```
            new_arr[i] ^= new_arr[j];
            new_arr[j] ^= new_arr[i];
            new_arr[i] ^= new_arr[j];
        }
    }
    return new_arr;
}

static key_t*
arr_gen_conj_d(key_t size) {
    key_t* new_arr = (key_t*) malloc( sizeof(key_t) * size);

    if (new_arr) {
        new_arr[0] = 0; // não podes saltar um item atrás de idx 0
        for (key_t i = 1; i < size; i++) {
              new_arr[i] = (randint(0,9) == 3) ? i : new_arr[i-1];
        }

        /* Knuth Shuffle */
        int i, j;
        for (j = size-1; j > 0; j--) {
            i = randint(0, j-1);
            new_arr[i] ^= new_arr[j];
            new_arr[j] ^= new_arr[i];
            new_arr[i] ^= new_arr[j];
        }

    }

    return new_arr;
}

static void
arr_print(key_t* arr, key_t size) {
    for (key_t k = 0; k < size; k++)
        printf("arr[%d] = %d\n", k, arr[k]);
}


BinTree
tree_binary_create(uint32_t initial_capacity) {
    BinTree btree = {initial_capacity, 0, NULL};
    btree.root = (BinTreeNode*) malloc(sizeof(BinTreeNode)*initial_capacity);

    if (btree.root) {
        BinTreeNode* nodeptr = btree.root;
        for (BinTreeNode* endptr = nodeptr + initial_capacity; nodeptr != endptr; nodeptr++ ) {
            nodeptr->data = 0;
            nodeptr->idx_left = 0;
            nodeptr->idx_right = 0;
        }
    }

    return btree;
}

void
tree_binary_destroy(BinTree btree) {
    free(btree.root);
}
```