1. Intro. On 07 June 2024, Jim Propp told me about an interesting bijective mapping on (ordered) trees that have more than node: "The rightmost child of the old root becomes the new root, and the old root becomes its leftmost child." (If the old children are $c_1, \ldots, c_k$, the new children are the children of c_k preceded by a subtree whose children are $c_1, \ldots, c_{k-1}$.) This mapping preserves the order of leaves. The main point is that *every node previously on an even level is now on an odd level, and vice versa.*

While playing with this transformation, I noticed that the number of trees that have m nodes on odd levels and n nodes on even levels, for $m, n > 0$, is the Narayana number

$$T(m,n) = \frac{(m+n-1)!\,(m+n-2)!}{m!\,(m-1)!\,n!\,(n-1)!},$$

which is well known to be the number of binary trees that have m null left links and n null right links. (The tree has $m+n$ nodes; the binary tree has $m+n-1$ nodes, $n-1$ nonnull left links, and $m-1$ nonnull right links. See, for example, exercise 2.3.4.6–3 in *The Art of Computer Programming.*)

So I looked for a bijection between such trees and such binary trees.

This program implements the bijection that I came up with. In a sense, it's a sequel to my "Three Catalan bijections," *Institut Mittag-Leffler Reports*, No. 04, 2004/2005, Spring (2005), 19 pp.

Unfortunately, I don't have time to provide extensive comments. Let the code speak for itself.

```
#define nodes 17   /* nodes in the tree; must be at least 2 */
#define vbose 0   /* set this nonzero to see details */
#include <stdio.h>
  int llink[nodes + 1], rlink[nodes];   /* links of the binary tree */
  int lchild[nodes + 1], rsib[nodes + 1];   /* leftmost child and right sibling in the tree */
  int count[nodes];
  ⟨Subroutines 3⟩;
  main()
  {
    register j, k, y;
    printf("Checking␣all␣trees␣with␣%d␣nodes...\n", nodes);
    ⟨Initialize Skarbek's algorithm 4⟩;
    while (1) {
      ⟨Find the tree (lchild, rsib) that corresponds to the binary tree (llink, rlink) 7⟩;
      if (vbose) ⟨Print the trees 2⟩;
      ⟨Check the null link counts and the level parity counts 9⟩;
      ⟨Move to the next binary tree (llink, rlink), or break 5⟩;
    }
    for (k = 1; count[k]; k++) printf("Altogether␣%d␣case%s␣with␣%d␣node%s␣at␣odd␣levels.\n",
          count[k], count[k] ≡ 1 ? "" : "s", k, k ≡ 1 ? "" : "s");
  }
```

2. ⟨Print the trees 2⟩ ≡

```
  {
    print_binary_tree();
    printf("␣->␣");
    print_tree();
    printf("\n");
  }
```

This code is used in sections 1 and 9.

3. **#define** *encode*(x) $((x) < 10 ? \texttt{'0'} + (x) : \texttt{'a'} + (x) - 10)$

⟨ Subroutines 3 ⟩ ≡

```
void print_binary_tree(void)
{
  register int k;
  for (k = 1; k < nodes; k++) printf("%c", encode(llink[k]));
  printf("|");
  for (k = 1; k < nodes; k++) printf("%c", encode(rlink[k]));
}
void print_tree(void)
{
  register int k;
  for (k = 1; k ≤ nodes; k++) printf("%c", encode(lchild[k]));
  printf("|");
  for (k = 1; k ≤ nodes; k++) printf("%c", encode(rsib[k]));
}
```

See also sections 6 and 8.

This code is used in section 1.

4. Skarbek's elegant algorithm (Algorithm 7.2.1.6B in *The Art of Computer Programming*, Volume 4A) is used to run through all linked binary trees with $nodes - 1$ nodes.

⟨ Initialize Skarbek's algorithm 4 ⟩ ≡

```
for (k = 1; k < nodes − 1; k++) llink[k] = k + 1, rlink[k] = 0;
llink[nodes − 1] = rlink[nodes − 1] = 0;
llink[nodes] = 1;
```

This code is used in section 1.

5. ⟨ Move to the next binary tree (*llink*, *rlink*), or **break** 5 ⟩ ≡

```
for (j = 1; ¬llink[j]; j++) rlink[j] = 0, llink[j] = j + 1;
if (j ≡ nodes) break;
for (k = 0, y = llink[j]; rlink[y]; k = y, y = rlink[y]) ;
if (k) rlink[k] = 0; else llink[j] = 0;
rlink[y] = rlink[j], rlink[j] = y;
```

This code is used in section 1.

6. The bijection is implemented by a recursive procedure, which has three parameters: p is the index of the first node not already created; r is the root of the binary tree to be converted to a tree; *parity* is 1 if we are interchanging *llink* with *rlink*.

This procedure returns the index of the root node of the constructed tree.

⟨ Subroutines 3 ⟩ +≡

```
int propp(int p, int r, int parity)
{
  register lam, rho;
  if (r ≡ 0) {
    lchild[p] = rsib[p] = 0;
    return p;
  }
  if (parity ≡ 0) {
    lam = propp(p, llink[r], 1);
    rho = propp(lam + 1, rlink[r], 0);
  } else {
    lam = propp(p, rlink[r], 0);
    rho = propp(lam + 1, llink[r], 1);
  }
  rsib[lam] = lchild[rho], lchild[rho] = lam;
  return rho;    /* note that rsib[rho] = 0 */
}
```

7. ⟨ Find the tree (*lchild*, *rsib*) that corresponds to the binary tree (*llink*, *rlink*) 7 ⟩ ≡

```
if (propp(1, 1, 0) ≠ nodes) fprintf(stderr, "I'm␣confused!\n");
```

This code is used in section 1.

8. The *lcount* routine determines how many nodes of a given nonempty tree, rooted at r, are at a level with a given parity. (The root is at level zero.)

⟨ Subroutines 3 ⟩ +≡

```
int lcount(int r, int parity)
{
  register int c, p;
  for (c = 1 − parity, p = lchild[r]; p; p = rsib[p]) c += lcount(p, 1 − parity);
  return c;
}
```

9. ⟨ Check the null link counts and the level parity counts 9 ⟩ ≡

```
for (j = 0, k = 1; k < nodes; k++)
  if (llink[k] ≡ 0) j++;
if (j ≠ lcount(nodes, 1)) {
  printf("Mismatch!␣");
  ⟨ Print the trees 2 ⟩;
}
count[j]++;
```

This code is used in section 1.

10. Index.

PROPP