

**1. Intro.** Here's an implementation of the Panholzer–Prodinger algorithm, which generates a uniformly random “decorated” ternary tree. (It generalizes the binary method of Rémy, Algorithm 7.2.1.6R, and solves exercise 7.2.1.6–65.) They presented it in *Discrete Mathematics* **250** (2002), 181–195, but without spelling out an efficient implementation.

Although the algorithm is short, it is not easy to discover; the reader who thinks otherwise is invited to invent it before reading further.

I'm using a linked structure as in the presentation of in Rémy's method in Volume 4A: There are  $3n + 1$  links  $L_0, L_1, \dots, L_{3n}$ , which are a permutation of the integers  $\{0, 1, \dots, 3n\}$ . Internal (branch) nodes have numbers congruent to 2 (mod 3). The root is node number  $L_0$ ; the descendants of branch  $3k - 1$  are the nodes numbered  $L_{3k-2}, L_{3k-1}, L_{3k}$ . For example, if  $n = 3$  and  $(L_0, L_1, \dots, L_9) = (5, 0, 1, 3, 2, 6, 7, 8, 4, 9)$ , the root is node 5 (a branch node); its left child is node 2 (another branch), its middle child is node 6 (external), and its right child is node 8 (yet another branch).

I also maintain the inverse permutation  $(P_0, \dots, P_{3n})$ , so that we can determine the parent of each node.

```
#define nmax 1000
#include <stdio.h>
#include <stdlib.h>
#include "gb_flip.h" /* random number generator from the Stanford GraphBase */
int nn, seed; /* command-line parameters */
int L[nmax], P[nmax]; /* the links and their inverses */
main(int argc, char *argv[])
{
    register int i, j, k, n, nnn, p, q, r, x;
    <Process the command line 2>;
    for (n = L[0] = P[0] = 0; n < nn; )
        <Extend a solution for n to a solution for n + 1, and increase n 3>;
    for (k = 0; k ≤ 3 * nn; k++) printf("%d", L[k]);
    printf("\n");
}

2. <Process the command line 2> ≡
if (argc ≠ 3 ∨ sscanf(argv[1], "%d", &nn) ≠ 1 ∨ sscanf(argv[2], "%d", &seed) ≠ 1) {
    fprintf(stderr, "Usage: %s %d %d\n", argv[0]);
    exit(-1);
}
gb_init_rand(seed);
```

This code is used in section 1.

**3. #define sanity\_checking 1**

```

⟨Extend a solution for  $n$  to a solution for  $n + 1$ , and increase  $n$  3⟩ ≡
{
    n++;
    nnn = 3 * n;
    x = gb_unif_rand(3 * (nnn - 1) * (nnn - 2));
    p = nnn - (x % 3);
    q = nnn - ((x + 1) % 3);
    r = nnn - ((x + 2) % 3);
    k = ((int)(x/3)) % (nnn - 2);
    j = (int)(x/(9 * n - 6));
    L[p] = nnn, P[nnn] = p;
    L[q] = L[k], P[L[k]] = q, L[k] = nnn - 1, P[nnn - 1] = k;
    ⟨Do the magic switcheroo 5⟩;
    if (sanity_checking) {
        for (i = 0; i ≤ nnn; i++)
            if (P[L[i]] ≠ i) {
                fprintf(stderr, "(whoa---the links are fouled up!)\n");
                exit(-2);
            }
    }
}

```

This code is used in section 1.

4. Variables  $j$  and  $L_k$  correspond to P-and-P's nodes  $y$  and  $x$ ; they are random integers with  $0 \leq j \leq 3n+1$  and  $0 \leq k \leq 3n$ . The basic idea is to insert a new branch node (node number  $3n - 1$ ) in place of node  $L_k$ ; but this new node has the old node  $L_k$  as one of its children (pointed to by  $L_q$ ), so we haven't really lost anything. Another child, pointed to by  $L_p$ , is the leaf  $3n$ . The third child, pointed to by  $L_r$ , has to somehow encode the fact that we also need to place the leaf  $3n - 2$  while maintaining randomness.

There are two main cases, depending on whether node number  $y$  is a proper ancestor of node number  $x$ .

The crucial point, proved in the paper, is that we can recover  $x$ ,  $y$ , and  $p$  by looking at the switched links.

```

5. ⟨Do the magic switcheroo 5⟩ ≡
for (i = k + 1 - ((k + 2) % 3); i > 0 ∧ i ≠ j; i = P[i] + 1 - ((P[i] + 2) % 3)) ;
if (i > 0) ⟨Do the harder case 6⟩
else {
    if (j ≡ L[q]) { /* y = x */
        L[r] = L[q], P[L[q]] = r, L[q] = nnn - 2, P[nnn - 2] = q;
    } else if (j ≡ nnn - 2) { /* y is the special leaf */
        L[r] = nnn - 2, P[nnn - 2] = r;
    } else L[P[j]] = nnn - 2, P[nnn - 2] = P[j], L[r] = j, P[j] = r;
}

```

This code is used in section 3.

6. The “harder case” isn't really hard for the computer; it's just harder for a human being to visualize.

```

⟨Do the harder case 6⟩ ≡
{
    L[k] = nnn - 2, P[nnn - 2] = k;
    i = P[j]; /* the link to y */
    L[i] = nnn - 1, P[nnn - 1] = i, L[r] = j, P[j] = r;
}

```

This code is used in section 5.

**7. Index.**

*argc*: [1](#), [2](#).  
*argv*: [1](#), [2](#).  
*exit*: [2](#), [3](#).  
*fprintf*: [2](#), [3](#).  
*gb\_init\_rand*: [2](#).  
*gb\_unif\_rand*: [3](#).  
*i*: [1](#).  
*j*: [1](#).  
*k*: [1](#).  
*L*: [1](#).  
*main*: [1](#).  
*n*: [1](#).  
*nmax*: [1](#).  
*nn*: [1](#), [2](#).  
*nnn*: [1](#), [3](#), [5](#), [6](#).  
*P*: [1](#).  
*p*: [1](#).  
*printf*: [1](#).  
*q*: [1](#).  
*r*: [1](#).  
*sanity\_checking*: [3](#).  
*seed*: [1](#), [2](#).  
*sscanf*: [2](#).  
*stderr*: [2](#), [3](#).  
*x*: [1](#).

⟨ Do the harder case 6 ⟩ Used in section 5.

⟨ Do the magic switcheroo 5 ⟩ Used in section 3.

⟨ Extend a solution for  $n$  to a solution for  $n + 1$ , and increase  $n$  3 ⟩ Used in section 1.

⟨ Process the command line 2 ⟩ Used in section 1.

RANDOM-TERNARY

	Section	Page
Intro .....	<a href="#">1</a>	1
Index .....	<a href="#">7</a>	3