```
      /* manter max heap */
      if (nodes[nodes[idx].right].priority > nodes[idx].priority) {
         idx = _treap_rotate_left(treap, idx);
      }
   }

   return idx;
}

/* inserir nó */
void
tree_treap_insert(Treap *treap, key_t key) {
   if (treap->capacity == treap->elements)
      tree_treap_resize(treap);

   treap->tree_root = _treap_insert_recursive(treap, treap->tree_root, key);
}

void
tree_treap_visualize(Treap *treap, idx_t root, int depth, const char *prefix, int is_left) {
   if (root == IDX_INVALID) return;

   TreapNode *node = &treap->nodes[root];

   // Print current node
   printf("%s", prefix);
   printf("%s", (depth == 0) ? "" : (is_left ? " ├── " : " └── "));
   printf("(%d, p=%u)\n", node->key, node->priority);

   // Prepare prefix for child nodes
   char new_prefix[256];
   snprintf(new_prefix, sizeof(new_prefix), "%s%s", prefix, (depth == 0) ? "" : (is_left ? " │   " : "    "));

   // Right child (printed first for better tree shape)
   tree_treap_visualize(treap, node->right, depth + 1, new_prefix, 1);

   // Left child
   tree_treap_visualize(treap, node->left, depth + 1, new_prefix, 0);
}

void
treap_test_and_log(key_t* arr, FILE *fptr) {

   Treap treap;
   clock_t start = 0, end = 0;
   clock_t total = 0;

   /* Reset global rotation counter */
   g_rotation_count = 0;


   for (int i = 0; i < 1; i++) {
      start = clock();
      treap = tree_treap_create(g_treesize);
      for (idx_t idx = 0; idx < g_treesize; idx++)
         tree_treap_insert(&treap, arr[idx]);
      end = clock();

      total += (end-start);
      tree_treap_destroy(&treap);
   }
```