */
        return;
    }
    swap(vec, left, (left + right) / 2); /* move partition elem */
    last = left; /* to vec[left] */
    for (i = left + 1; i <= right; i++) /* partition */
        if (vec[i] < vec[left])
            swap(vec, ++last, i);
    swap(vec, left, last);
    qsort(vec, left, last - 1);
    qsort(vec, last + 1, right);
}

Why implicit typing is bad:

Do 1 I=1.100
#include <time.h>
#include <curses.h>
#define P(A,B,C,D,E) mvaddch(b+A,a+B,(q[y]&C)?D:E);
#define O(A,B,C) case A:if(q[x]&B)C;break;
#define R rand()
#define U 0,1,4,5
#define J(x) (1<<x)
#define V ' ' 

int r[27] = {0,J(0)
,2,1,3,1,
,3,5,2,6,5,
,4,6,4,5,6,
,7,4,9,-49,-7,1,
,q[343],x,y,d,l=342,a,b,
,j='w';int main(){srand(time(0));for(x
=0;x<343;x++)q[x]=0;x=R%343;while(l--

==((x+u[d] )/0x31)
==(((x+u[d] )%7)==((x+u[d] )/7)
==((x+u[d] )%7)==((x+u[d] )/7)
 unforec())

;crmode();clear(11);while(

while(1){mvaddch("You Escaped!\n",x=294+R%0x31;initscr()

;noecho();crmode()
;refresh()
;while(


}mvaddch(J(1),+((y
%7)*3),J
(1)+((y

ref() ;mvaddch(1+((x%7
/7)*7)+3
,J(1+)++;((x/7)%
7)+3,')
;refresh()

;mvaddch(J(1)+((x%7
%7)*3),J(1+)
++;((x/7)%
7)+3,')
;refresh()

;mvaddch(J(1)+((x%7
%7)*3),J(1+)
++;((x/7)%
7)+3,')
;refresh()

;mvaddch(J(1)+((x%7
%7)*3),J(1+)
++;((x/7)%
7)+3,')
;refresh()

;mvaddch(J(1)+((x%7
%7)*3),J(1+)
++;((x/7)%
7)+3,')
;refresh()

;mvaddch(J(1)+((x%7
%7)*3),J(1+)
++;((x/7)%
7)+3,')
;refresh()

;mvaddch(J(1)+((x%7
%7)*3),J(1+)
++;((x/7)%
7)+3,')
;refresh()
### Some items of C syntax compared to other languages

<table>
<thead>
<tr>
<th>meaning</th>
<th>C</th>
<th>FORTRAN IV/77</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>end of statement</td>
<td>;</td>
<td>end of line</td>
<td>; (only between statements)</td>
</tr>
<tr>
<td>line continuation</td>
<td>not needed</td>
<td>digit in column 6</td>
<td>not needed</td>
</tr>
<tr>
<td>comment</td>
<td>/<em>...</em>/</td>
<td>cols. 72-80, C col. 1</td>
<td>(<em>...</em>)</td>
</tr>
<tr>
<td>begin block</td>
<td>{</td>
<td>IF, DO, etc.</td>
<td>begin</td>
</tr>
<tr>
<td>end block</td>
<td>}</td>
<td>ENDF, etc.</td>
<td>end</td>
</tr>
<tr>
<td>i holds an integer</td>
<td>int i;</td>
<td>INTEGER I</td>
<td>var i: integer</td>
</tr>
<tr>
<td>x holds a real number</td>
<td>double x;</td>
<td>REAL*8 X</td>
<td>var x: double</td>
</tr>
<tr>
<td>s holds a 100-byte string</td>
<td>char s[100];</td>
<td>CHARACTER*100 S</td>
<td>type str100=array[1..100] of character; var s: str100</td>
</tr>
<tr>
<td>implicit typing</td>
<td>generally not allowed</td>
<td>allowed, causes trouble</td>
<td>not allowed</td>
</tr>
<tr>
<td>composite object</td>
<td>struct</td>
<td>COMMON (very limited)</td>
<td>record</td>
</tr>
<tr>
<td>assignment</td>
<td>=</td>
<td>=</td>
<td>:=</td>
</tr>
<tr>
<td>equality test</td>
<td>==</td>
<td>.EQ.</td>
<td>=</td>
</tr>
<tr>
<td>inequality test</td>
<td>!=</td>
<td>.NE.</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>Boolean false</td>
<td>0</td>
<td>.FALSE.</td>
<td>false</td>
</tr>
<tr>
<td>Boolean true</td>
<td>.TRUE.</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>Boolean subexpressions</td>
<td>stop when determined</td>
<td>evaluate all</td>
<td>evaluate all</td>
</tr>
<tr>
<td>parameter passing</td>
<td>by value or pointer</td>
<td>by reference</td>
<td>by value or reference</td>
</tr>
<tr>
<td>function return</td>
<td>return value</td>
<td>name=value</td>
<td>name=value</td>
</tr>
<tr>
<td>function returning double</td>
<td>double name()</td>
<td>FUNCTION name</td>
<td>function name: double</td>
</tr>
<tr>
<td>function returning nothing</td>
<td>void name()</td>
<td>REAL*8 name</td>
<td>procedure name</td>
</tr>
<tr>
<td>local storage</td>
<td>stack, constant, or static</td>
<td>implementation-dependent</td>
<td>stack or constant</td>
</tr>
</tbody>
</table>
examples and exercises

1. Hello  /* hello.c
   * 7/96 D. Rabson
   * The canonical "hello, world" program in C.
   */
#include <stdio.h>

int
main()
{
    printf("Hello, world\n");
    return 0; /* required by ANSI standard */
}

2. Temperature table
   /* ttable.c
   * 7/96 DAR
   * Print a table of Celsius temperatures and their conversions to Fahrenheit.
   * Exercise: modify the print format so that it comes out prettier.
   */
#include <stdio.h>

int
main()
{
    double celsius; /* hold the Celsius temperature */

    /* For loop: first part is initialization, second is the
     * condition to be tested before each iteration of the loop,
     * last is an action to take at the end of each iteration. Any
     * of the parts may be blank (although it's not sensible for the
     * second one to be blank).
     */
    for(celsius=5; celsius<=40; celsius+=2)
        printf("%f %f\n", celsius, 1.8*celsius + 32.);

    return 0; /* ANSI standard requires this. */
}

3. Using an array
   Make a lookup table of Farhenheit equivalents for integral Celsius temperatures
   from 0 to 25, then print it. Here's a start. Be sure to add comments.
   #define LARGEST 25  /* largest C temperature */
int
main()
{
    double fahrenheit[LARGEST+1]; /* room for 0 to LARGEST */
    void initialize_table(double [], int); /* below */
    void print_table(double *, int); /* below */

    initialize_table(fahrenheit, LARGEST+1); /* initialize */
    print_table(fahrenheit, LARGEST+1); /* print it out */
    return 0;
}
4. Fahrenheit–Celsius

The program is correct as printed here, but you will be given a version with two bugs slipped in. Use a debugger to find them.

```c
/* fahr2cels.c
* 7/96 DAR
* convert Fahrenheit to Celsius degrees
*     C = (5/9)(F - 32)
* *
* usage:
*    fahr2cels < input > output
*    converts each line of input from Fahrenheit
*    to Celsius, ignoring blank lines. Other lines that
*    cannot be converted into numbers will be interpreted
*    as zero.
*    fahr2cels number number number ...
*    converts each argument from Fahrenheit to Celsius
* *
* I've written this to demonstrate as many as possible of the items in
* the syntax-comparison chart: it could be much shorter.
*/

#include <stdlib.h> /* include standard declarations */
#include <stdio.h> /* standard I/O declarations */
#include <ctype.h> /* includes "isspace()" */

/* preprocessor directives: these take effect when the compiler reads
* the source code.
*/
#define SCALE (5./9.) /* note: no semicolons */
#define OFFSET (32)

/* Define a preprocessor "function" -- beware side effects! */
#define MAX(a,b) ( (a)>(b) ? (a) : (b) )

/* Statics: these are accessible by any function in this file. */
static double absolute0 = -273.15; /* absolute zero in degrees Celsius */

/* The main program: if called with any arguments, it tries to interpret
* each in turn as a Fahrenheit degree, to be converted to Celsius. If
* there are no arguments, the program reads lines of input.
* *
* Note that argv[0] is the name by which the program was called.
* Note that we've not declared a type for main(). The type of a
* function (but not a variable) defaults to "int."
*/
main(int argc, char **argv)
{
    static void print_conversion(double); /* a function defined below */
    char buf[1024]; /* room to hold a line */
    static char *find_first_nonspace(char *); /* a function below */
    char *line; /* a line we got from fgets() */

    if(argc>1) { /* braces optional: block 1 statement */
        while(--argc) /* ditto -- left out here */
            /* atof() is declared in stdlib.h */
            print_conversion(atof(*++argv));
    } else {
        /* fgets() is declared in stdio.h */
```
while((line=find_first_nonspace(fgets(buf,sizeof(buf)-1,stdin)))
  != 0 ) { /* the !=0 is optional */
  if(!line[0])
    continue;    /* blank line */
  print_conversion(atof(line));
}

return 0;    /* ANSI C requires return value */

/* Print the conversion of a double from Fahrenheit to Celsius. */
/* "static" means that it can be called only from this file. */
static void
print_conversion(double f)
{
  /* This is a somewhat silly way to initialize the scale factor
   * of 5/9, but I do it this way for demonstration purposes.
   */
  static double scale = 0;  /* 0 before this is ever run */
  double c;    /* hold a degrees-Celsius temperature */

  if(!scale)        /* first time run only */
    scale = SCALE;

  /* I don’t combine these two lines into one, because the MAX()
   * preprocessor macro evaluates one of its arguments twice.
   */
  c = scale*(f-OFFSET);
  printf("%.1f\n", MAX(c,absolute0) );    /* print Celsius conversion */
}

/* Find and return a pointer to the first non-space in a string. */
static char *
find_first_nonspace(char *s)
{
  /* I’ve deliberately written this less than optimally in order to
   * demonstrate some pieces of syntax.
   */
  while(s && *s && isspace(*s) && s++)
    ;
  return s;
}
Mechanics of compiling and debugging a C program

```
cc -g -o program program.c  # (-g allows debugging)
```

```
cc -g -c prog1.c
cc -g -c prog2.c
cc -g -o program prog1.o prog2.o -lm
```

```
shell% cat Makefile
CFLAGS=-g
LDLAGS=-g
LIBS=-lm  # -lm to link to math library
program: prog1.o prog2.o
          cc $(LDLAGS) -o program prog1.o prog2.o $(LIBS)
shell%
shell% make
```

A typical compilation actually goes through several steps. For instance, when I say `cc -c program.c`, the file `program.c` is first passed through the C preprocessor, which expands the `#` directives. It then goes through one or more passes of the main compiler, including optional optimization (`-O`), to create a temporary assembly-code file. The assembly code can be saved in a `.s` file with the `-S` switch. The assembler, `as`, is then called to create an object (.o) file, which is finally linked together by `ld` with one or more startup modules and numerous default and user-specified run-time and static libraries (see the `ld` manual page for the `-l` and `-L` flags) to create an executable file.

Debugging

“Don’t make mistakes; they’re a waste of time.”

**TRACER** Scattering `printf` calls throughout a program is often the easiest debugging method. My files `/home/5156/rabsonlib/src/debug.h` and `/home/5156/rabsonlib/src/debug.c` show an unnecessarily fancy way of implementing tracers.

**dbx** Standard Unix debugger: available on most platforms but a little non-Unix-like in its syntax. Not available on Linux.

**gdb** Similar to `dbx`, different syntax. Use the `help` command for help. See also my initialization file, `/home/5156/rabsonlib/.gdbinit`.

**ddd** Graphical front end to `gdb`. This works fairly well and is self-explanatory.

**xxgdb** Graphical front end to `gdb`; not as fancy as `ddd`, but works without the Motif library.

**pgdbg** This graphical debugger comes with the Portland-group compilers and may be less buggy than `ddd` and `xxgdb`.

**adb** assembly-level debugger for Real Programmers. It even has an option for examining and modifying variables in a running kernel. (Not available on Linux.)

**gprof** A profiler, not a debugger. It is rarely good practice to write every routine to be as fast as possible. Rather, you will find that most numerical programs spend 90% of their time executing 1% of the code. Compile with `-pg`, then run `gprof` to find the 1% of code whose efficiency matters.
The stack

(schematic: implementation-dependent)

int factorial(int n) {
    if (m <= 0) return 1;
    return (n * factorial(m));
}

int main() {
    static int i;
    i = factorial(10);           .           .           .}

This picture is oversimplified. Automatic variables will often be put into registers and saved on the stack along with return addresses. Also many machines have both a stack and a frame pointer to reduce the overhead of each subroutine. The model, however, is right for the C language, which does not depend on hardware.

storage (definitions boxed; declarations underlined)

```c
int lineno=0;
static char v[]="1.2";
extern double d;
static void g(char *);
char *malloc(unsigned);

func()
{
    char *s=v;
    static int count=0;
    extern int errno;
    s = g(++s);
    printf("%s.%d\n", s, ++count);
}

static void g(char *x)
{
    char *y=malloc(strlen(x));
    y[0]=\'\0\';
    ...
}
```
static char arr[] = "STRING";
char *p = arr;            /* same as &arr[0] */
printf("p=%x=%s\n",p,p);  /* p=1020=STRING */
printf("&p=%x\n",&p);      /* &p=5008 */
printf("&arr[0]=%x\n",&arr[0]);    /* 1020 */
printf("arr[2]=%c\n", arr[2]);    /* R */
printf("p[2]=%c\n", p[2]);        /* R */
printf("%d,%d\n",sizeof(p),sizeof(arr)); /* 4,7 */
Some mistakes in C

```c
/* Get a string from standard input, truncating at the newline.
* Use this instead of gets() because gets() can overflow.
* THIS CODE IS WRONG!!!!!
*/
char *
safe_gets()
{
    char buffer[512];
    char *rvalue=fgets(buffer, sizeof(buffer), stdin);
    return rvalue;
}

/* The string sent to this routine should be of the form <integer> <double>.
* The caller passes variables i and x to stuff with these two numbers.
* We return 0 on success, -1 on trouble.
* THIS CODE IS WRONG!!!!!
*/
int
get_int_and_double(char *input, int i, double x)
{
    if(sscanf(input, "%d %lf", i, x)==2)
        return 0;
    else
        return -1; /* failure */
}

/* Return 1 if the three integers are all equal, 0 otherwise.
* THIS CODE IS WRONG!!!!!
*/
int
equal3(a, b, c) /* old-style function: this isn’t the bug. */
int a, b, c; /* this line optional in old style */
{
    if( (a=b) && (b=c) )
        return 1;
    else
        return 0;
}

/* Increment the integer pointed to by ip.
* THIS CODE IS WRONG!!!!!
*/
void
incr(int *ip)
{
    *ip++;
}
```
1. Histograms and an Introduction to C

Structures

A binary tree

```c
#include <stdlib.h>
#include <stdio.h>
#define ALLOC(x) ((x *)malloc(sizeof(x)))

/* The binary-tree node */
struct node {
  char *string;
  struct node *left;
  struct node *right;
};

/* Copy a string into dynamic storage */
char *copy(char *s) {
  char *rvalue = malloc(strlen(s)+1);
  strcpy(rvalue, s);
  return(rvalue);
}

/* Copy string into dynamic store & return node. */
struct node *newnode(char *string) {
  struct node *rvalue = ALLOC(struct node);
  rvalue->string = copy(string);
  rvalue->right = rvalue->left = 0;
  return rvalue;
}

/* Truncate a string @ first newline, if any. */
char *stripnl(char *s) {
  char c, *rvalue=s; /* original s */
  if(!s) return 0;
  while(c= *s) {
    if(c=='
') {
      *s = 0;
      return rvalue;
    }
    s++;
  }
  return rvalue;
}

/* Find where a string belongs in a binary tree: */
/* return pointer to stuff. */
struct node **
locate(char *s, struct node **rootptr) {
  struct node *n= *rootptr, **r;
  while(n)/* traverse tree */
    n = *(r=(strcmp(n->string,s)>0
      ?&n->left :&n->right));
  return r;
}

/* Add a string to the binary tree. */
void add(char *s, struct node **rootptr) {
  /* make node & find where to put it */
  *locate(s, rootptr) = newnode(s);
}

/* Print the binary tree, recursively. */
void out(struct node *node) {
  if(!node) return; /* no more */
  out(node->left); /* before this */
  printf("%s\n", node->string); /* print this node*/
  out(node->right); /* after this */
}

/* Main routine: take input until done, then print tree */
main() {
  struct node *root=0; /* the root */
  char buf[256]; /* room for input */
  while(fgets(buf,sizeof(buf)-1,stdin)) /* read */
    add(stripnl(buf), &root); /* add */
  out(root); /* print tree */
  return 0;
}
```

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it)
is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.

/* bintree.c */

Sort input lines alphabetically w/ binary tree.

One of the chief advantages of a binary tree (although this program does not exploit it) is the ability to mix additions and deletions yet with the tree completely sorted at all times. For large applications, such a tree should use balancing techniques, which I have not implemented here.