

# Projeto 2 - Estruturas de Dados e Algoritmos I

## Grupo

| Nome                 | Email                   | Github                         |
|----------------------|-------------------------|--------------------------------|
| Eduardo Lima Ribeiro | eduardolimrib@gmail.com | <a href="#">@Eduardolimr</a>   |
| Samuel Barros Borges | samuelmordred@gmail.com | <a href="#">@SamuelMordred</a> |

## Diário

| Dia        | Atividade                                                                                                                     | Observações                                                      |
|------------|-------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|
| 23/11/2017 | Commit inicial do projeto, iniciação da pesquisa.                                                                             | Criação da estrutura de árvore AVL e de hash.                    |
| 25/11/2017 | Implementação das funções de hash e de árvore AVL.                                                                            | Função de comparação AVL ainda incompleta, hash possui colisões. |
| 26/11/2017 | Segunda tentativa para auto balanceamento da AVL e primeira versão funcional do código da AVL atingido na terceira tentativa. | -                                                                |
| 28/11/2017 | Começo da implementação da estrutura AVL no código de corretor ortográfico.                                                   | Colisões no código hash ainda presentes.                         |
| 29/11/2017 | Término da implementação da AVL no corretor ortográfico e hash.                                                               |                                                                  |
| 30/11/2017 | Refatoração das colisões de hash                                                                                              |                                                                  |
| 31/11/2017 | Finalização das colisões de hash                                                                                              |                                                                  |

## Objetivo

O objetivo deste projeto é utilizar de duas formas de estruturas de dados diferentes (Árvores AVL e Hashtables) para duas implementações de um corretor ortográfico em C.


## Requisitos

- Árvores
- Árvores de busca
- Alocação dinâmica
- Ponteiros
- Manuseio de arquivos
- Hashtables
- Manuseio de strings


# Projeto

Foram implementadas três as soluções tanto em árvores AVL como em hashtable, havendo uma diferença de performance entre as duas, a ser destacada à seguir.

alice.txt

|                                                                                         | AVL  | Hash | Otimo |
|-----------------------------------------------------------------------------------------|------|------|-------|
| Carga Dicionario                                                                        | 0.15 | 0.04 | 0.03  |
| Check do Arquivo                                                                        | 0.02 | 0.01 | 0.02  |
| Calculo Tamanho                                                                         | 0    | 0    | 0.01  |
| Limpeza Memória                                                                         | 0.01 | 0.02 | 0.02  |
|  Total | 0.18 | 0.07 | 0.07  |

holmes.txt

|                                                                                           | AVL  | Hash | Otimo |
|-------------------------------------------------------------------------------------------|------|------|-------|
| Carga Dicionario                                                                          | 0.12 | 0.05 | 0.03  |
| Check do Arquivo                                                                          | 1    | 0.35 | 0.31  |
| Calculo Tamanho                                                                           | 0    | 0    | 0     |
| Limpeza Memória                                                                           | 0.01 | 0.02 | 0.02  |
|  Total | 1.13 | 0.42 | 0.36  |

Como pode-se perceber, o programa utilizando hashtables tem uma performance bem melhor, o que já era esperado pois ele não precisa fazer busca alguma durante a sua execução, enquanto a árvore AVL se torna menos eficiente e rápido quanto maior o arquivo texto testado. A solução "ótima" foi alcançada utilizando uma função hash desenvolvida por terceiros tendo como objetivo, rapidez, poucas colisões, e alta eficiência na aplicação a strings. Ainda assim, o aumento de eficiência no programa em que ela foi utilizada foi quase imperceptível.

## Explicações

### Árvore AVL

Para a solução utilizando AVL foi utilizada uma **struct** Node:

```
~~~~struct Node{ char nome[45]; struct Node *left; struct Node *right; int height; };
```

Além disso, foi utilizada uma função de comparação alfabética para a inserção das palavras do dicionário na estrutura, e para a busca de possíveis palavras da estrutura:

```
## Comparação Alfabética
```

```
int compare(char str1, char *str2){ while ( *str1 != '\0' && *str1 == *str2 ) { ++str1; ++str2; } return (str1 - *str2); }
```

Utilizando a comparação alfabética se realiza a inserção da mesma forma que se realizaria numa árvore binária de busca:

```
if (node == NULL) return(newNode(palavra));
```

```
if (compare(palavra, node->nome)<0){
    node->left = insert(node->left, palavra);
}
else if (compare(palavra, node->nome)>0){
    node->right = insert(node->right, palavra);
}
else { // Equal keys are not allowed in BST
    return node;
}
```

A altura dos nós da árvore é definido pelas funções a seguir:

Função para achar o maior entre dois números:

```
int max(int a, int b){ if(a>b){ return a; } return b; }
```

Função para retornar a altura:

```
int height(struct Node *N) { if (N == NULL){ return 0; } return N->height; }
```

Então, a altura de cada nó é estabelecida no final de cada iteração de inserção pelo comando a seguir:

```
node->height = 1 + max(height(node->left), height(node->right));
```

## ## Rotações

Após a inserção dos nós, é realizado o balanceamento das árvores se necessário. A checagem do fator de balanceamento é realizado pela função a seguir:

```
int getBalance(struct Node *N) { if (N == NULL){ return 0; } return height(N->left) - height(N->right); }
```

As rotações são implementadas utilizando as funções a seguir:

```
struct Node rightRotate(struct Node *y){ struct Node no; no = y->left; y->left = no->right; no->right =
```

y;

```
// Update heights
y->height = max(height(y->left), height(y->right))+1;
no->height = max(height(no->left), height(no->right))+1;

// Return new root
return no;
```

}

```
struct Node leftRotate(struct Node *y) { struct Node no; no = y->right; y->right = no->left; no->left = y;
```

```
// Update heights y->height = max(height(y->left), height(y->right))+1; no->height =
max(height(no->left), height(no->right))+1;
```

```
// Return new root return no; }
```

Com os casos das rotações duplas resolvidos da seguinte maneira:

```
if (balance > 1 && compare(palavra, node->left->nome)<0){ return rightRotate(node); } // Right Right
Case if (balance < -1 && compare(palavra, node->right->nome)>0){ return leftRotate(node); } // Left
Right Case if (balance > 1 && compare(palavra, node->left->nome)>0) { node->left =
leftRotate(node->left); return rightRotate(node); }
```

```
// Right Left Case
if (balance < -1 && compare(palavra, node->right->nome)<0)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```

A contagem de elementos do dicionário é realizada após a inserção de todos os elementos na árvore pela função recursiva a seguir:

```
int NumDicio = 0; //variavel declarada no global
```

```
int inOrder(struct Node *root){ if(root != NULL){ NumDicio++; inOrder(root->left);
inOrder(root->right); } return NumDicio; }
```

A AVL por ser uma estrutura que necessita de alocação dinâmica necessitou de funções recursiva para a liberação de memória ao fim do programa. Elas serão exibidas a seguir:

```
struct Node *destroiArvore(struct Node *raiz){ if(raiz->right != NULL){ raiz->right =
destroiArvore(raiz->right); } if(raiz->left != NULL){ raiz->left = destroiArvore(raiz->left); } free(raiz);
return NULL; }
```

```
/* Descarrega dicionario da memoria. Retorna true se ok e false se algo deu errado */ bool
descarregaDicionario(void) { raiz = destroiArvore(raiz); if(raiz==NULL){ return true; } return false; }
```

```
## Hashtable
Para a implementação da hashtable, foi utilizada uma variável global do tipo
**struct** palavra:

~~~#define TAM_DICIO 1000003
typedef struct palavra{
    char *palavra;
    struct palavra *prox;
    struct palavra *ant;
}palavra;

palavra dicionario[TAM_DICIO] = { [0 ... 1000002] = NULL };
```

Onde se inicializa o vetor dicionario como **NULL** e **TAM\_DICIO** é o tamanho dado ao vetor dicionário, no caso um número primo de ordem elevada para acomodar a função hash sem muitas colisões.

Para a implementação em hashtables, foi utilizada a função **DEKHash** pois em testes com diversos arquivos e tamanhos de vetor, foi a que resultou em menos colisões e maior performance quando usada a divisão inteira pelo tamanho do dicionário.

## Inserção na memória do dicionário

```
/* Procedimento de inserção caso a posição da hashtable já esteja ocupada. */
void insereNaoNula(palavra *dic, char *temp){
    palavra *ant, *novo, *p;
    int cont;

    cont = 0;
    p = dic;
    novo = (palavra *) malloc (sizeof(palavra));
    novo->palavra = (char *) malloc (sizeof(char)*TAM_MAX);
    do{
        ant = p;
        p = p->prox;
        ant->prox = p;
        if(p != NULL){
            p->ant = ant;
        }
    }while(p != NULL);
    strcpy(novo->palavra, temp);
```

```

    ant->prox = novo;
    novo->ant = ant;
} /* fim-insereNaoNula */

```

```

/* Carrega dicionario na memoria. Retorna true se sucesso; senao retorna false. */
bool carregaDicionario(){
    int i;
    FILE *fd;
    char *temp;
    palavra *novo, *p;

    fd = fopen(NOME_DICIONARIO, "r");
    if(fd != NULL){
        temp = (char *) malloc (sizeof(char)*TAM_MAX);
        while(fgets(temp, TAM_MAX, fd)){
            temp[strlen(temp)-1] = '\0';
            i = DEKHash(temp, strlen(temp));
            if(dicionario[i].palavra != NULL){
                p = &dicionario[i];
                insereNaoNula(p, temp);
            }
            else if (dicionario[i].palavra == NULL){
                dicionario[i].palavra = (char *) malloc (sizeof(char)*TAM_MAX);
                strcpy(dicionario[i].palavra, temp);
                dicionario[i].prox = NULL;
                dicionario[i].ant = NULL;
            }
        }
        free(temp);
        fclose(fd);
        return true;
    }
    return false;
} /* fim-carregaDicionario */

```

O carregamento do dicionário na memória segue a lógica de que, caso se consiga ler o arquivo, se o vetor de índice igual ao hash da palavra estiver vazio, insere-se normalmente. Caso o contrário, é usado o método de *chaining* por meio de listas encadeadas e alocada uma nova palavra na próxima posição disponível para o vetor de índice hash.

## Confere palavra

```

/* Retorna true se a palavra esta no dicionario. Do contrario, retorna false */
bool conferePalavra(const char *pal) {
    palavra *p;
    int i;

    i = DEKHash(pal, strlen(pal));
    p = &dicionario[i];
    if(dicionario[i].palavra != NULL){
        if(!strcmp(dicionario[i].palavra, pal)){
            return true;
        }
    }
}

```

```

    else{
        do{
            p = p->prox;
            if(p!= NULL && !strcmp(p->palavra, pal)){
                return true;
            }
        }while(p != NULL);
    }
}
return false;
} /* fim-conferePalavra */

```

A função de conferir palavra checa se a palavra dentro do vetor de índice hash é nula, se não for, realiza a checagem, caso o contrário, retorna falso. Durante a checagem, ele percorre e compara a palavra com todas as palavras existentes na lista encadeada; se não achar nenhuma dessa maneira ele sai do loop pois p irá apontar para nulo e retornará falso.

## Conta palavras

```

/* Retorna qtde palavras do dicionario, se carregado; senao carregado retorna zero
*/
unsigned int contaPalavrasDic(void) {
    FILE *fd;
    char temp[TAM_MAX];
    int i;

    i = 0;
    fd = fopen(NOME_DICIONARIO, "r");
    if(fd != NULL){
        while(fgets(temp, TAM_MAX, fd)){
            i++;
        }
        fclose(fd);
        return i;
    }
    return 0;
} /* fim-contaPalavrasDic */

```

A função de contagem de palavras é simples, ela simplesmente percorre o arquivo dicionário e incrementa o contador a cada iteração, retornando o valor ao final da função.

## Descarrega dicionário

```

/* Descarrega dicionario da memoria. Retorna true se ok e false se algo deu errado
*/
bool descarregaDicionario(void) {
    FILE *fd;
    int i;
    char temp[TAM_MAX];
    palavra *p, *ant;

    fd = fopen(NOME_DICIONARIO, "r");

```

```

if(fd != NULL){
    while(fgets(temp, TAM_MAX, fd)){
        i = DEKHash(temp, strlen(temp));
        if(dicionario[i].palavra != NULL){
            p = &dicionario[i];
            do{
                ant = p;
                p = p->prox;
                free(ant->palavra);
            }while(p != NULL);
        }
    }
    return true;
}
return false;
} /* fim-descarregaDicionario */

```

A função de descarregamento percorre o arquivo novamente, pois não é possível percorrer o vetor de hash simplesmente incrementando o índice de modo sequencial. Ao chegar em cada palavra, é percorrida a lista encadeada e dado um *free()* para a palavra e então para cada nó.

## Falhas/Limitações

O programa apresentava falhas na hora de ser rodado, retornando menos palavra erradas do que deveria. Ao examinar o arquivo dicionário o foi encontrada uma palavra estourava o limite de 45 letras. Ao aumentar o limite de 45 letras para 50 letras o programa passou a funcionar corretamente.

## Opiniões

O trabalho foi importante para o aprendizado dessas estruturas de dados e para a formação de uma nova perspectiva sobre a eficiência que elas proporcionam, além de dar uma ótima lição em trabalho em grupo e divisão de tarefas para um projeto.

## Referências

<http://www.cse.yorku.ca/~oz/hash.html> (Hash ótima para strings, DJB2)

<http://www.geeksforgeeks.org/avl-tree-set-1-insertion/> (Site de referência para a construção da AVL)