MINISTÉRIO DA EDUCAÇÃO

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Campus Santa Mônica

CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RENAN JUSTINO REZENDE SILVA

YURI HENRIQUE BERNARDES MACIEL

ALGORITMOS E ESTRUTURAS DE DADOS 2

ATIVIDADE AVALIATIVA 3 - AVL & LIST DICTIONARY

UBERLÂNDIA-MG

OUTUBRO/2021

RENAN JUSTINO

YURI HENRIQUE BERNARDES MACIEL

ALGORITMOS E ESTRUTURAS DE DADOS 2

ATIVIDADE AVALIATIVA 3 - AVL & LIST DICTIONARY

Trabalho avaliativo apresentado como requisito parcial para obtenção de nota na disciplina de Algoritmos e Estruturas de Dados 2 do Curso de Bacharelado em Ciência da Computação da Universidade Federal de Uberlândia - Campus Santa Mônica.

Professor/Orientador:
Prof.ª Dra. Maria Adriana Vidigal de Lima

UBERLÂNDIA-MG

OUTUBRO/2021

SUMÁRIO

1 INTRODUÇÃO

Este projeto de relatório irá apresentar as seções de introdução, solução, implementação e conclusão de um dicionário da língua portuguesa utilizando as estruturas de dados árvore AVL e Lista.

Uma árvore AVL é uma árvore binária de busca balanceada. Uma árvore balanceada são as árvores que diminuem o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrências idênticas. Em uma árvore AVL, as alturas das duas subárvores filhas de qualquer nó diferem em no máximo um e se a qualquer momento eles diferirem em mais de um, o balanceamento é realizado para restaurar essa propriedade

Uma lista encadeada é um conjunto linear de elementos de dados, na qual a ordem linear não é fornecida por sua colocação física na memória. Em vez disso, cada elemento aponta para o próximo. É uma estrutura de dados que consiste em um grupo de nós que juntos representam uma sequência

Seguindo as especificações do projeto, devemos realizar uma implementação que consiga realizar as seguintes operações:

1. Carregar (ou atualizar) o dicionário a partir de um arquivo (texto) de entrada na árvore AVL e também em uma lista simplesmente encadeada (inserção ordenada);
2. Buscar uma entrada na AVL: exibir o significado de uma palavra e sua classificação;

2a. Comparar a busca na AVL com a busca na lista exibindo os tempos;

3. Inserir uma entrada (palavra, classificação, significado) na AVL / lista;
4. Remover uma entrada (palavra, classificação, significado) na AVL / lista;
5. Dada uma frase, exibir as palavras/significados que foram encontrados no dicionário a partir da AVL;
6. Exibir um relatório consolidado sobre o dicionário com a contagem das palavras para cada letra do alfabeto a partir da AVL;
7. Salvar o dicionário em um arquivo texto (percurso escolhido pelo usuário: pré ou pós-ordem) a partir da AVL.

2 DEFINIÇÃO DA SOLUÇÃO

Na solução, os arquivos estão organizados em duas pastas, uma pasta para os arquivos da árvore (tree) e outra pasta para os arquivos da lista (list), além do código main e do arquivo dict.txt que contém o dicionário com as palavras organizadas em triplas. Na pasta tree se encontra os arquivos tree.c e tree.h. Na pasta list se encontra os arquivos list.c e list.h.

Tanto a Lista quanto a árvore AVL foram implementadas para armazenar uma tripla de informações sobre uma palavra do dicionário: a palavra, sua classificação e seu significado. O tamanho máximo para cada campo da tripla é de 255 caracteres – 254 na prática, pois o último caractere é reservado para '\0' em C.

Após as definições dos protótipos e estruturas, foi implementado as funções nos arquivos .c que estão na seção de implementação. O código main contém a execução principal em forma de menu com a chamada das funções pedidas para o trabalho, tendo operações executadas simultaneamente na árvore AVL e na Lista quando necessário.

Repositório com a solução: https://replit.com/@YuriHenrique1/trabalho03AED22021

3 IMPLEMENTAÇÃO

Segue abaixo o código fonte de todas funções do enunciado e de toda implementação do trabalho.

Código 1 – Arquivo tree.h (cabeçalho de funções da árvore AVL do trabalho)

```h
#include <dirent.h>

typedef struct node* AVL;

AVL* createAVL();
void freeAVL(AVL* root);
void printAVL(AVL* root);
void countLettersAVL(AVL* root, int count[26]);
int isEmptyAVL(AVL* root);
int totalNodesAVL(AVL* root);
int equalsAVL(AVL* r1, AVL* r2);
int AVLHeight(struct node* node);
int removeNodeAVL(AVL* root, char* data);
int removeMinorsAVL(AVL* root, char* data);
int insertNodeAVL(AVL* root, char data[3][MAX_INPUT]);
char** queryAVL(AVL* root, char* data);
```

Fonte: autores deste relatório

Código 2 – Arquivo tree.c (Implementação Funções Árvore AVL)

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include "tree.h"

struct node {
    // info[0] -> Palavra
    // info[1] -> Classificação
    // info[2] -> Significado
    // Até 255 caracteres para cada informação
    char info[3][MAX_INPUT];
    // Altura do nó
    int height;
    // Ponteiro para subárvore a esquerda
    struct node* left;
    // Ponteiro para subárvore a direita
    struct node* right;
};
```

```c
// Cria uma árvore
AVL* createAVL() {
    AVL* root = (AVL*)malloc(sizeof(AVL));
    if (root != NULL)
        *root = NULL;
    return root;
}

// Libera um nó da memória
void freeNode(struct node* node) {
    if (node == NULL)
        return;
    freeNode(node->left);
    freeNode(node->right);
    free(node);
    node = NULL;
}

// Libera uma árvore da memória
void freeAVL(AVL* root) {
    if (root == NULL)
        return;
    // Libera cada nó
    freeNode(*root);
    // Libera a raiz
    free(root);
}

// Retorna a altura no nó
int nodeHeight(struct node* node) {
    if (node == NULL)
        return -1;
    else
        return node->height;
}

// Retorna fator de balanceamento do nó
int nodeBalanceFactor(struct node* node) {
    return labs(nodeHeight(node->left) - nodeHeight(node->right));
}

// Verifica se a árvore está vazia
int isEmptyAVL(AVL* root) {
    if (root == NULL)
        return 1;
    if (*root == NULL)
```

```
        return 1;
    return 0;
}

// Retorna o total de nós da árvore
int totalNodesAVL(AVL* root) {
    if (root == NULL)
        return 0;
    if (*root == NULL)
        return 0;
    int h_left = totalNodesAVL(&((*root)->left));
    int h_right = totalNodesAVL(&((*root)->right));
    return(h_left + h_right + 1);
}

// Retorna a altura da árvore
int AVLHeight(struct node* node) {
    if (node == NULL)
        return 0;
    int h_left = AVLHeight(node->left);
    int h_right = AVLHeight(node->right);
    if (h_left > h_right)
        return (h_left + 1);
    else
        return(h_right + 1);
}

// Imprime a árvore em stdout
void printAVL(AVL* root) {
    if (root == NULL)
        return;
    if (*root != NULL) {
        printAVL(&((*root)->left));
        printf("%s\n%s\n%s\n", (*root)->info[0], (*root)->info[1],
(*root)->info[2]);
        printAVL(&((*root)->right));
    }
}

// Recebe uma árvore AVL que representa um dicionário e um vetor de 26
posições.
// Cada índice do vetor é incrementado com base no caractere inicial da
palavra pela qual a função está passando.
void countLettersAVL(AVL* root, int count[26]) {
    if (root == NULL)
        return;
    if (*root != NULL) {
```

```
        countLettersAVL(&((*root)->left), count);
        // Incrementa o vetor no índice da letra inicial da palavra.
        // Cálculo feito com base na tabela ASCII.
        // 97 == a -> 97 - 97 == 0.
        // Todas as palavras são armzenadas em lowercase, o que permite
essa operação sempre ser válida.
        count[(*root)->info[0][0] - 97]++;
        countLettersAVL(&((*root)->right), count);
    }
}

// Verifica se um valor está presente na árvore.
// Caso positivo, retorna a informação no nó em que ele se encontra.
char** queryAVL(AVL* root, char* data) {
    if (root == NULL)
        return NULL;

    char** r = (char**)malloc(3 * sizeof(char*));
    for (size_t i = 0; i < 3; i++)
        r[i] = (char*)malloc(MAX_INPUT * sizeof(char));

    struct node* current = *root;
    while (current != NULL) {
        if (strcmp(data, current->info[0]) == 0) {
            for (size_t i = 0; i < 3; i++)
                strcpy(r[i], current->info[i]);

            return r;
        }
        if (strcmp(data, current->info[0]) > 0)
            current = current->right;
        else
            current = current->left;
    }

    return NULL;
}

// Retorna o maior valor entre x e y
int greater(int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}

// Realiza uma rotação a direita na árvore
```

```c
void rotationRight(AVL* A) {//LL
    printf("\nRotacao direita\n");

    struct node* B = (*A)->left;
    (*A)->left = B->right;
    B->right = (*A);

    (*A)->height = greater(nodeHeight((*A)->left), nodeHeight((*A)->right)) + 1;
    B->height = greater(nodeHeight(B->left), (*A)->height) + 1;

    *A = B;
}

// Realiza uma rotação a esquerda na árvore
void rotationLeft(AVL* A) {//RR
    printf("\nRotacao esquerda\n");

    struct node* B = (*A)->right;
    (*A)->right = B->left;
    B->left = (*A);

    (*A)->height = greater(nodeHeight((*A)->left), nodeHeight((*A)->right)) + 1;
    B->height = greater(nodeHeight(B->right), (*A)->height) + 1;

    (*A) = B;
}

// Realiza uma rotação dupla a direita na árvore
void doubleRotationRight(AVL* A) {//LR
    rotationLeft(&(*A)->left);
    rotationRight(A);
}

// Realiza uma rotação dupla a esquerda na árvore
void doubleRotationLeft(AVL* A) {//RL
    rotationRight(&(*A)->right);
    rotationLeft(A);
}

// Insere um novo nó na árvore
// data é um vetor de strings
int insertNodeAVL(AVL* root, char data[3][MAX_INPUT]) {
    int r;
    // Árvore vazia ou nó folha
    if (*root == NULL) {
```

```
        struct node* new;
        new = (struct node*)malloc(sizeof(struct node));
        if (new == NULL)
            return 0;

        strcpy(new->info[0], data[0]);
        strcpy(new->info[1], data[1]);
        strcpy(new->info[2], data[2]);
        new->height = 0;
        new->left = NULL;
        new->right = NULL;
        *root = new;
        return 1;
    }

    struct node* current = *root;
    if (strcmp(data[0], current->info[0]) < 0) {
        r = insertNodeAVL(&(current->left), data);
        if (r == 1) {
            if (nodeBalanceFactor(current) >= 2) {
                if (strcmp(data[0], (*root)->left->info[0]) < 0) {
                    rotationRight(root);
                } else {
                    doubleRotationRight(root);
                }
            }
        }
    } else {
        if (strcmp(data[0], current->info[0]) > 0) {
            r = insertNodeAVL(&(current->right), data);
            if (r == 1) {
                if (nodeBalanceFactor(current) >= 2) {
                    if ((*root)->right->info < data) {
                        rotationLeft(root);
                    } else {
                        doubleRotationLeft(root);
                    }
                }
            }
        } else {
            printf("Valor duplicado!!\n");
            return 0;
        }
    }

    current->height = greater(nodeHeight(current->left),
nodeHeight(current->right)) + 1;
```

```
    return r;
}

// Retorna o nó com o menor valor na ávore
struct node* searchMinor(struct node* root) {
    struct node* current, * next;
    if (isEmptyAVL(&root)) return NULL;
    current = root;
    next = root->left;
    while (next != NULL) {
        current = next;
        next = next->left;
    }
    return current;
}

// Remove o nó que contem a o valor de data em info
int removeNodeAVL(AVL* root, char* data) {
    int r;

    // Valor não existe
    if (*root == NULL) {
        printf("valor não encontrado!!\n");
        return 0;
    }

    if (strcmp(data, (*root)->info[0]) < 0) {
        r = removeNodeAVL(&(*root)->left, data);
        if (r == 1) {
            if (nodeBalanceFactor(*root) >= 2) {
                if (nodeHeight((*root)->right->left) <=
nodeHeight((*root)->right->right))
                    rotationLeft(root);
                else
                    doubleRotationLeft(root);
            }
        }
    }

    if (strcmp((*root)->info[0], data) < 0) {
        r = removeNodeAVL(&(*root)->right, data);
        if (r == 1) {
            if (nodeBalanceFactor(*root) >= 2) {
                if (nodeHeight((*root)->left->right) <=
nodeHeight((*root)->left->left))
                    rotationRight(root);
```

```
                else
                    doubleRotationRight(root);
            }
        }
    }

    if (strcmp((*root)->info[0], data) == 0) {
        if (((*root)->left == NULL || (*root)->right == NULL)) {// nó
tem 1 filho ou nenhum
            struct node* older = (*root);
            if ((*root)->left != NULL)
                *root = (*root)->left;
            else
                *root = (*root)->right;
            free(older);
        } else {
            // Nó tem 2 filhos
            struct node* temp = searchMinor((*root)->right);
            strcpy((*root)->info[0], temp->info[0]);
            strcpy((*root)->info[1], temp->info[1]);
            strcpy((*root)->info[2], temp->info[2]);
            removeNodeAVL(&(*root)->right, (*root)->info[0]);
            if (nodeBalanceFactor(*root) >= 2) {
                if (nodeHeight((*root)->left->right) <=
nodeHeight((*root)->left->left))
                    rotationRight(root);
                else
                    doubleRotationRight(root);
            }
        }
        if (*root != NULL)
            (*root)->height = greater(nodeHeight((*root)->left),
nodeHeight((*root)->right)) + 1;
        return 1;
    }

    (*root)->height = greater(nodeHeight((*root)->left),
nodeHeight((*root)->right)) + 1;
    return r;
}
```

Fonte: autores deste relatório

Código 3 – Arquivo list.h (Cabeçalho Funções Lista)

```c
#include <dirent.h>

typedef struct node* List;

List createList();
int isEmptyList(List list);
int insertOrdList(List* list, char data[3][MAX_INPUT]);
int removeOrdList(List* list, char* data);
int printList(List list);
int freeList(List* list);
char** queryList(List list, char* word);
```

Fonte: autores deste relatório

Código 4 – Arquivo list.c (Implementação Funções Lista)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"

struct node {
    // info[0] -> Palavra
    // info[1] -> Classificação
    // info[2] -> Significado
    // Até 255 caracteres para cada informação
    char info[3][MAX_INPUT];
    //Enderço do nó sucessor
    struct node* prox;
};

// Retorna um ponteiro vazio para a lista
List createList() {
    return NULL;
}

// Verifica se a lista esta vazia
int isEmptyList(List list) {
    if (list == NULL)
        return 1; //Vazia
    else
        return 0; //Não é vazia
}

// Insere elementos na lista de forma ordenada
```

```c
// data é um vetor de strings
int insertOrdList(List* list, char data[3][MAX_INPUT]) {
    //Envolve o percorrimento da lista
    List N = (List)malloc(sizeof(struct node));
    if (N == NULL)
        return 0;
    strcpy(N->info[0], data[0]);
    strcpy(N->info[1], data[1]);
    strcpy(N->info[2], data[2]);
    if (isEmptyList(*list) || strcmp(data[0], (*list)->info[0]) <= 0) {
        N->prox = *list;
        *list = N;
        return 1;
    }

    // Aponta para o 1° nó
    List aux = *list;
    // Percorre lista até achar o local correto para inserção do nó
    while (aux->prox != NULL && strcmp(data[0], aux->prox->info[0]) >
0)
        aux = aux->prox;
    N->prox = aux->prox;
    aux->prox = N;
    return 1;
}

// Realiza a remoção de um elemento em uma lista ordenada
// data é uma string contendo a palavra a ser removida
int removeOrdList(List* list, char* data) {
    if (isEmptyList(*list) || strcmp(data, (*list)->info[0]) < 0)
        return 0;
    // Aponta para o 1° no
    List aux = *list;
    if (strcmp(data, (*list)->info[0]) == 0) {
        *list = aux->prox;
        free(aux);
        return 1;
    }
    while (aux->prox != NULL && strcmp(aux->prox->info[0], data) < 0)
        aux = aux->prox; //Avança
    if (aux->prox == NULL || strcmp(aux->prox->info[0], data) > 0)
        return 0;
    List aux_2 = aux->prox; // Aponta para no a ser removido
    aux->prox = aux_2->prox; // Retira no da list
    free(aux_2); // Libera memória alocada
    return 1;
}
```

```c
// Função interna utilizada para impressão da lista
// Dada um nó da lista, retorna por referência o dado contido nele e o
aponta para o próximo nó
// data é um vetor de strings
int getPosList(List* aux, char** data) {
    if (*aux == NULL)
        return 0;
    strcpy(data[0], (*aux)->info[0]);
    strcpy(data[1], (*aux)->info[1]);
    strcpy(data[2], (*aux)->info[2]);
    *aux = (*aux)->prox;
    return 1;
}


// Imprime a lista em stdout
char** queryList(List list, char* word) {
    if (isEmptyList(list))
        return NULL;

    char** r = (char**)malloc(3 * sizeof(char*));
    for (size_t i = 0; i < 3; i++)
        r[i] = (char*)malloc(MAX_INPUT * sizeof(char));

    while (getPosList(&list, r) == 1)
        if (strcmp(word, r[0]) == 0) return r;

    return NULL;
}


// Imprime a lista em stdout
int printList(List list) {
    if (isEmptyList(list))
        return 0;

    printf("Dicionário (lista):");
    char** data = (char**)malloc(3 * sizeof(char*));
    for (size_t i = 0; i < 3; i++)
        data[i] = (char*)malloc(MAX_INPUT * sizeof(char));
    while (getPosList(&list, data) == 1)
        printf("\n-> %s\n - %s\n - %s\n", list->info[0], list->info[1],
list->info[2]);

    for (size_t i = 0; i < 3; i++)
        free(data[i]);
```

```
    free(data);

    return 1;
}

// Libera a lista da memória
int freeList(List* list) {
    if (isEmptyList(*list))
        return 0;

    while ((*list)->prox != NULL) {
        List aux = *list;
        *list = (*list)->prox;
        free(aux);
    }
    free(*list);
    *list = NULL;
    return 1;
}
```

Fonte: autores deste relatório

Código 5 – Arquivo principal main.c (Implementação código main.c)

```
#include <wait.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#define BILLION  1000000000.0

#include "tree/tree.h"
#include "list/list.h"

int searchWord(AVL* avl, List list, char* word);
int searchPhraseAVL(AVL* avl, char* word);
int readArchive(char arq[PATH_MAX], AVL* avl, List* list);
int writeArchiveAVL(char arq[PATH_MAX], AVL* avl);
char** split_string(char* phrase, int* r);
int toLower(char* str, size_t len);
void dicionaryReport(AVL* avl);


int main() {
```

```c
AVL* avl = createAVL();
List list = createList();

int op, var;
char word[MAX_INPUT], arq[PATH_MAX];

int exit = 0;
while (!exit) {

   printf("\n - Menu Arvore AVL Dicionario - \n"
      "1:Carregar dados a partir de arquivo\n"
      "2:Buscar palavra no dicionario\n"
      "3:Inserir dados no dicionario\n"
      "4:Remover palavra do dicionario\n"
      "5:Exibir significados dado frase\n"
      "6:Exibir Relatório\n"
      "7:Salvar dicionario em arquivo\n"
      "8:Encerrar\n -> ");

   scanf("%d", &op);

   switch (op) {

   case 1:
      printf("Carregar arquivo dicionario. \n");
      printf("Digite o path para o arquivo: ");
      scanf(" %4095[^\n]", arq);

      readArchive(arq, avl, &list) ?
         printf("Arvore atualizada com sucesso!\n") :
         printf("Nao foi possivel ler o arquivo: %s\n", arq);

      break;
   case 2:
      if (isEmptyAVL(avl)) {
         printf("Arvore vazia!");
         break;
      }

      printf("\nPalavra que deseja consultar: ");
      scanf(" %254[^\n]", word);

      if (toLower(word, strlen(word)) < 1) {
         printf("\nErro na consulta da palavra.\n");
         break;
      }
```

```
            searchWord(avl, list, word);

            break;
        case 3:
            char data[3][MAX_INPUT];
            printf("\nQual palavra deseja inserir: ");
            scanf(" %254[^\n]", data[0]);
            if (toLower(data[0], strlen(data[0])) < 1) {
                printf("\nErro no cadastro da palavra.\n");
                break;
            }

            printf("Classificacao de %s: ", data[0]);
            scanf(" %254[^\n]", data[1]);

            printf("Significado de %s: ", data[0]);
            scanf(" %254[^\n]", data[2]);

            insertNodeAVL(avl, data) && insertOrdList(&list, data) ?
                printf("Dados inseridos.\n") :
                printf("Houve um erro ao inserir os dados.\n");

            break;
        case 4:
            if (isEmptyAVL(avl)) {
                printf("\nDicionario esta vazio.\n");
                break;
            }

            printf("\nEntrada a ser removida: ");
            scanf(" %254[^\n]", word);
            if (toLower(word, strlen(word)) < 1) {
                printf("\nErro na remocao da palavra.\n");
                break;
            }

            removeNodeAVL(avl, word) && removeOrdList(&list, word) ?
                printf("Entrada removida.\n") :
                printf("Entrada nao encontrada.\n");

            break;

        case 5:
            if (isEmptyAVL(avl)) {
                printf("\nDicionario esta vazio.\n");
                break;
            }
```

```
         char phrase[MAX_INPUT];
         printf("\nDigite uma frase (até 254 caracteres) para exibir
significados: ");
         scanf(" %254[^\n]", phrase);
         if (toLower(phrase, strlen(phrase)) < 1) {
            printf("\nErro na remocao da palavra.\n");
            break;
         }

         searchPhraseAVL(avl, phrase);

         break;
      case 6:
         if (isEmptyAVL(avl)) {
            printf("\nDicionario esta vazio.\n");
            break;
         }

         dicionaryReport(avl);

         break;
      case 7:
         if (isEmptyAVL(avl)) {
            printf("\nDicionario esta vazio.\n");
            break;
         }

         printf("\nDigite o path para onde o dicionario deve ser salvo:
\n");
         scanf(" %4095[^\n]", arq);

         writeArchiveAVL(arq, avl) < 0 ?
            printf("\nNao foi possivel criar o arquivo!\n") :
            printf("\nSalvo com sucesso em: %s", arq);

         break;
      case 8:
         exit = 1;

         break;
      default:
         printf("\nOpcao invalida.\n");
      }
   }

   return 0;
```

```c
}

// Função que lê os dados em um arquivo e os insere na árvore AVL e na
Lista.
int readArchive(char arq[PATH_MAX], AVL* avl, List* list) {
   FILE* file = fopen(arq, "r");
   if (!file) {
      printf("Arquivo não encontrado!\n");
      return 0;
   }

   char data[3][MAX_INPUT];

   while (1) {

      if (fgets(data[0], 254, file) == NULL) break;
      data[0][strcspn(data[0], "\n")] = '\0';
      toLower(data[0], strlen(data[0]));

      if (fgets(data[1], 254, file) == NULL) break;
      data[1][strcspn(data[1], "\n")] = '\0';

      if (fgets(data[2], 254, file) == NULL) break;
      data[2][strcspn(data[2], "\n")] = '\0';

      printf("\n %s\n %s\n %s\n", data[0], data[1], data[2]);

      insertNodeAVL(avl, data);
      insertOrdList(list, data);

   }

   fclose(file);
   return 1;
}

// Função que escreve os dados contidos na árvore AVL em um arquivo.
int writeArchiveAVL(char arq[PATH_MAX], AVL* avl) {
   pid_t pid = fork();

   if (pid == -1) {
      printf("Erro na execucao da funcao.\n");
      return -1;
   } else if (pid == 0) {
      if (!freopen(arq, "w", stdout))
         return 0;
```

```
      printAVL(avl);
      exit(0);

   } else
      while (wait(NULL) > 0);

   return 1;
}

// Dada uma palavra (word), verifica se ela esta no dicionário e
retorna suas informações.
// A consulta é feita na AVL e na Lista. O tempo da consulta em cada
caso é impresso em stdout.
int searchWord(AVL* avl, List list, char* word) {
   struct timespec start, end;
   double timeAVL, timeList;

   clock_gettime(CLOCK_REALTIME, &start);
   char** dataList = queryList(list, word);
   clock_gettime(CLOCK_REALTIME, &end);
   timeList = (end.tv_sec - start.tv_sec) + (end.tv_nsec -
start.tv_nsec) / BILLION;

   clock_gettime(CLOCK_REALTIME, &start);
   char** dataAVL = queryAVL(avl, word);
   clock_gettime(CLOCK_REALTIME, &end);
   timeAVL = (end.tv_sec - start.tv_sec) + (end.tv_nsec -
start.tv_nsec) / BILLION;

   // Verifica se encontrou o dado e obteve o mesmo resultado na Lista
e na AVL (garantia).
   dataAVL != NULL && strcmp(dataAVL[0], dataList[0]) == 0 ?
      printf("\n %s\n %s\n %s\n", dataAVL[0], dataAVL[1], dataAVL[2]) :
      printf("Palavra não foi encontrada no dicionario.\n");

   printf(
      "\nO tempo da consulta na AVL foi %lf segundos.\n"
      "O tempo da consulta na Lista foi %lf segundos.\n",
      timeAVL, timeList
   );

   return 1;
}

// Dada uma frase (phrase), busca na árvore entradas contidas nela.
int searchPhraseAVL(AVL* avl, char* phrase) {
   int r;
```

```c
    char** words = split_string(phrase, &r);

    if (r < 1) {
        printf("Erro ao buscar frase.");
        return 0;
    }

    int i = 0;
    char** m;
    while (words[i] != NULL) {
        m = queryAVL(avl, words[i]);

        if (m != NULL)
            printf("\n %s\n %s\n %s\n", m[0], m[1], m[2]);

        i++;
    }

    return 1;

}

void dicionaryReport(AVL* avl) {
    // Cada índice do vetor representa uma letra do alfabeto: 0 == a; 25
== z.
    int count[26];
    for (size_t i = 0; i < 26; i++) count[i] = 0;
    countLettersAVL(avl, count);

    printf("\nRelatorio do dicionario:\n");

    //
    for (size_t i = 0; i < 26; i++)
        if (count[i] > 0) printf("%d palavra(as) com a letra %c\n",
count[i], i + 97);

}

// Dado uma frase (phrase), retorna um vetor de strings contendo todas
as palavras contidas nela.
// As palavras são separadas por " ".
// r returna um valor de erro -> r == 1: ok; r != 1: erro.
char** split_string(char* phrase, int* r) {
    *r = 1;
    // De inicio alocamos o vetor para conter até 10 strings.
    size_t bufsize = 10;
    char** words = malloc(bufsize * sizeof(char*));
```

```
   if (!words) {
      printf("split_string: Erro na alocacao de memoria.\n");
      *r = -1;
   }
   // A função strtok() divide a string phrase em diversos tokens
separados pelo caractere " ".
   char* token = strtok(phrase, " ");

   size_t i = 0;
   while (token != NULL) {
      // Enquanto existir um token, armazenamos ele em um índice do
vetor.
      // Tratamento para que palavras no final de frases sejam
interpretadas (são seguidas por '.');
      token[strlen(token) - 1] = token[strlen(token) - 1] == '.' ? '\0'
: token[strlen(token) - 1];
      words[i] = token;
      i++;
      // Caso 10 índices não sejam o suficiente, alocamos mais 10.
      if (i >= bufsize) {
         bufsize += 10;
         words = realloc(words, bufsize * sizeof(char*));

         if (!words) {
            printf("split_string: Erro na alocacao de memoria.\n");
            *r = -1;
         }
      }
      token = strtok(NULL, " ");
   }
   // Garantimos que o último índice do vetor seja NULL.
   words[i] = NULL;
   return words;
}

// Coloca toda uma string em letras minúsculas.
int toLower(char* str, size_t len) {
   if (len <= 0)
      return 0;
   for (size_t i = 0; i < len; i++)
      str[i] = str[i] >= 'A' && str[i] <= 'Z' ? str[i] | 0x60 : str[i];

   return 1;
}
```

Fonte: autores deste relatório

4 CONCLUSÃO

Ambas as estruturas utilizadas na implementação do programa (Lista e Árvore AVL) são eficientes e trazem bons resultados. Entretanto, a árvore AVL será sempre mais eficiente na busca de elementos, pois não precisa percorrer sequencialmente todos os elementos até chegar no local desejado.

Um árvore AVL entrega tempo de execução em O(log n) em todos os casos – inserção, busca, remoção – enquanto uma Lista ordenada realiza as mesmas operações em O(n). Esse tempo de execução aprimorado da árvore acontece graças ao balanceamento que é feito durante as operações de inserção e remoção.

Árvores AVL são ótimas para aplicações onde a busca é corriqueira, enquanto não há necessidade de eficiência máxima na inserção. As inserções em uma árvore AVL consomem tempo para realizar o balanceamento, em troca de obter buscas mais eficientes. Se for determinado que a inserção de dados deve ser mais eficiente, utiliza-se uma árvore rubro-negra.

A seguir, duas imagens comparando tempo de busca por duas palavras no dicionário do programa implementado: abraço, que no caso era primeira palavra do dicionário, e gafar que era a última. Note nas imagens que, apesar de o tempo da consulta ser mais rápido na lista no primeiro caso, provavelmente devido à alguma peculiaridade na implementação, no segundo caso temos uma grande vantagem no uso da árvore.

Figura 1 – Consulta da palavra abraço no dicionário

```
 - Menu Arvore AVL Dicionario -
1:Carregar dados a partir de arquivo
2:Buscar palavra no dicionario
3:Inserir dados no dicionario
4:Remover palavra do dicionario
5:Exibir significados dado frase
6:Exibir Relatório
7:Salvar dicionario em arquivo
8:Encerrar
 → 2

Palavra que deseja consultar: abraço

 abraço
 substantivo
 Ação de envolver algo ou alguém com os braços

O tempo da consulta na AVL foi 0.000004 segundos.
O tempo da consulta na Lista foi 0.000003 segundos.
```

Fonte: autores deste relatório

Figura 2 – Consulta da palavra gafar no dicionário

```
 - Menu Arvore AVL Dicionario -
1:Carregar dados a partir de arquivo
2:Buscar palavra no dicionario
3:Inserir dados no dicionario
4:Remover palavra do dicionario
5:Exibir significados dado frase
6:Exibir Relatório
7:Salvar dicionario em arquivo
8:Encerrar
 → 2

Palavra que deseja consultar: gafar

 gafar
 v. t.
 Contagiar com gafa.#

O tempo da consulta na AVL foi 0.000002 segundos.
O tempo da consulta na Lista foi 0.000049 segundos.
```

Fonte: autores deste relatório

5 REFERÊNCIAS

MARIAADRIANAAD1. Repositório Árvore AVL. **replit**, 2021.
Disponível em: https://replit.com/@MariaAdrianaAd1/ArvBinariaAVL, acessado em 22 out. 2021

Big-O Cheat Sheet. Common Data Structure Operations. **Site Big-O Cheat Sheet**, 2021.
Disponível em: https://www.bigocheatsheet.com/, acessado em 22 out. 2021

ASHOKGELAL. Difference between a LinkedList and a Binary Search Tree. **StackOverflow**, 2008.
Disponível em: https://stackoverflow.com/questions/270080/difference-between-a-linkedlist-and-a-binary-search-tree

RAJPUT, Abhishek. Red Black Tree vs AVL Tree. **GeeksforGeeks**, 2019.
Disponível em: https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/