Relatório Projeto 2 V1.0 AED 2024/2025

Nome: Vasco Guilherme da Silva Jacinto Gaspar Alves Nº Estudante: 202222 8207
PL (inscrição): PL-6 Email: vasco.guilherme.alves@gmail.com

IMPORTANTE:

- **Os textos das conclusões devem ser manuscritos...** o texto deve obedecer a este requisito para não ser penalizado.
- Texto para além das linhas reservadas, ou que não seja legível para um leitor comum, não será tido em conta.
- **O relatório deve ser submetido num único PDF** que deve incluir os anexos. A não observância deste formato é penalizada.

1. Planeamento

	Semana 1	**Semana 2**	**Semana 3**	**Semana 4**	**Semana 5**
Árvore Binária					
Árvore AVL					
Árvore VP					
Árvore TREAP					
Finalização Relatório					

2. Recolha de Resultados *(tabelas)*

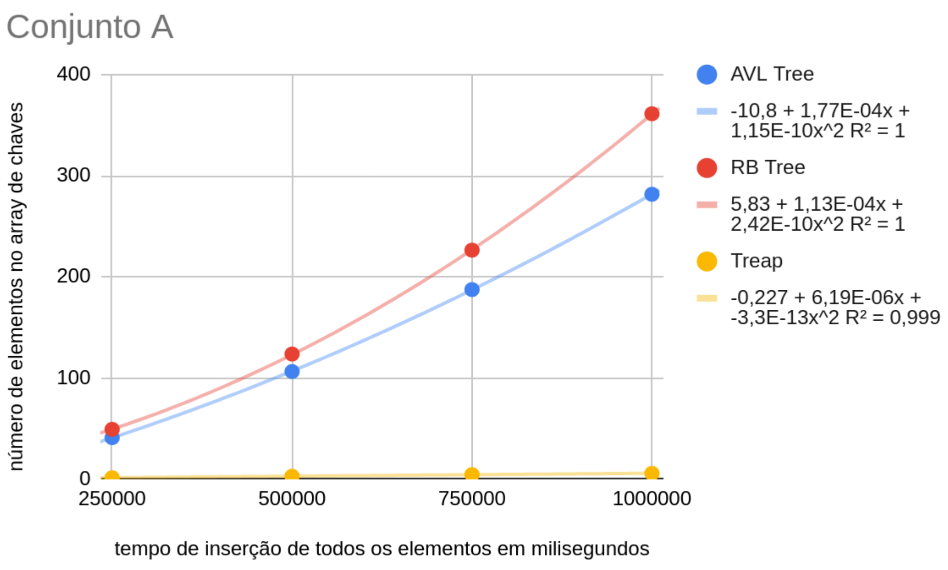
Conjunto A								
	Binary Tree		AVL Tree		RB Tree		Treap	
keys = N	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações
250000	8342,166	0	40,8736	224988	49,0773	224989	1,328	22499
500000	33115,158	0	106,3296	450115	123,523	450116	2,7023	45011
750000	79265,0047	0	187,4332	675117	226,2721	675118	4,3136	67512
1000000	166340,049	0	281,7198	900038	361,1859	900039	5,6055	90004

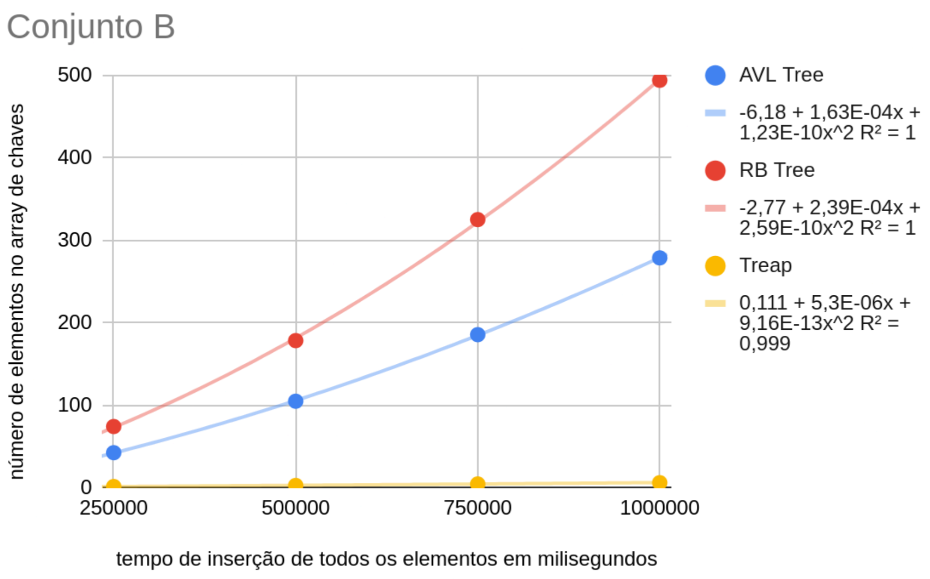
Conjunto B								
	Binary Tree		AVL Tree		RB Tree		Treap	
keys = N	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações
250000	8371,9533	0	42,4671	337675	74,2349	450229	1,5236	44802
500000	32117,1306	0	104,9327	674866	178,4558	899819	2,8972	76505
750000	78140,9478	0	185,7806	1012481	325,2855	1349973	4,6919	116310
1000000	163106,4253	0	278,9249	1350254	494,2021	1800336	6,2946	168312

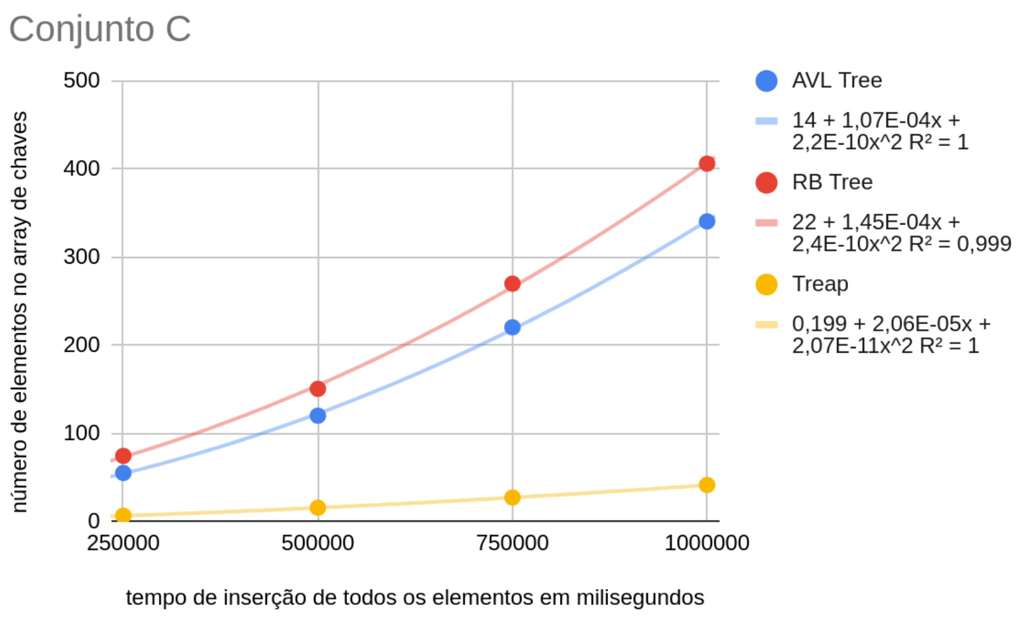
Conjunto C								
	Binary Tree		AVL Tree		RB Tree		Treap	
keys = N	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações
250000	8997,0735	0	55,2627	157031	74,6654	266380	6,6315	45009
500000	34528,6301	0	120,2459	314627	150,7121	534194	15,6595	90081
750000	84910,7065	0	220,476	472301	270,0461	802722	27,2432	135226
1000000	168333,5287	0	340,5817	629331	406,143	1067687	41,4357	180144

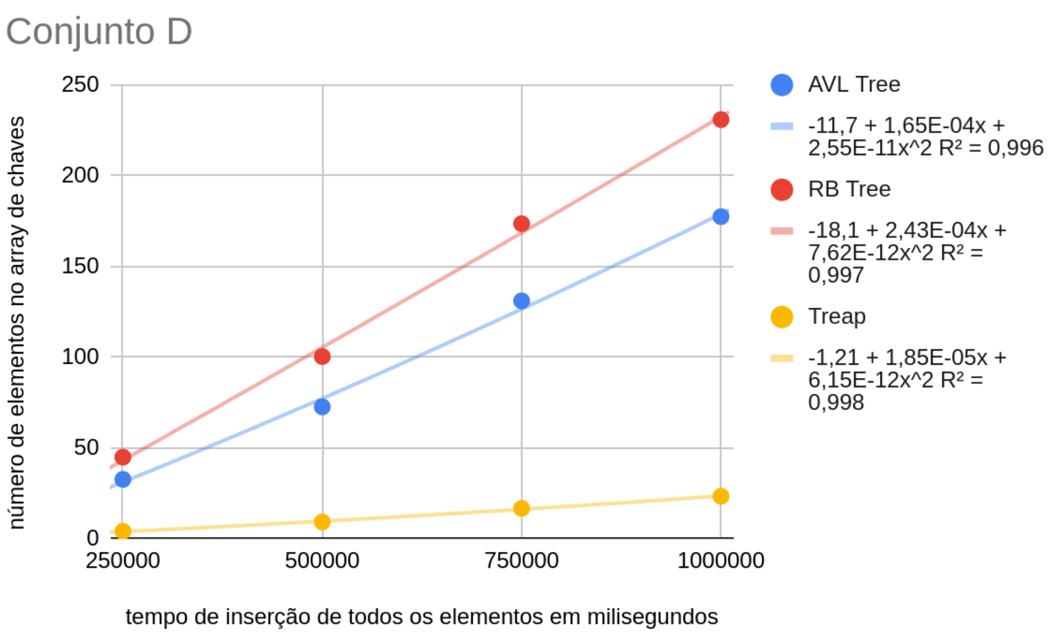
Conjunto D								
	Binary Tree		AVL Tree		RB Tree		Treap	
keys = N	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações	Inserir (ms)	Rotações
250000	891,4834	0	32,668	17215	44,832	29251	3,9513	4945
500000	3458,5128	0	72,6603	34905	100,2848	59198	9,1167	10001
750000	8052,4947	0	131,0583	52377	173,6042	89013	16,5911	15100
1000000	13778,8157	0	177,4236	69678	230,9611	118171	23,2947	20009

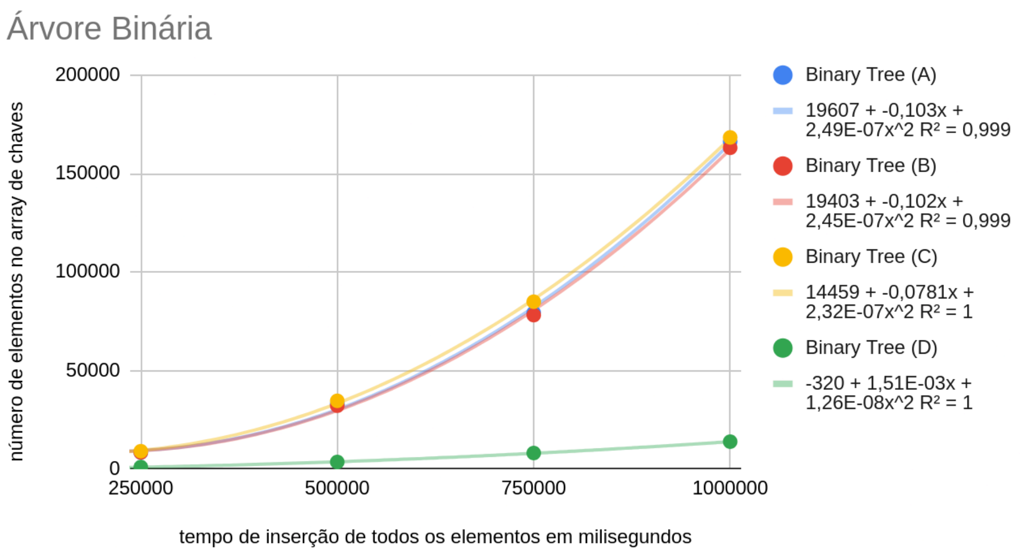
3. Visualização de Resultados *(gráficos)*








Conjunto D
250
200
150
100
50
0
250000
500000
750000
1000000
número de elementos no array de chaves
tempo de inserção de todos os elementos em milisegundos
AVL Tree
-11,7 + 1,65E-04x + 2,55E-11x^2 R² = 0,996
RB Tree
-18,1 + 2,43E-04x + 7,62E-12x^2 R² = 0,997
Treap
-1,21 + 1,85E-05x + 6,15E-12x^2 R² = 0,998


Árvore Binária
200000
150000
100000
50000
0
250000
500000
750000
1000000
número de elementos no array de chaves
tempo de inserção de todos os elementos em milisegundos
Binary Tree (A)
19607 + -0,103x + 2,49E-07x^2 R² = 0,999
Binary Tree (B)
19403 + -0,102x + 2,45E-07x^2 R² = 0,999
Binary Tree (C)
14459 + -0,0781x + 2,32E-07x^2 R² = 1
Binary Tree (D)
-320 + 1,51E-03x + 1,26E-08x^2 R² = 1

4. Conclusões *(as linhas no template representam a extensão máxima de texto manuscrito)*

4.1 Tarefa 1

A árvore binária tem a maior complexidade para as operações de pesquisa e inserção. No pior caso, inserir requer pesquisar todos os elementos. Sendo assim, inserir N chaves tem uma complexidade de $O(n \cdot (n+1)/2)$ (Soma de Gauss), ou aproximadamente $O(n^2)$. A regressão feita no grafo correspondente suporta esta conclusão.

Por outro lado, a natureza implícita desta estrutura e a falta de ordenação permite remover elementos em $O(1)$ e N elementos em $O(N)$.

4.2 Tarefa 2

A árvore AVL tem o melhor balanceamento por natureza, e logo deveria realizar ~~rotações~~ um maior número de rotações que as restantes. Para conjuntos pequenos de keys este overhead pode não valer a pena, mas para conjuntos maiores o balanceamento mais estricto resulta em operações de pesquisa mais eficientes.

A AVL mantém uma complexidade tempo de $O(\log_2 N)$ consistente para as suas pesquisas. ~~Pa~~

4.3 Tarefa 3

A árvore Red-Black é menos estritamente balanceada que a AVL, o que é benéfico para conjuntos com tamanhos mais pequenos e tamanhos médios. O menor número de rotações significa que é mais rápido inserir em média, mas à medida que o balanceamento vai diminuindo, esta vantagem pode desaparecer, o que aparenta ser o caso nas ~~[illegible]~~ medidas tomadas.

A operação de pesquisa ~~[illegible]~~ pode perder eficiência quando o balanço diminuir.

4.4 Tarefa 4

A Treap teve o melhor desempenho de todas as árvores. A sua natureza aleatória permite inserir elementos com um número mínimo de rotações. O balanceamento é o menos restrito de todas as árvores logo as pesquisas vão ser mais demoradas em média do que nas árvores AVL e Red Black.

Noto também que para o conjuntos A e B, os quais são ordenados, o desempenho é significativamente melhor que para os conjuntos aleatórios C e D.

Anexo B - Código de Autor

```
/* Código feito para C99, compilado com GCC 14.2.1 com flags --std=c99 -O2 e --fast-math
 *
 * Hardware Original: TOSHIBA SATELLITE_C50-A PSCG6P-01YAR1,
 * CPU: Intel i5-3320M (4) @ 3.300GHz,
 * GPU: Intel 3rd Gen Core processor Graphics Controller
 * RAM: 7821MiB, SSD SATA3 1TB
 *
 * Aluno: Vasco Alves, 2022228207
*/

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>
#include <time.h>

#define RESIZE_FACTOR 1.61803

#define IDX_INVALID 4294967295

#define SEED 95911405

#define BLACK 0
#define RED 1

typedef uint32_t idx_t;
typedef int32_t key_t;

static int32_t g_treesize;
static int32_t g_average;
static uint32_t g_rotation_count = 0;

typedef struct BinTreeNode {
    key_t data;
    idx_t idx_left;
    idx_t idx_right;
} BinTreeNode; // TOTAL = 12 Bytes per node;

typedef struct BinaryTree {
    uint32_t capacity;
    uint32_t elements;
    BinTreeNode *root;
} BinTree ;

typedef struct AVLNode {
    idx_t left;
    idx_t right;
    int key;
    int height;
} AVLNode;

typedef struct AVLTree {
    AVLNode *nodes;
    idx_t tree_root; // rotations cause the root to change
    idx_t elements;
    idx_t capacity;
} AVLTree; // 20 bytes

typedef struct RBNode {
    idx_t left;     // 4 bytes
    idx_t right;    // 4 bytes
    key_t key;      // 4 bytes
    int8_t color;   // 1 bytes
} RBNode;

typedef struct RBTree {
    RBNode *nodes;
    idx_t tree_root;
    idx_t elements;
    idx_t capacity;
} RBTree;

typedef struct TreapNode {
    key_t key;
    idx_t priority;
    idx_t left;
    idx_t right;
} TreapNode;

typedef struct Treap {
    TreapNode* nodes;
    idx_t tree_root;
    idx_t elements;
    idx_t capacity;
} Treap;

/* === HELPER FUNCTIONS === */
static inline int randint(int a, int b);
static inline idx_t rand_idx(idx_t a, idx_t b);
static inline int max(int a, int b);
static key_t*   arr_gen_conj_a(const key_t size); // ordem crescent, pouca repetição
static key_t*   arr_gen_conj_b(const key_t size); // ordem decrescent, pouca repetição
static key_t*   arr_gen_conj_c(const key_t size); // ordem aleatoria, pouca repetição
static key_t*   arr_gen_conj_d(const key_t size); // ordem aleatoria, 90% repetidos
static void     arr_print(key_t* arr, key_t size);

/* ===== BINARY TREE ===== */
extern BinTree  tree_binary_create(uint32_t initial_capacity); // Creates binary tree with inicialized
elements
extern void     tree_binary_destroy(BinTree btree); // Frees binary tree
extern void     tree_binary_resize(BinTree *btree);  // Resize binary tree
extern void     tree_binary_insert(BinTree *btree, key_t key); // insert key in binary tree, NO
DUPLICATES
extern void     tree_binary_insert_arr(BinTree *btree, key_t* arr, key_t size); // insert array of keys
extern void     tree_binary_print_inorder(BinTree *btree); // in order print according to tree
extern void     tree_binary_print(BinTree *btree);  // print by levels for visual accuracy
extern idx_t    tree_binary_search_key_inorder(BinTree btree, int32_t key); // search for key in binary
tree by order
extern idx_t    tree_binary_search_key_level(BinTree btree, int32_t key); // faster than inorder
because of this structure
extern void     binary_test_and_log(key_t* arr, FILE *fptr);

/* ===== AVL TREE ===== */
extern AVLTree tree_avl_create(idx_t inicial_capacity);
extern void    tree_avl_destroy(AVLTree* avl);
extern void    tree_avl_resize(AVLTree *avl);
static int     _avl_get_height(AVLTree* avl, idx_t index);
static int     _avl_get_balance(AVLTree* avl, idx_t index);
static idx_t   _avl_rotate_right(AVLTree *avl, idx_t y_index);
static idx_t   _avl_rotate_left(AVLTree *avl, idx_t x_index);
static idx_t   _avl_insert_recursive(AVLTree *avl, idx_t node_index, int key);
extern void     tree_avl_insert(AVLTree *avl, int key);
extern void     tree_avl_insert_arr(AVLTree *avl, key_t* arr, size_t size);
extern AVLNode* tree_avl_search(AVLTree *avl, int key);
extern void     tree_avl_in_order(AVLTree *avl); // in-order print

/* ===== RED BLACK TREE ===== */
extern RBTree  tree_rb_create(uint32_t initial_capacity);
extern void    tree_rb_destroy(RBTree *rb);
extern void    tree_rb_resize(RBTree *rb);
static int     _rb_is_red(RBTree *tree, idx_t i);
static idx_t   _rb_rotate_left(RBTree *tree, idx_t h);
static idx_t   _rb_rotate_right(RBTree *tree, idx_t h);
static void    _rb_flip_colors(RBTree *tree, idx_t h);
static idx_t   _rb_fix_up(RBTree *tree, idx_t h);
static idx_t   _rb_insert_recursive(RBTree *tree, idx_t h, key_t key);
extern void    tree_rb_insert(RBTree *tree, key_t key);
extern int     tree_rb_search(RBTree *rb, int key);

/* ===== TREAP ===== */
extern Treap tree_treap_create(idx_t initial_capacity);
extern void  tree_treap_resize(Treap *treap);
extern void  tree_treap_destroy(Treap *treap);
static idx_t _treap_rotate_right(Treap *treap, idx_t x_idx);
static idx_t _treap_rotate_left(Treap *treap, idx_t x_idx);
static idx_t _treap_insert_recursive(Treap *treap, idx_t idx, key_t key);
extern void  tree_treap_insert(Treap *treap, key_t key);

/* ==== FUNCTION DECLATRATIONS ==== */
static inline int
randint(int a, int b) {
  if (a > b) {
    a ^= b;
    b ^= a ;
    a ^= b;
    }
    return a + rand() % (b - a + 1);
}

static idx_t rng_state = SEED;

static inline idx_t
rand_idx(idx_t a, idx_t b) {
    rng_state ^= rng_state << 13;
    rng_state ^= rng_state >> 17;
    rng_state ^= rng_state << 5;
    return a + rng_state % (b - a + 1);
}

static inline int
max(int a, int b) {
    return (a > b) ? a : b;
}

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

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

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

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

    return new_arr;
}

static key_t*
arr_gen_conj_c(const key_t size) {

    /* Array crescente com repetição minima */
    key_t* new_arr = arr_gen_conj_a(size);

    if (new_arr) {
        /* 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 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);
}
```

```
void
tree_binary_resize(BinTree *btree) {
   uint32_t new_capacity = btree->capacity*RESIZE_FACTOR;
   /* In case of overflow */
   if (new_capacity < btree->capacity) return;

   BinTreeNode* new_root = NULL;
   uint64_t tries = 0;
   while (new_root == NULL && tries < 1000) {
      new_root = (BinTreeNode*) realloc(btree->root, sizeof(BinTreeNode)*new_capacity);
      tries++;
   }

   if (new_root == NULL) {
      puts("Failed to allocate enough memory for tree resize.\n");
      exit(EXIT_FAILURE);
   }

   btree->root = new_root;
   btree->capacity = new_capacity;
}

void
tree_binary_insert(BinTree *btree, key_t key) {

   /* NA OPERAÇÃO DE INSERÇÃO QUANDO UMA CHAVE JÁ EXISTIR, NÃO É CRIADA NOVA CHAVE */
   if (IDX_INVALID != tree_binary_search_key_level(*btree, key)) {
      return;
   };

   if (btree->elements == btree-> capacity) {
      /*puts("capacity exceeded");*/
      /*printf("capacity = %d\n", btree->capacity);*/
      tree_binary_resize(btree);
      /*printf("new capacity = %d\n", btree->capacity);*/
   }

   BinTreeNode* root = btree->root;
   BinTreeNode* node = NULL;
   uint32_t inicial_elements = btree->elements;

   /* Está vazia */
   if (inicial_elements == 0) {
      node = btree->root;
   } else {
      /* Devido às propriedades das àrvores binárias implicitas,
       * o indice corresponde ao número de elementos-1,
       * como é o próximo indice, -1+1 = 0 */
      node = &root[inicial_elements];
      /* O pai do NOVO filho está em (Nº Elementos+1) / 2 arredondado para baixo -1 para o indice
       * O que pode ser simplificado para simplesmente o numero de elements>>1
       * O módulo diz-nos se é para a esquerda ou direita */
      BinTreeNode* parent_node = &root[inicial_elements>>1];
      if (inicial_elements % 2 == 0) {
         parent_node->idx_left = inicial_elements;
      } else {
         parent_node->idx_right = inicial_elements;
      }
   }

   /* Inserir nova chave  */
```

```
    node->data = key;
    btree->elements = inicial_elements + 1;
}

void
tree_binary_insert_arr(BinTree *btree, key_t* arr, key_t size) {
    for (key_t k = 0; k < size; k++ ) {
        tree_binary_insert(btree, arr[k]);
    }
}

void
tree_binary_print_inorder(BinTree *btree) {

    /* Helper function to print the entire binary tree */
    void inorder(BinTree *btree, idx_t idx) {
        if (idx == 0) return;
        BinTreeNode* node = btree->root + idx;
        (void) inorder(btree, node->idx_left);
        (void) printf("%d ", node->data);
        (void) inorder(btree, node->idx_right);
    }

    BinTreeNode* root = btree->root;
    (void) printf("In-order traversal of the binary tree:\n");
    (void) printf("%d ", root->data);
    (void) inorder(btree, root->idx_left);
    (void) inorder(btree, root->idx_right);
    (void) puts("\n");
}

void
tree_binary_print(BinTree *btree) {

    uint32_t levels = 0;
    uint32_t n_elem = btree->elements;

    /* Contar niveis */
    while (n_elem > 1) {
        n_elem = n_elem >> 1;
        levels++;
    }

    BinTreeNode *root = btree->root;
    printf("%2d\n", root->data);

    /* Imprimir cada nivel */
    uint32_t idx = 1;
    uint32_t n_nodes = 1;
    for (uint32_t l = 0; l < levels; l++) {
        /* Cada nivel tem o dobro dos elementos no maximo  */
        n_nodes = n_nodes << 1;
        for (uint32_t i = 0; i < n_nodes; i++) {
            if (idx == btree->elements-1) break;
            printf("%2d  ", (root+idx)->data );
            idx++;
        }
        puts("");
    }

}
```

```
idx_t
tree_binary_search_key_inorder(BinTree btree, int32_t key) {

   /*Helper function */
   idx_t tree_binary_search(BinTreeNode *root, idx_t idx, int32_t key) {
      if (idx == IDX_INVALID || idx == 0) return IDX_INVALID;

      BinTreeNode node = root[idx];
      if (node.data == key) return idx;

      idx_t left = tree_binary_search(root, node.idx_left, key);
      if (left != IDX_INVALID) return left;

      idx_t right = tree_binary_search(root, node.idx_right, key);
      if (right != IDX_INVALID) return right;

      return IDX_INVALID;
   }

   BinTreeNode* root = btree.root;
   if (root->data == key) return 0;

   idx_t left = tree_binary_search(root, root->idx_left, key);
   if (left != IDX_INVALID) return left;

   idx_t right = tree_binary_search(root, root->idx_right, key);
   if (right != IDX_INVALID) return right;


   return IDX_INVALID;
}

idx_t
tree_binary_search_key_level(BinTree btree, int32_t key) {

   /* Como não existe ordem inerente nesta àrvore binária, os nós estão
    * inseridos no array da esquerda para a direita, logo posso percorrer o array.
    * Vou optimizar porque sim. */

   register BinTreeNode *ptr_front = btree.root;
   register BinTreeNode *ptr_back = btree.root + btree.elements;

   /*printf("Search key = %d\n", key);*/

   register int found = 0;
   while (ptr_front + 7 < ptr_back) {

      /* prefetch */
      __builtin_prefetch(ptr_front + 32, 0, 1);
      __builtin_prefetch(ptr_back - 32, 0, 1);

      /* Unrolling */
      found |= (ptr_front->data == key);
      found |= ((ptr_front + 1)->data == key);
      found |= ((ptr_front + 2)->data == key);
      found |= ((ptr_front + 3)->data == key);

      found |= (ptr_back->data == key);
      found |= ((ptr_back - 1)->data == key);
      found |= ((ptr_back - 2)->data == key);
```

```
        found |= ((ptr_back - 3)->data == key);

        ptr_front += 4;
        ptr_back -= 4;

        if (found) return 0;
    }

    /* elementos restante */
    while (ptr_front <= ptr_back) {
        found |= (ptr_front->data == key) | (ptr_back->data == key);
        ptr_front++;
        ptr_back--;
    }

    return found ? 0 : IDX_INVALID;
}

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

    BinTree btree;
    clock_t start = 0, end = 0;
    clock_t total = 0;

    for (int i = 0; i < g_average; i++) {
        start = clock();
        btree = tree_binary_create(g_treesize);
        tree_binary_insert_arr(&btree, arr, g_treesize);
        end = clock();

        total += (end-start);
        tree_binary_destroy(btree);
    }

    double total_time = ((double) total*1000) / CLOCKS_PER_SEC;
    fprintf(fptr, "Binary Tree = %0.4lfms\t(0 rotations)\n", total_time/g_average);
}

AVLTree
tree_avl_create(idx_t inicial_capacity) {
    assert(inicial_capacity > 0);

    AVLTree avl = {NULL, 0, 0, inicial_capacity};
    avl.nodes = (AVLNode*) malloc( sizeof(AVLNode) * inicial_capacity);

    if (avl.nodes == NULL) {
        perror("Couldn't allocate AVL tree.");
        exit(EXIT_FAILURE);
    }

    for (idx_t i = 0; i < inicial_capacity; i++) {
        avl.nodes[i] = (AVLNode) {IDX_INVALID, IDX_INVALID, 0, 1};
    }

    return avl;
}

void
tree_avl_destroy(AVLTree* avl) {
    assert(avl);
    free(avl->nodes);
}

void
tree_avl_resize(AVLTree *avl) {
    idx_t new_capacity = avl->capacity*RESIZE_FACTOR;
    if (new_capacity >= IDX_INVALID) {
        perror("AVL tree exceeded maximum capacity.");
        exit(EXIT_FAILURE);
    }

    AVLNode* new_nodes = (AVLNode*) realloc(avl->nodes, sizeof(AVLNode)*new_capacity);
    if (new_nodes == NULL) {
        perror("Failed to allocate enough memory for tree resize.");
        exit(EXIT_FAILURE);
    }

    avl->nodes = new_nodes;
    avl->capacity = new_capacity;
}


static int
_avl_get_height(AVLTree* avl, idx_t index) {
    return (index == IDX_INVALID) ? 0 : avl->nodes[index].height;
}

static int
_avl_get_balance(AVLTree* avl, idx_t index) {
    if (index == IDX_INVALID) return 0;
    return _avl_get_height(avl, avl->nodes[index].left) - _avl_get_height(avl, avl->nodes[index].right);
}


static idx_t
_avl_rotate_right(AVLTree *avl, idx_t node_idx) {
    g_rotation_count++;

    idx_t pivot = avl->nodes[node_idx].left;
    idx_t T2 = avl->nodes[pivot].right;

    // Perform rotation:
    avl->nodes[pivot].right = node_idx;
    avl->nodes[node_idx].left = T2;

    // Update heights:
    avl->nodes[node_idx].height = 1 + max(_avl_get_height(avl, avl->nodes[node_idx].left),
_avl_get_height(avl, avl->nodes[node_idx].right));
    avl->nodes[pivot].height = 1 + max(_avl_get_height(avl, avl->nodes[pivot].left), _avl_get_height(avl,
avl->nodes[pivot].right));
    return pivot;
}

static idx_t
_avl_rotate_left(AVLTree *avl, idx_t x_index) {
    g_rotation_count++;

    idx_t pivot = avl->nodes[x_index].right;
    idx_t T2 = avl->nodes[pivot].left;

    // Perform rotation:
```

```
    avl->nodes[pivot].left = x_index;
    avl->nodes[x_index].right = T2;

    // Update heights:
    avl->nodes[x_index].height = 1 + max(_avl_get_height(avl, avl->nodes[x_index].left),
_avl_get_height(avl, avl->nodes[x_index].right));
    avl->nodes[pivot].height = 1 + max(_avl_get_height(avl, avl->nodes[pivot].left), _avl_get_height(avl,
avl->nodes[pivot].right));
    return pivot;
}

static idx_t
_avl_insert_recursive(AVLTree *avl, idx_t node_index, int key) {

    if (node_index == IDX_INVALID) {
        idx_t new_index = avl->elements;
        AVLNode* new_node = &avl->nodes[new_index];
        new_node->key = key;
        new_node->left = IDX_INVALID;
        new_node->right = IDX_INVALID;
        new_node->height = 1;
        avl->elements++;
        return new_index;
    }

    /* Binary Search Tree */
    if (key < avl->nodes[node_index].key) {
        avl->nodes[node_index].left = _avl_insert_recursive(avl, avl->nodes[node_index].left, key);
    } else if (key > avl->nodes[node_index].key) {
        avl->nodes[node_index].right = _avl_insert_recursive(avl, avl->nodes[node_index].right, key);
    } else {
        return node_index;
    }

    avl->nodes[node_index].height = 1 + max(_avl_get_height(avl, avl->nodes[node_index].left),
                                    _avl_get_height(avl, avl->nodes[node_index].right));

    // Get balance factor to check if rebalancing is needed.
    int balance = _avl_get_balance(avl, node_index);

    /* Left Left */
    if (balance > 1 && key < avl->nodes[avl->nodes[node_index].left].key)
        return _avl_rotate_right(avl, node_index);

    /* Right Right */
    if (balance < -1 && key > avl->nodes[avl->nodes[node_index].right].key)
        return _avl_rotate_left(avl, node_index);

    /* Left Right */
    if (balance > 1 && key > avl->nodes[avl->nodes[node_index].left].key) {
        avl->nodes[node_index].left = _avl_rotate_left(avl, avl->nodes[node_index].left);
        return _avl_rotate_right(avl, node_index);
    }

    /* Right Left */
    if (balance < -1 && key < avl->nodes[avl->nodes[node_index].right].key) {
        avl->nodes[node_index].right = _avl_rotate_right(avl, avl->nodes[node_index].right);
        return _avl_rotate_left(avl, node_index);
    }

    return node_index;
```

```
}

void
tree_avl_insert(AVLTree *avl, int key) {

   if (avl->elements == avl->capacity) {
      tree_avl_resize(avl);
   }


   /* For an empty tree, set the new node as root. */
   if (avl->elements == 0) {
      avl->tree_root = _avl_insert_recursive(avl, IDX_INVALID, key);
   } else {
      avl->tree_root = _avl_insert_recursive(avl, avl->tree_root, key);
   }
}
void
tree_avl_in_order(AVLTree *avl) {

   void _traverse(AVLNode *nodes, idx_t i) {
      if (i == IDX_INVALID) return;
      AVLNode no = nodes[i];
      _traverse(nodes, no.left);
      printf("%d ", no.key);
      _traverse(nodes, no.right);
   }

   _traverse(avl->nodes, avl->tree_root);
   puts("");
}

void
tree_avl_in_order_non(AVLTree *avl) {

   AVLNode current_node;
   idx_t current_index = avl->tree_root;

   while (current_index != IDX_INVALID) {
      current_node = avl->nodes[avl->tree_root];

      /* Traverse left sub-tree until leaf */
      if (current_node.left != IDX_INVALID) {
         current_index = current_node.left;
         continue;
      }
      printf("%d ", current_node.key);
   }
}


AVLNode*
tree_avl_search(AVLTree *avl, int key) {
   idx_t current_index = avl->tree_root;
   while (current_index != IDX_INVALID) {
      if (avl->nodes[current_index].key == key)
         return &avl->nodes[current_index];

      else if (key < avl->nodes[current_index].key)
         current_index = avl->nodes[current_index].left;
```

```
        else
            current_index = avl->nodes[current_index].right;
    }
    return NULL;  // Key not found.
}


void
tree_avl_insert_arr(AVLTree *avl, key_t* arr, size_t size) {
    assert(avl && arr);
    for (key_t* endptr = arr+size; arr != endptr && arr != NULL; arr++) {
        tree_avl_insert(avl, *arr);
    }
}


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

    AVLTree avl;
    clock_t start = 0, end = 0;
    clock_t total = 0;

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

    for (int i = 0; i < g_average; i++) {
        start = clock();
        avl = tree_avl_create(10);
        tree_avl_insert_arr(&avl, arr, g_treesize);
        end = clock();

        total += (end-start);
        tree_avl_destroy(&avl);
    }

    double total_time = ((double) total*1000) / CLOCKS_PER_SEC;
    fprintf(fptr, "AVL Tree = %0.4lfms\t(%d rotations)\n", total_time/g_average,
g_rotation_count/g_average);
}

/* Red Black Tree Implementation */
/* Criar arvore */
RBTree
tree_rb_create(uint32_t initial_capacity) {
    RBTree tree;
    tree.nodes = malloc(initial_capacity * sizeof(RBNode));
    assert(tree.nodes != NULL);

    tree.elements = 0;
    tree.capacity = initial_capacity;

    // a raiz da arvore é invalida inicialmente
    // para que seja pintada correctamente de preto (caso especial)
    // e inserida imediatamente
    tree.tree_root = IDX_INVALID;

    RBNode *endptr = tree.nodes+tree.capacity;
    for (RBNode *ptr = tree.nodes; ptr != endptr; ptr++) {
        ptr->key = 0;
        ptr->color = -1;
    ptr->left = IDX_INVALID;
    ptr->right = IDX_INVALID;
  }
  return tree;
}

/* Destruir árvore */
void
tree_rb_destroy(RBTree *rb) {
  free(rb->nodes);
}

/* Aumentar capacidade */
void
tree_rb_resize(RBTree *tree) {
  assert(tree != NULL);
  uint32_t new_capacity = tree->capacity * RESIZE_FACTOR;

  RBNode *new_nodes = realloc(tree->nodes, new_capacity * sizeof(RBNode));
  if (new_nodes == NULL) {
    free(tree->nodes);
    perror("Failed to allocate more nodes.");
    exit(EXIT_FAILURE);
  }

  // inicializar novos nós
  RBNode *endptr = new_nodes + new_capacity;
  for (RBNode *ptr = new_nodes + tree->capacity; ptr != endptr; ptr++) {
    ptr->key = 0;
    ptr->color = -1;
    ptr->left = IDX_INVALID;
    ptr->right = IDX_INVALID;
  }

  tree->nodes = new_nodes;
  tree->capacity = new_capacity;
}


/* 1 (verdadeiro) se o nó for vermelho */
static int
_rb_is_red(RBTree *tree, idx_t i) {
  // a raiz da arvore é invalida inicialmente
  // isto significa que vai ser pintada correctamente de preto
  if (i == IDX_INVALID) return 0;
  return (tree->nodes[i].color == RED);
}

/* rotação à esquerda */
static idx_t
_rb_rotate_left(RBTree *tree, idx_t h) {
  g_rotation_count++;
  idx_t pivot = tree->nodes[h].right;
  tree->nodes[h].right = tree->nodes[pivot].left;
  tree->nodes[pivot].left = h;
  tree->nodes[pivot].color = tree->nodes[h].color;
  tree->nodes[h].color = RED;
  return pivot;
}

/* rotação à direita */
```

```
static idx_t
_rb_rotate_right(RBTree *tree, idx_t h) {
   g_rotation_count++;
   idx_t pivot = tree->nodes[h].left;
   tree->nodes[h].left = tree->nodes[pivot].right;
   tree->nodes[pivot].right = h;
   tree->nodes[pivot].color = tree->nodes[h].color;
   tree->nodes[h].color = RED;
   return pivot;
}

/* Inverter cores */
static void
_rb_flip_colors(RBTree *tree, idx_t h) {
   tree->nodes[h].color = !tree->nodes[h].color;
   idx_t left = tree->nodes[h].left;
   idx_t right = tree->nodes[h].right;
   if (left != IDX_INVALID)
      tree->nodes[left].color = !tree->nodes[left].color;
   if (right != IDX_INVALID)
      tree->nodes[right].color = !tree->nodes[right].color;
}

/* Resolve comflictos */
static idx_t
_rb_fix_up(RBTree *tree, idx_t h) {
   /* Caso 1: direita vermelha e esquerda preta -> rotação à esquerda */
   if (_rb_is_red(tree, tree->nodes[h].right) && !_rb_is_red(tree, tree->nodes[h].left))
      h = _rb_rotate_left(tree, h);
   /* Caso 2: filho esquerdo vermelho e neto esquerdo vermelho -> rotação à direita */
   if (_rb_is_red(tree, tree->nodes[h].left) && _rb_is_red(tree, tree->nodes[tree->nodes[h].left].left))
      h = _rb_rotate_right(tree, h);
   /* Caso 3: ambos os filhos são vermelhos */
   if (_rb_is_red(tree, tree->nodes[h].left) && _rb_is_red(tree, tree->nodes[h].right))
      _rb_flip_colors(tree, h);
   return h;
}

/* Inserção recursiva: Devolve o novo indice da raiz se inserir */
static idx_t
_rb_insert_recursive(RBTree *tree, idx_t h, key_t key) {

   /* Inserir após encontrar nova folha
    * (chamada anterior para filho que não existe) */
   if (h == IDX_INVALID) {
      idx_t new_index = tree->elements;
      tree->nodes[new_index].key = key;
      tree->nodes[new_index].left = IDX_INVALID;
      tree->nodes[new_index].right = IDX_INVALID;
      tree->nodes[new_index].color = RED;  // sempre vermelho
      tree->elements++;
      return new_index;
   }

   /* Recursão equivalente a binary search tree */
   if (key < tree->nodes[h].key) {
      tree->nodes[h].left = _rb_insert_recursive(tree, tree->nodes[h].left, key);
   } else if (key > tree->nodes[h].key) {
      tree->nodes[h].right = _rb_insert_recursive(tree, tree->nodes[h].right, key);
   }
```

```
    /* É necessário corrigir erros causados pela inserção */
    h = _rb_fix_up(tree, h);
    return h;
}

/* Inserir nó */
void
tree_rb_insert(RBTree *tree, key_t key) {

    /* Aumentar capacidade  se for necessário */
    if (tree->capacity == tree->elements)
        tree_rb_resize(tree);

    tree->tree_root = _rb_insert_recursive(tree, tree->tree_root, key);
    tree->nodes[tree->tree_root].color = BLACK;
}

/* Pesquisa */
int
tree_rb_search(RBTree *tree, int key) {
    RBNode *nodes = tree->nodes;
    idx_t current = tree->tree_root;

    /* binary search tree search */
    while (current != IDX_INVALID) {
        printf("current = %d\n", current);
        if (key < nodes[current].key)
            current = nodes[current].left;
        else if (key > nodes[current].key)
            current = nodes[current].right;
        else
            return current;
    }

    return -1;
}


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

    RBTree vp;
    clock_t start = 0, end = 0;
    clock_t total = 0;

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

    for (int i = 0; i < g_average; i++) { start = clock();
        vp = tree_rb_create(g_treesize);
        for (idx_t idx = 0; idx < g_treesize; idx++)
            tree_rb_insert(&vp, arr[idx]);
        end = clock();

        total += (end-start);
        tree_rb_destroy(&vp);
    }

    double total_time = ((double) total*1000) / CLOCKS_PER_SEC;
    fprintf(fptr, "RB Tree = %0.4lfms\t(%d rotations)\n", total_time/g_average,
g_rotation_count/g_average);
```

```
}

/* Treap Functions */
Treap
tree_treap_create(idx_t initial_capacity) {
   Treap new_treap;
   new_treap.nodes = (TreapNode*) malloc(sizeof(TreapNode) * initial_capacity);
   if (new_treap.nodes == NULL) {
      perror("Failed to allocate Treap.");
      exit(EXIT_FAILURE);
   }

   new_treap.tree_root = IDX_INVALID;
   new_treap.elements = 0;
   new_treap.capacity = initial_capacity;

   /* inicializar novos nós */
   TreapNode* endptr = new_treap.nodes + initial_capacity;
   for (TreapNode *ptr = new_treap.nodes; ptr != endptr; ptr++) {
      *ptr = (TreapNode){0, 0, IDX_INVALID, IDX_INVALID};
   }

   return new_treap;
}

void tree_treap_resize(Treap *treap) {
   /* se a capacity for máxima */
   if (treap->capacity == IDX_INVALID - 1) return;

   idx_t old_capacity = treap->capacity;
   idx_t new_capacity = (idx_t)(treap->capacity * RESIZE_FACTOR);

   /* se a nova capacity for maior que a capacidade máxima */
   if (new_capacity < treap->capacity) {
      new_capacity = IDX_INVALID - 1;
   }

   TreapNode *new_nodes = (TreapNode*) realloc(treap->nodes, sizeof(TreapNode) * new_capacity);
   if (new_nodes == NULL) {
      perror("Failed to realloc new nodes.");
      exit(EXIT_FAILURE);
   }
   treap->nodes = new_nodes;

   /* incializar nova memóra */
   for (idx_t i = old_capacity; i < new_capacity; i++) {
      treap->nodes[i] = (TreapNode){0, 0, IDX_INVALID, IDX_INVALID};
   }

   treap->capacity = new_capacity;
}

void
tree_treap_destroy(Treap *treap) {
   free(treap->nodes);
   treap->capacity = 0;
   treap->elements = 0;
}


static idx_t
```

```
_treap_rotate_right(Treap *treap, idx_t no_idx) {
    g_rotation_count++;

    TreapNode *nodes = treap->nodes;
    idx_t pivot_idx = nodes[no_idx].left;

    /* à esquerda agora fica a subtree do pivot */
    nodes[no_idx].left = nodes[pivot_idx].right;
    /* à direita do pivot fica o nó atual */
    nodes[pivot_idx].right = no_idx;

    /* o pivot não muda de sitio mas pode passar a ser a nova raiz */
    return pivot_idx;
}

static idx_t
_treap_rotate_left(Treap *treap, idx_t no_idx) {
    g_rotation_count++;

    TreapNode *nodes = treap->nodes;
    idx_t pivot_idx = nodes[no_idx].right;

    /* subtree do pivot */
    nodes[no_idx].right = nodes[pivot_idx].left;
    /* o pivot agora leva ao nó */
    nodes[pivot_idx].left = no_idx;

    /* o pivot pode passar a ser a nova raiz */
    return pivot_idx;
}

static idx_t
_treap_insert_recursive(Treap *treap, idx_t idx, key_t key) {
    TreapNode *nodes = treap->nodes;

    /* Criamos um novo nó quando quando o idx é inválido,
     * ou seja, quando o BST tenta inserir numa folha
     * depois devolvemos o novo indice à chamada anterior desta função*/
    if (idx == IDX_INVALID) {
        idx_t new_index = treap->elements;
        treap->elements++;
        nodes[new_index] = (TreapNode){
            .key = key,
            .priority = (idx_t)rand_idx(1, IDX_INVALID - 1),
            .left = IDX_INVALID,
            .right = IDX_INVALID
        };
        return new_index;
    }

    /* inserir tipo binary search tree */
    if (key < nodes[idx].key) {
        nodes[idx].left = _treap_insert_recursive(treap, nodes[idx].left, key);

        /* manter max heap */
        if (nodes[nodes[idx].left].priority > nodes[idx].priority) {
            idx = _treap_rotate_right(treap, idx);
        }
    } else if (key > nodes[idx].key) {
        nodes[idx].right = _treap_insert_recursive(treap, nodes[idx].right, key);
```

```c
        /* 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);
    }
    double total_time = ((double) total*1000) / CLOCKS_PER_SEC;
    fprintf(fptr, "TREAP = %0.4lfms\t(%d rotations)\n", total_time/g_average,
g_rotation_count/g_average);
}

void tree_treap_inorder_print(Treap *treap, idx_t root) {
    if (root == IDX_INVALID) return;
    tree_treap_inorder_print(treap, treap->nodes[root].left);
    printf("%d ", treap->nodes[root].key);
    tree_treap_inorder_print(treap, treap->nodes[root].right);
}

int
main(int argc, char *argv[]) {

    if (argc != 3) {
        puts("aed-prej2 [treesize] [average]");
        exit(EXIT_FAILURE);
    }

    g_treesize = atoi(argv[1]);
    g_average = atoi(argv[2]);

    if (g_treesize < 0) {
        puts("Invalid tree size.");
        exit(EXIT_FAILURE);
    } else if (g_average < 0) {
        puts("Invalid average.");
        exit(EXIT_FAILURE);
    }

    srand(SEED);

    key_t *conjunto_a = arr_gen_conj_a(g_treesize);
    key_t *conjunto_b = arr_gen_conj_b(g_treesize);
    key_t *conjunto_c = arr_gen_conj_c(g_treesize);
    key_t *conjunto_d = arr_gen_conj_d(g_treesize);

    FILE* filelog = fopen("log.txt", "a");
    fprintf(filelog, "\n=== NEW LOG === (Treesize = %d, Average = &d)\n", g_treesize, g_average);

    puts("Testing binary search tree...");
    binary_test_and_log(conjunto_a, filelog);
    binary_test_and_log(conjunto_b, filelog);
    binary_test_and_log(conjunto_c, filelog);
    binary_test_and_log(conjunto_d, filelog);

    puts("Testing AVL tree...");
    avl_test_and_log(conjunto_a, filelog);
    avl_test_and_log(conjunto_b, filelog);
    avl_test_and_log(conjunto_c, filelog);
    avl_test_and_log(conjunto_d, filelog);

    puts("Testing Red-Black tree...");
    rb_test_and_log(conjunto_a, filelog);
    rb_test_and_log(conjunto_b, filelog);
    rb_test_and_log(conjunto_c, filelog);
    rb_test_and_log(conjunto_d, filelog);

    puts("Testing RB search tree...");
    treap_test_and_log(conjunto_a, filelog);
    treap_test_and_log(conjunto_b, filelog);
    treap_test_and_log(conjunto_c, filelog);
    treap_test_and_log(conjunto_d, filelog);

    free(conjunto_a);
    free(conjunto_b);
    free(conjunto_c);
    free(conjunto_d);

    fclose(filelog);

    puts("Done!");
    system("notify-send -u critical done");
    return 0;
}
```