1. Intro. This (hastily written) program computes the twintree that corresponds to a given floorplan specification. See exercises MPR–135 and 7.2.2.1–372 in Volume 4B of *The Art of Computer Programming* for an introduction to the relevant concepts and terminology.

Each room of the floorplan is specified on *stdin* by a line that gives its name, followed by the names of its top bound, bottom bound, left bound, and right bound. For example, the following ten lines specify the example in that exercise:

```
A h0 h3 v0 v1
B h0 h1 v1 v5
C h1 h3 v1 v3
D h3 h5 v0 v2
E h5 h6 v0 v2
F h3 h6 v2 v3
G h1 h2 v3 v5
H h2 h4 v3 v4
I h4 h6 v3 v4
J h2 h6 v4 v5
```

Each name should have at most seven characters (visible ASCII). The rooms can be listed in any order.

The output consists of the corresponding twintrees T_0 and T_1. (Each root is identified, followed by the node names and left/right child links, in symmetric order. A null link is rendered '/\'.)

```
#define bufsize 80   /* maximum length of input lines */
#define maxrooms 1024
#define maxnames (2 * maxrooms + 3)
#define maxjuncs (2 * maxrooms + 3)
#define panic(m, s)
        { fprintf(stderr, "%s! (%s)\n", m, s); exit(-666); }   /* rudely reject bad data */
#define pan(m)
        { fprintf(stderr, "%s!\n", m); exit(-66); }   /* rudely stop on inconsistency */
#define panicic(m, s1, s2)
        { fprintf(stderr, "%s! (%s and %s)\n", m, s1, s2); exit(-666); }
            /* rudely stop with two reasons */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  ⟨Global variables 3⟩;
  ⟨Subroutines 32⟩;
  void main()
  {
    register int i, j, k, l, m, n, q, nameloc, nametyp, rooms, hbounds, vbounds, todo;
    ⟨Input the floorplan 2⟩;
    ⟨Find the junctions 12⟩;
    ⟨Create the twintree 30⟩;
    ⟨Output the twintree 33⟩;
  }
```

2. The input phase. We begin with the easy stuff. Names are remembered in the *name* array, and classified as either rooms or bounds. We store five things for each room, namely the relevant indices *top*[*i*] and *bot*[*i*] which point into *hbound*, the relevant indices *lft*[*i*] and *rt*[*i*] which point into *vbound*, and the index *room*[*i*] of its name.

```
⟨Input the floorplan 2⟩ ≡
  rooms = hbounds = vbounds = 0;
  while (1) {
    if (¬fgets(buf, bufsize, stdin)) break;
    k = 0;
    ⟨Scan the name of room[i] 4⟩;
    ⟨Scan the name of its top bound, top[i] 6⟩;
    ⟨Scan the name of its bottom bound, bot[i] 8⟩;
    ⟨Scan the name of its left bound, lft[i] 9⟩;
    ⟨Scan the name of its right bound, rt[i] 11⟩;
  }
  fprintf(stderr, "(OK,␣I've␣read␣the␣specs␣for␣%d␣rooms,␣%d␣horizontal␣bounds,", rooms,
      hbounds);
  fprintf(stderr, "␣%d␣vertical␣bounds)\n", vbounds);
  if (hbounds + vbounds ≠ rooms + 3) panic("but␣those␣totals␣can't␣be␣right", "not␣h+v=r+3");
```

This code is used in section 1.

3.

```
⟨Global variables 3⟩ ≡
  char buf[bufsize];
  char name[maxnames + 1][8];
  char typ[maxnames];   /* 1 = room, 2 = horiz bound, 3 = vert bound */
  int nameptr;   /* we've seen this many names so far */
  int inx[maxnames];   /* pointer into room or hbound or vbound */
  int room[maxrooms + 1], hbound[maxrooms + 1], vbound[maxrooms + 1];   /* pointers back to names */
  int top[maxrooms], bot[maxrooms], lft[maxrooms], rt[maxrooms];   /* the room's boundaries */
```

See also sections 7, 10, 29, and 31.

This code is used in section 1.

4.

```
⟨Scan the name of room[i] 4⟩ ≡
  ⟨Scan a name 5⟩;
  if (nametyp) panic("duplicate␣room␣name", name[nameloc]);
  i = rooms, room[rooms++] = nameloc;
  typ[nameloc] = 1, inx[nameloc] = i;
```

This code is used in section 2.

5. ⟨Scan a name 5⟩ ≡

```
while (buf[k] ≡ '␣') k++;
if (buf[k] < '␣' ∨ buf[k] > '~') panic("input␣line␣must␣have␣five␣names", buf);
for (j = 0; buf[k] > '␣' ∧ buf[k] ≤ '~'; j++, k++) {
  if (j ≡ 7) panic("name␣longer␣than␣seven␣characters", name[nameptr]);
  name[nameptr][j] = buf[k];
}
name[nameptr][j] = '\0';
for (nameloc = 0; strcmp(name[nameloc], name[nameptr]); nameloc++) ;
if (nameloc < nameptr) nametyp = typ[nameloc];
else {    /* name not seen before */
  nametyp = 0;
  if (++nameptr > maxnames) panic("too␣many␣names", "recompile?");
}
```

This code is used in sections 4, 6, 8, 9, and 11.

6. The jth horizontal bound is named *name*[*hbound*[j]]. It adjoins *tnbrs*[j] rooms above and *bnbrs*[j] rooms below. Those neighbors appear in arrays called *tnbr*[j] and *bnbr*[j].

⟨Scan the name of its top bound, *top*[i] 6⟩ ≡

```
⟨Scan a name 5⟩;
if (¬nametyp) typ[nameloc] = 2, inx[nameloc] = hbounds, hbound[hbounds++] = nameloc;
else if (nametyp ≠ 2) panic("not␣a␣horizontal␣bound", name[nameloc]);
j = top[i] = inx[nameloc];
bnbr[j][bnbrs[j]++] = i;
```

This code is used in section 2.

7. ⟨Global variables 3⟩ +≡

```
int tnbr[maxrooms + 1][maxrooms], bnbr[maxrooms + 1][maxrooms];
int tnbrs[maxrooms + 1], bnbrs[maxrooms + 1];
```

8. ⟨Scan the name of its bottom bound, *bot*[i] 8⟩ ≡

```
⟨Scan a name 5⟩;
if (¬nametyp) typ[nameloc] = 2, inx[nameloc] = hbounds, hbound[hbounds++] = nameloc;
else if (nametyp ≠ 2) panic("not␣a␣horizontal␣bound", name[nameloc]);
j = bot[i] = inx[nameloc];
tnbr[j][tnbrs[j]++] = i;
if (bot[i] ≡ top[i]) panic("room␣of␣zero␣height", name[i]);
```

This code is used in section 2.

9. Similarly, the jth vertical bound is named *name*[*vbound*[j]]. It adjoins *lnbrs*[j] rooms to its left and *rnbrs*[j] rooms to its right. Those neighbors appear in arrays called *lnbr*[j] and *rnbr*[j].

⟨Scan the name of its left bound, *lft*[i] 9⟩ ≡

```
⟨Scan a name 5⟩;
if (¬nametyp) typ[nameloc] = 3, inx[nameloc] = vbounds, vbound[vbounds++] = nameloc;
else if (nametyp ≠ 3) panic("not␣a␣vertical␣bound", name[nameloc]);
j = lft[i] = inx[nameloc];
rnbr[j][rnbrs[j]++] = i;
```

This code is used in section 2.

10. ⟨ Global variables 3 ⟩ +≡
  int $lnbr[maxrooms + 1][maxrooms], rnbr[maxrooms + 1][maxrooms]$;
  int $lnbrs[maxrooms + 1], rnbrs[maxrooms + 1]$;

11. ⟨ Scan the name of its right bound, $rt[i]$ 11 ⟩ ≡
  ⟨ Scan a name 5 ⟩;
  if $(\neg nametyp)$ $typ[nameloc] = 3, inx[nameloc] = vbounds, vbound[vbounds{+}{+}] = nameloc$;
  else if $(nametyp \neq 3)$ $panic($`"not␣a␣vertical␣bound"`$, name[nameloc])$;
  $j = rt[i] = inx[nameloc]$;
  $lnbr[j][lnbrs[j]{+}{+}] = i$;
  if $(lft[i] \equiv rt[i])$ $panic($`"room␣of␣zero␣width"`$, name[i])$;

This code is used in section 2.

12. The setup phase. Now we want to discover the junction points, where a horizontal bound meets a vertical bound. Every horizontal bound runs from a '$\vdash$' junction on its left to a '$\dashv$' junction on its right. (Well, this isn't strictly true for the topmost and bottommost horizontal lines; but we shall treat the floorplan's corners as if they were junctions of two different kinds.)

At each junction point j we'll determine two of the rooms that adjoin it in northeast, southeast, southwest, and northwest directions. Those rooms will be called $ne[j]$, $se[j]$, $sw[j]$, and $nw[j]$, respectively. We set only $nw[j]$ and $ne[j]$ if j is a '$\perp$'; we set only $nw[j]$ and $sw[j]$ if j is a '$\dashv$'; we set only $sw[j]$ and $se[j]$ if j is a '$\top$'; we set only $ne[j]$ and $se[j]$ if j is a '$\vdash$'. The two unset rooms aren't always known, and in any case they're irrelevant.

Empty space surrounding the floorplan is considered to be in a room with the nonexistent number *rooms*. (It shows up only in the four junctions at the extreme corner points.)

The strategy we'll use is quite simple: First we identify the bottom-right corner. Then we work from right to left for every $\dashv$ junction that we know, and from bottom to top for every $\perp$ junction that we know, finding the mates of those junctions as we discover new ones.

Of course many floorplan specifications are actually impossible, or disconnected, etc. We'll want to detect any such anomalies as we go.

⟨ Find the junctions 12 ⟩ ≡
  ⟨ Locate the bottom-right room and bounds 13 ⟩;
  ⟨ Process each bound that's connected to a known junction 17 ⟩;
  ⟨ Make every room point to its corner junctions 28 ⟩;

This code is used in section 1.

13. ⟨ Locate the bottom-right room and bounds 13 ⟩ ≡
  ⟨ Set i to the number of the rightmost vertical bound 14 ⟩;
  ⟨ Set j to the number of the bottom horizontal bound 15 ⟩;
  ⟨ Set l to the number of the bottom-right room 16 ⟩;

This code is used in section 12.

14. ⟨ Set i to the number of the rightmost vertical bound 14 ⟩ ≡

```
for (i = −1, k = 0; k < vbounds; k++)
  if (¬rnbrs[k]) {     /* a vertical with no neighbor on the right */
    if (i ≥ 0) panicic("both␣are␣rightmost", name[vbound[i]], name[vbound[k]]);
    i = k;
  }
if (i < 0) pan("there's␣no␣rightmost␣bound");
```

This code is used in section 13.

15. ⟨ Set j to the number of the bottom horizontal bound 15 ⟩ ≡

```
for (j = −1, k = 0; k < hbounds; k++)
  if (¬bnbrs[k]) {     /* a horizontal with no neighbor below */
    if (j ≥ 0) panicic("both␣are␣at␣the␣bottom", name[hbound[j]], name[hbound[k]]);
    j = k;
  }
if (j < 0) pan("there's␣no␣bottom␣line");
```

This code is used in section 13.

16. ⟨ Set l to the number of the bottom-right room 16 ⟩ ≡

```
for (l = −1, k = 0; k < tnbrs[j]; k++)
  if (rt[tnbr[j][k]] ≡ i) {
    if (l ≥ 0) panicic("both␣are␣at␣bottom-right", name[room[l]], name[room[rt[tnbr[j][k]]]]);
    l = tnbr[j][k];
  }
if (l < 0) pan("there's␣no␣bottom-right␣room");
```

This code is used in section 13.

17. ⟨ Process each bound that's connected to a known junction 17 ⟩ ≡

```
nw[0] = l, ne[0] = sw[0] = rooms;     /* the rooms touching junc[0] */
vjunc[i] = hjunc[j] = 0;
jtyp[0] = #8, vstack[0] = i, hstack[0] = j;
jptr = hptr = vptr = 1;
todo = hbounds + vbounds;
while (hptr + vptr) {
  if (hptr) {
    j = hstack[−−hptr];
    ⟨ Process horizontal bound j 18 ⟩;
    todo−−;
  } else {
    i = vstack[−−vptr];
    ⟨ Process vertical bound i 23 ⟩;
    todo−−;
  }
}
if (todo) pan("disconnected␣floorplan");
```

This code is used in section 12.

18. At this point we know that horizontal bound j has its right end at the ⊣ junction $hjunc[j]$. We want to rearrange its lists of neighbors, and to establish new junctions that we encounter along the way.

⟨ Process horizontal bound j 18 ⟩ ≡

```
⟨ Rearrange the rooms just below bound j 19 ⟩;
⟨ Rearrange the rooms just above bound j 20 ⟩;
⟨ Establish the ⊢ junction at the left of bound j 21 ⟩;
⟨ Launch new ⊥ junctions in bound j 22 ⟩;
```

This code is used in section 17.

19. I use the simplest possible "brute force" approach when rearranging rooms within the neighbor lists. So the rearrangements done here might take quadratic time.

However, if the floorplan specifications are input in the diagonal order of rooms, no rearrangements will be needed, and this entire algorithm will take linear time.

⟨Rearrange the rooms just below bound j 19⟩ ≡

```
l = sw[hjunc[j]];    /* rightmost room below j */
if (l < rooms) {
  for (q = rt[l], i = bnbrs[j] - 1; i; i--) {
    for (k = 0; k ≤ i; k++)
      if (rt[bnbr[j][k]] ≡ q) break;
    if (k > i) panicic("can't␣find␣NE␣room", name[hbound[j]], name[vbound[q]]);
    if (k < i) q = bnbr[j][k], bnbr[j][k] = bnbr[j][i], bnbr[j][i] = q;
    q = lft[bnbr[j][i]];
  }
}
```

This code is used in section 18.

20. ⟨Rearrange the rooms just above bound j 20⟩ ≡

```
l = nw[hjunc[j]];    /* rightmost room above j */
if (l < rooms) {
  for (q = rt[l], i = tnbrs[j] - 1; i; i--) {
    for (k = 0; k ≤ i; k++)
      if (rt[tnbr[j][k]] ≡ q) break;
    if (k > i) panicic("can't␣find␣NW␣room", name[hbound[j]], name[vbound[q]]);
    if (k < i) q = tnbr[j][k], tnbr[j][k] = tnbr[j][i], tnbr[j][i] = q;
    q = lft[tnbr[j][i]];
  }
}
```

This code is used in section 18.

21. Interesting subtleties arise here: We need to launch the vertical bound at the extreme left, if j is the horizontal bound at the very bottom. (This actually happens if and only if $jptr = 1$, because that horizontal bound was placed on the stack first when we began.)

That vertical bound will, similarly, launch the horizontal bound at the extreme top, and it will determine the top left corner (called *tlc*) at that time. When we're processing *that* horizontal bound, we don't want to make another junction at the top left corner.

⟨Establish the ⊢ junction at the left of bound j 21⟩ ≡

```
ne[jptr] = tnbr[j][0], se[jptr] = bnbr[j][0];
if (¬tnbrs[j]) {
  if (se[jptr] ≠ se[tlc]) pan("this␣can't␣happen");
} else if (¬bnbrs[j])
  se[jptr] = nw[jptr] = rooms, q = lft[ne[jptr]], vjunc[q] = jptr, vstack[vptr++] = q, jtyp[jptr++] = #4;
else jtyp[jptr++] = #6;
```

This code is used in section 18.

22. If k rooms are above j, we launch $k-1$ junctions and put the relevant vertical bounds on *vstack*.

⟨ Launch new ⊥ junctions in bound j 22 ⟩ ≡
```
for (k = 1; k < tnbrs[j]; k++) {
  q = lft[tnbr[j][k]], vjunc[q] = jptr, vstack[vptr++] = q;
  nw[jptr] = tnbr[j][k - 1], ne[jptr] = tnbr[j][k], jtyp[jptr] = #c, jptr++;
}
```
This code is used in section 18.

23. Vertical bounds are treated the same, but with dimensions swapped.

⟨ Process vertical bound i 23 ⟩ ≡
  ⟨ Rearrange the rooms just right of bound i 24 ⟩;
  ⟨ Rearrange the rooms just left of bound i 25 ⟩;
  ⟨ Establish the ⊤ junction at the top of bound i 26 ⟩;
  ⟨ Launch new ⊣ junctions in bound i 27 ⟩;

This code is used in section 17.

24. ⟨ Rearrange the rooms just right of bound i 24 ⟩ ≡
```
l = ne[vjunc[i]];   /* lowest room to the right of i */
if (l < rooms) {
  for (q = bot[l], j = rnbrs[i] - 1; j; j--) {
    for (k = 0; k ≤ j; k++)
      if (bot[rnbr[i][k]] ≡ q) break;
    if (k > j) panicic("can't␣find␣SW␣room", name[hbound[q]], name[vbound[i]]);
    if (k < j) q = rnbr[i][k], rnbr[i][k] = rnbr[i][j], rnbr[i][j] = q;
    q = top[rnbr[i][j]];
  }
}
```
This code is used in section 23.

25. ⟨ Rearrange the rooms just left of bound i 25 ⟩ ≡
```
l = nw[vjunc[i]];   /* lowest room to the left of i */
if (l < rooms) {
  for (q = bot[l], j = lnbrs[i] - 1; j; j--) {
    for (k = 0; k ≤ j; k++)
      if (bot[lnbr[i][k]] ≡ q) break;
    if (k > j) panicic("can't␣find␣SE␣room", name[hbound[q]], name[vbound[i]]);
    if (k < j) q = lnbr[i][k], lnbr[i][k] = lnbr[i][j], lnbr[i][j] = q;
    q = top[lnbr[i][j]];
  }
}
```
This code is used in section 23.

26. ⟨ Establish the ⊤ junction at the top of bound i 26 ⟩ ≡
```
sw[jptr] = lnbr[i][0], se[jptr] = rnbr[i][0];
if (¬lnbrs[i]) sw[jptr] = ne[jptr] = rooms, tlc = jptr, jtyp[jptr] = #2;
else if (¬rnbrs[i])
  se[jptr] = nw[jptr] = rooms, q = top[sw[jptr]], hjunc[q] = jptr, hstack[hptr++] = q, jtyp[jptr] = #1;
else jtyp[jptr] = #3;
jptr++;
```
This code is used in section 23.

27. If k rooms are left of i, we launch $k-1$ junctions and put the relevant horizontal bounds on *hstack*.

⟨ Launch new ⊣ junctions in bound i 27 ⟩ ≡
```
for (k = 1; k < lnbrs[i]; k++) {
   q = top[lnbr[i][k]], hjunc[q] = jptr, hstack[hptr++] = q;
   nw[jptr] = lnbr[i][k − 1], sw[jptr] = lnbr[i][k], jtyp[jptr] = #9, jptr++;
}
```
This code is used in section 23.

28. Finally, each junction identifies itself to the rooms that it knows.

⟨ Make every room point to its corner junctions 28 ⟩ ≡
```
for (k = 0; k < jptr; k++) {
   q = jtyp[k];
   if (q & #1) tr[sw[k]] = k;
   if (q & #2) tl[se[k]] = k;
   if (q & #4) bl[ne[k]] = k;
   if (q & #8) br[nw[k]] = k;
}
```
This code is used in section 12.

29. ⟨ Global variables 3 ⟩ +≡
```
int hjunc[maxrooms + 1], vjunc[maxrooms + 1];
int hstack[maxrooms + 1], vstack[maxrooms + 1];
int hptr, vptr;   /* sizes of the stacks */
int junc[maxjuncs];
int jptr;   /* we've seen this many junctions so far */
char jtyp[maxjuncs];   /* #3 = ⊤, #c = ⊥, #6 = ⊢, #9 = ⊣ */
int nw[maxjuncs], ne[maxjuncs], se[maxjuncs], sw[maxjuncs];
int tl[maxrooms], tr[maxrooms], bl[maxrooms], br[maxrooms];
   /* top left, top right, bottom left, and bottom-right junctions */
int tlc;   /* the top-left corner junction */
```

30. The cool phase. Now we're ready to construct the twintree, using a reformulation of the remarkably simple method discovered by Bo Yao, Hongyu Chen, Chung-Kuan Cheng, and Ronald Graham in *ACM Transactions on Design Automation of Electronic Systems* **8** (2003), 55–80.

From this construction we see that many of the arrays above are superfluous, and we needn't have bothered to compute them!

```
⟨ Create the twintree 30 ⟩ ≡
  null = rooms;
  for (k = 0; k < rooms; k++) {
    j = tl[k];
    if (jtyp[j] ≡ #3) l0[k] = null, l1[k] = sw[j];
    else l0[k] = ne[j], l1[k] = null;
    j = br[k];
    if (jtyp[j] ≡ #9) r0[k] = null, r1[k] = sw[j];
    else r0[k] = ne[j], r1[k] = null;
  }
  root0 = ne[1], root1 = sw[tlc + 1];
```

This code is used in section 1.

31. ⟨ Global variables 3 ⟩ +≡

```
  int root0, l0[maxrooms], r0[maxrooms], root1, l1[maxrooms], r1[maxrooms];
  int null;     /* the null room */
```

32. The output phase.

```
#define rjustname(k) (int)(8 - strlen(name[room[k]])), "", name[room[k]]

⟨ Subroutines 32 ⟩ ≡
  void inorder0(int root)
  {
    if (l0[root] ≠ null) inorder0(l0[root]);
    printf("%*s%s:␣%*s%s,␣%*s%s\n", rjustname(root), rjustname(l0[root]), rjustname(r0[root]));
    if (r0[root] ≠ null) inorder0(r0[root]);
  }
  void inorder1(int root)
  {
    if (l1[root] ≠ null) inorder1(l1[root]);
    printf("%*s%s:␣%*s%s,␣%*s%s\n", rjustname(root), rjustname(l1[root]), rjustname(r1[root]));
    if (r1[root] ≠ null) inorder1(r1[root]);
  }
```

This code is used in section 1.

33. ⟨ Output the twintree 33 ⟩ ≡

```
  room[rooms] = nameptr;
  strcpy(name[nameptr], "/\\");
  printf("T0␣(rooted␣at␣%s)\n", name[room[root0]]);
  inorder0(root0);
  printf("T1␣(rooted␣at␣%s)\n", name[room[root1]]);
  inorder1(root1);
```

This code is used in section 1.

34. Index.

FLOORPLAN-TO-TWINTREE