1. Directory Trees. Our object is to print out a directory hierarchy in some pleasant way. The program takes output from `find * -type d -print | sort` and produces a nicer-looking listing. More precisely, our input, which is the output of `find` followed by `sort`, is a list of fully qualified directory names (parent and child separated by slashes `'/'`); everything has already been sorted nicely into lexicographic order.

The `treeprint` routine takes one option, `"-p"`, which tells it to use the printer's line-drawing set, rather than the terminal's.

```
⟨ Global definitions 13 ⟩
⟨ Global include files 6 ⟩
⟨ Global declarations 3 ⟩
⟨ Prototypes 2 ⟩
```

2.

```
⟨ Prototypes 2 ⟩ ≡
  void read_tree(FILE *fp, struct tnode **rootptr);
  void add_tree(struct tnode **rootptr, char *p);
  void print_node(FILE *fp, char *indent_string, struct tnode *node);
  int main(int argc, char **argv)
  {
    ⟨ main variable declarations 4 ⟩;
    ⟨ Search for options and set special characters on "-p" 15 ⟩;
    ⟨ Read output from find and enter into tree 12 ⟩;
    ⟨ Write tree on standard output 19 ⟩
    exit(0);
  }
```

This code is used in section 1.

3. We make all the siblings of a directory a linked list off of its left child, and the offspring a linked list off the right side. Data are just directory names.

```
#define sibling left
#define child right

⟨ Global declarations 3 ⟩ ≡
  typedef struct tnode {
    struct tnode *left, *right;
    char *data;
  } TNODE;
```

See also sections 11, 14, and 16.

This code is used in section 1.

4.

```
⟨ main variable declarations 4 ⟩ ≡
  struct tnode *root = Λ;
```

This code is used in section 2.

5. Input. Reading the tree is simple—we read one line at a time, and call on the recursive *add_tree* procedure.

```
void read_tree(fp, rootptr)
    FILE *fp;
    struct tnode **rootptr;
{
  char buf[255], *p;
  while ((fgets(buf, 255, fp)) ≠ Λ) {
    ⟨If buf contains a newline, make it end there 7⟩;
    add_tree(rootptr, buf);
  }
}
```

6. ⟨Global include files 6⟩ ≡

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

This code is used in section 1.

7. Depending what system you're on, you may or may not get a newline in *buf*.

⟨If *buf* contains a newline, make it end there 7⟩ ≡

```
  p = buf;
  while (*p ≠ '\0' ∧ *p ≠ '\n') p++;
  *p = '\0';
```

This code is used in section 5.

8. To add a string, we split off the first part of the name and insert it into the sibling list. We then do the rest of the string as a child of the new node.

```
void add_tree(rootptr, p)
    struct tnode **rootptr;
    char *p;
{
  char *s;
  int slashed;
  if (*p ≡ '\0') return;
  ⟨Break up the string so p is the first word, s points at null-begun remainder, and slashed tells whether
      *s ≡ '/' on entry 9⟩;
  if (*rootptr ≡ Λ) {
    ⟨Allocate new node to hold string of size strlen(p) 10⟩;
    strcpy((*rootptr)→data, p);
  }
  if (strcmp((*rootptr)→data, p) ≡ 0) {
    if (slashed) ++s;
    add_tree(&((*rootptr)→child), s);
  }
  else {
    if (slashed) *s = '/';
    add_tree(&((*rootptr)→sibling), p);
  }
}
```

9. We perform some nonsense to cut off the string p so that p just holds the first word of a multiword name. Variable s points at what was either the end of p or a slash delimiting names. In either case $*s$ is made `'\0'`. Later, depending on whether we want to pass the whole string or the last piece, we will restore the slash or advance s one character to the right.

⟨ Break up the string so p is the first word, s points at null-begun remainder, and *slashed* tells whether $*s \equiv$ `'/'` on entry 9 ⟩ ≡

```
for (s = p; *s ≠ '\0' ∧ *s ≠ '/'; ) s++;
if (*s ≡ '/') {
  slashed = 1;
  *s = '\0';
}
else slashed = 0;
```

This code is used in section 8.

10. Node allocation is perfectly standard ...

⟨ Allocate new node to hold string of size $strlen(p)$ 10 ⟩ ≡

```
*rootptr = (struct tnode *) malloc(sizeof(struct tnode));
(*rootptr)→left = (*rootptr)→right = Λ;
(*rootptr)→data = malloc(strlen(p) + 1);
```

This code is used in section 8.

11.

⟨ Global declarations 3 ⟩ +≡ /* char *malloc(); */

12. In this simple implementation, we just read from standard input.

⟨ Read output from find and enter into tree 12 ⟩ ≡

```
read_tree(stdin, &root);
```

This code is used in section 2.

13. Output. We begin by defining some lines, tees, and corners. The *s* stands for screen and the *p* for printer. You will have to change this for your line-drawing set.

⟨ Global definitions 13 ⟩ ≡

```
#define svert '|'
#define shoriz '-'
#define scross '+'
#define scorner '\\'   /* lower left corner */
#define pvert '|'
#define phoriz '-'
#define pcross '+'
#define pcorner '\\'   /* lower left corner */
```

This code is used in section 1.

14. The default is to use the terminal's line drawing set.

⟨ Global declarations 3 ⟩ +≡

```
char vert = svert;
char horiz = shoriz;
char cross = scross;
char corner = scorner;
```

15. With option "-p" use the printer character set.

⟨ Search for options and set special characters on "-p" 15 ⟩ ≡

```
while (--argc > 0) {
  if (**++argv ≡ '-') {
    switch (*++(*argv)) {
    case 'p': vert = pvert;
      horiz = phoriz;
      cross = pcross;
      corner = pcorner;
      break;
    default: fprintf(stderr, "treeprint:␣bad␣option␣-%c\n", **argv);
      break;
    }
  }
}
```

This code is used in section 2.

16. We play games with a character stack to figure out when to put in vertical bars. A vertical bar connects every sibling with its successor, but the last sibling in a list is followed by blanks, not by vertical bars. The state of bar-ness or space-ness for each preceding sibling is recorded in the *indent_string* variable, one character (bar or blank) per sibling.

⟨ Global declarations 3 ⟩ +≡

```
char indent_string[100] = "";
```

17. Children get printed before siblings. We don't bother trying to bring children up to the same line as their parents, because the UNIX filenames are so long.

We define a predicate telling us when a sibling is the last in a series.

```
#define is_last(S) (S→sibling ≡ Λ)

  void print_node(fp, indent_string, node)
      FILE *fp;
      char *indent_string;
      struct tnode *node;
  {
    char string[255];
    int i;
    char *p, *is;
    if (node ≡ Λ) { }
    else {
      *string = '\0';
      for (i = strlen(indent_string); i > 0; i−−) strcat(string, "␣|␣␣");
      strcat(string, "␣+--");
      ⟨Replace chars in string with chars from line-drawing set and from indent_string 18⟩;
      fprintf(fp, "%s%s\n", string, node→data);
        /* Add vertical bar or space for this sibling (claim *is ≡ '\0') */
      *is++ = (is_last(node) ? '␣' : vert);
      *is = '\0';
      print_node(fp, indent_string, node→child);     /* extended indent_string */
      *−−is = '\0';
      print_node(fp, indent_string, node→sibling);     /* original indent_string */
    }
  }
```

18. For simplicity, we originally wrote connecting lines with '|', '+', and '-'. Now we replace those characters with appropriate characters from the line-drawing set. We take the early vertical bars and replace them with characters from *indent_string*, and we replace the other characters appropriately. We are sure to put a *corner*, not a *cross*, on the last sibling in a group.

```
⟨Replace chars in string with chars from line-drawing set and from indent_string 18⟩ ≡
  is = indent_string;
  for (p = string; *p ≠ '\0'; p++)
    switch (*p) {
    case '|': *p = *is++;
      break;
    case '+': *p = (is_last(node) ? corner : cross);
      break;
    case '-': *p = horiz;
      break;
    default: break;
    }
```

This code is used in section 17.

19. For this simple implementation, we just write on standard output.

```
⟨Write tree on standard output 19⟩ ≡
  print_node(stdout, indent_string, root);
```

This code is used in section 2.

20. Index.

TREEPRINT