# Universidade Federal de Uberlândia

Faculdade de computação

Curso de Ciências da computação

**Maycon Douglas Batista Dos Santos 11921BSI209**

**&**

**Sávio Dias Araújo BSI 11921BSI234**

**Estudo sobre árvores AVL de busca aplicado ao dicionário de palavras**

Resumo

Este trabalho tem como função pesquisar e implementar um programa em linguagem C afim de simular um dicionário implementando na estrutura de uma árvore AVL tendo como chave a palavra e guardando também a classificação e o significado fim de reduzir principalmente a o tempo de busca.

Palavras chaves: **Árvore AVL, Estrutura de dados tipo dicionário.**

# 1 Introdução

## 1.1 O que é árvores AVL

Árvore AVL é uma árvore binária de busca balanceada, ou seja, uma árvore balanceada são as árvores que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrências idênticas

## 1.2 complexidade de uma árvore avl

As operações de busca, inserção e remoção de elementos possuem complexidade $O\log(N)$(no qual N é o número de elementos da árvore), que são aplicados a árvore de busca binária.

# 2 Lógica e Estrutura do programa

## 2.1 Estrutura

O programa foi estruturado com 4 pastas dentro do arquivo **trabTree** são elas:

- Árvore onde temos todos os arquivos o que contem todas as funções do tipo árvore avl;
- Dados onde ficam as estruturas de entrada e saída para arquivos alem dos arquivos textos a serem lidos;
- Fila que contes uma estrutura de dado auxiliar do tipo fila para uso dentro das funções de percurso dentro da nossa árvore avl;
- lista a nossa outra estrutura do tipo lista que precisaremos usar afim de comparação para podermos ter um pouco mais noção da diferença de tempo gasto entre a árvore a a lista
- Retornos.h onde ficam todos as variáveis auxiliares que nos ajudam a entender o que esta acontecendo e os dados usados pela avl e lista;
- Teste.h com algum funcionas prontas para testar determinadas funções das nossas estrutura de dados;
- main.c onde realmente começa nosso programa;
- MAKEFILE estrutura pra facilitar a compilação do programa.

## 2.2 **Códigos estrutura avl**

### 2.2.1 **construtor e estruturas do tipo avl**

Aqui guardamos os dados da nossa estrutura no campo informação temos a nosso campo chave que é usada para comparação, altura e os ponteiros para esquerda e direita: **struct AVL_Node**

```
{
	char chave[SIZE_CHAR];
	struct DADO infor;
	int altura;

	struct AVL_Node *esq;
	struct AVL_Node *dir;
};

AVL_Tree* constructor_AVL_Tree (void)
{
	AVL_Tree *raiz = (AVL_Tree*) malloc (sizeof(AVL_Tree));

	if(raiz != NULL)
	{
		*(raiz) = NULL;
	}
return raiz;
}
```

### 2.2.2 **Funções auxiliares**

```
int altura_node (struct AVL_Node *node)

{
	if(node == NULL)
		return INVALID_POINTER;
	else
		return node->altura;
}

int fator_balanceamento_node (struct AVL_Node *node)
{
	return labs(altura_node(node->esq) - altura_node(node->dir));
}

int maior (int x,int y){return x > y ? x : y; }
```

### 2.2.3 **funções de rotação**

```
void Rotacao_SimpDir (AVL_Tree *A)
{
	struct AVL_Node *B;

	B = (*A)->esq;
```

```
        (*A)->esq = B->dir;

        B->dir = *A;

        (*A)->altura = maior(altura_node((*A)->esq), altura_node((*A)->dir)) +1;

        B->altura = maior(altura_node(B->esq), (*A)->altura) +1;

        (*A) = B;
} // LL

void Rotacao_SimpEsq (AVL_Tree *A)
{
        struct AVL_Node *B;
        B = (*A)->dir;

        (*A)->dir = B->esq;

        B->esq = (*A);

        (*A)->altura = maior (altura_node((*A)->esq), altura_node((*A)->dir)) +1;

        B->altura = maior (altura_node(B->dir), (*A)->altura) +1;

        (*A) = B;
} // RR

void Rotacao_DupDir (AVL_Tree *A)
{
        Rotacao_SimpEsq (&(*A)->esq);
        Rotacao_SimpDir (A);
} // LR

void Rotacao_DupEsq (AVL_Tree *A)
{
        Rotacao_SimpDir (&(*A)->dir);
        Rotacao_SimpEsq (A);
} // RL
```

2.2.4 **Função de inserção**

A inserção e dividida em duas partes primeira recursivamente procurar o local correto de inserção movendo de através dos ponteiros de esquerda e direita, logo após temos o backtracking ajustando nossa avl atualizando a altura e rotacionando para que ela continue balanceada.

```
int insere_AVL_Tree (AVL_Tree *raiz, char chave[],struct DADO infor)

{
        int retorno;
```

```
if(*raiz == NULL)
{
	struct AVL_Node *novo = (struct AVL_Node*) malloc (sizeof(struct AVL_Node));

	if(novo == NULL) return OUT_NOT_MEMORY;

	strcpy(novo->chave, chave);
	novo->infor = infor;
	novo->esq = NULL;
	novo->dir = NULL;
	novo->altura = 0;
	*raiz = novo;

	return SUCCESS;
}

struct AVL_Node *atual = *raiz;

if(strcasecmp(chave, atual->chave ) < 0)
{
	retorno = insere_AVL_Tree(&(atual->esq), chave, infor);

	if(retorno == 1)
	{
		if(fator_balanceamento_node (atual) >= 2)
		{
			if(strcasecmp(chave, (*raiz)->esq->chave) < 0)
			{
				Rotacao_SimpDir (raiz); // ll simples direita
			}
			else
			{
				Rotacao_DupDir (raiz); // lr dupla direita
			}
		}
	}
}
else if(strcasecmp(chave, atual->chave) > 0)
{
	retorno = insere_AVL_Tree(&(atual->dir), chave, infor);

	if(retorno == 1)
	{
		if(fator_balanceamento_node(atual) >= 2)
		{
			if(strcasecmp(chave, (*raiz)->dir->chave) > 0)
			{
				Rotacao_SimpEsq (raiz); // rr simpes esquerda
			}
			else
			{
```

```
                                        Rotacao_DupEsq (raiz); // rl dupla direita
                                }
                        }
                }
        }
        else
        {
                return INVALID_OPERATION;
        }
        atual->altura = maior(altura_node(atual->esq), altura_node(atual->dir)) + 1;

        return retorno;
        }
```

### 2.2.5 **função de busca**

A função de busca eu recebo uma chave e percorro minha árvore avl se minha chave for menor que meu elemento atual eu vou para esquerda, se não se for maior eu vou para direita, se não ou seja igual eu retorno ele, mais se ele passou por todos não existe mais nó e não encontrou e por que aquela chave não existe.

```
struct DADO busca_AVL_Tree (const AVL_Tree *tree, char chave[])

{
        struct DADO dados;
        strcpy(dados.palavra, "Null\n\0");
        strcpy(dados.classificacao, "Null\n\0");
        strcpy(dados.significado, "Null\n\0");

        if(tree != NULL && (*tree) != NULL)
        {
                struct AVL_Node *aux = *(tree);

                while(aux != NULL)
                {
                        if(strcasecmp(chave, aux->chave) < 0)
                        {
                                aux = aux->esq;
                        }
                        else if(strcasecmp(chave, aux->chave) > 0)
                        {
                                aux = aux->dir;
                        }
                        else
                        {
                                dados = aux->infor;
                                break;
                        }
                }
        }
```

```
        return dados;
    }
```

### 2.2.6 **funções de remoção**

A remoção em árvores AVL é similar à remoção em uma árvore binária de busca. Entretanto, pode ser necessário o rebalanceamento da parte afetada pela remoção.

1. Se p possui nenhum filho: remover p, rebalancear nós antecessores afetados, se necessário.
2. Se p possui um filho: subir este filho, remover p, rebalancear antecessores, se necessário.
3. Se p possui dois filhos:
    1. Encontrar o menor na subárvore à direita e substituir seu valor em p;
    2. Executar a remoção do menor na subárvore à direita de p (este nó possuirá um ou nenhum filho);
    3. Após a remoção do menor à direita, rebalancear p, caso necessário.

```
struct AVL_Node* procura_menor (struct AVL_Node *node)

{
    struct AVL_Node *aux = node;

    if(node != NULL)
    {
        while(aux->esq != NULL)
        aux = aux->esq;
    }

    return aux;
}

int remove_AVL_Tree (AVL_Tree *raiz, char chave[])
{
    if(raiz == NULL || (*raiz) == NULL) return OUT_NOT_RANGE;

    int resposta;

    if(strcasecmp(chave, (*raiz)->chave) < 0)
    {
        resposta = remove_AVL_Tree (&(*raiz)->esq, chave);

        if(resposta == SUCCESS)
        {
            if(fator_balanceamento_node(*raiz) >= 2)
            {
                if(altura_node((*raiz)->dir->esq) <= altura_node((*raiz)->dir->dir))
                {
```

```
                                Rotacao_SimpEsq (raiz);
                        }
                        else
                                Rotacao_DupEsq (raiz);
                }
        }
}
else if(strcasecmp(chave, (*raiz)->chave) > 0)
{
        resposta = remove_AVL_Tree(&(*raiz)->dir, chave);

        if(resposta == SUCCESS)
        {
                if(fator_balanceamento_node(*raiz) >= 2)
                {
                        if(altura_node((*raiz)->esq->dir) <= altura_node((*raiz)->esq->esq))
                        {
                                Rotacao_SimpDir(raiz);
                        }
                        else
                                Rotacao_DupDir(raiz);
                }
        }
}
else
{
        if((*raiz)->esq == NULL || (*raiz)->dir == NULL)
        {
                struct AVL_Node *aux = (*raiz);

                if((*raiz)->esq != NULL)
                {
                        (*raiz) = (*raiz)->esq;
                }
                else
                        (*raiz) = (*raiz)->dir;
                free(aux);
        }
        else
        {
        struct AVL_Node *aux = procura_menor((*raiz)->dir);

        strcpy((*raiz)->chave, aux->chave);

        remove_AVL_Tree(&(*raiz)->dir, (*raiz)->chave);

        if(fator_balanceamento_node(*raiz) >= 2)
        {
        if(altura_node((*raiz)->esq->dir) <= altura_node((*raiz)->esq->esq))
        {
                Rotacao_SimpDir(raiz);
```

```
                }
                else
                        Rotacao_DupDir(raiz);
                }
        }

                if((*raiz) != NULL) (*raiz)->altura = maior(altura_node((*raiz)->esq), altura_node((*raiz)-
        >dir)) +1;

                return SUCCESS;
        }

        (*raiz)->altura = maior(altura_node((*raiz)->esq), altura_node((*raiz)->dir)) +1;

        return resposta;
}
```

### 2.2.7 funções de percurso

**Pré ordem:**

```
void pre_ordem_AVL(const AVL_Tree *raiz, FILA *fila)

{
        if(raiz != NULL && (*raiz) != NULL && fila != NULL)
        {
                push_FILA(fila, (*raiz)->infor);
                in_ordem_AVL(&(*raiz)->esq, fila);
                in_ordem_AVL(&(*raiz)->dir, fila);
        }
}
```

**In ordem:**

```
void in_ordem_AVL(const AVL_Tree *raiz, FILA *fila)
{
        if(raiz != NULL && (*raiz) != NULL && fila != NULL)
        {
                in_ordem_AVL(&(*raiz)->esq, fila);
                push_FILA(fila, (*raiz)->infor);
                in_ordem_AVL(&(*raiz)->dir, fila);
        }
}
```

**Pós ordem:**

```
void pos_ordem_AVL(const AVL_Tree *raiz, FILA *fila)
{
```

```
	if(raiz != NULL && (*raiz) != NULL && fila != NULL)
	{
		in_ordem_AVL(&(*raiz)->esq, fila);
		in_ordem_AVL(&(*raiz)->dir, fila);
		push_FILA(fila, (*raiz)->infor);
	}
}
```

### 2.2.8 destrutor e funções extras

**void exibir_AVL_Tree (const AVL_Tree *raiz)**

```
{
	if(raiz == NULL)
		return;
	if(*raiz != NULL)
	{
		exibir_AVL_Tree (&((*raiz)->esq));

		printf("\n+++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n");
		printf("chave: %s",(*raiz)->infor.palavra);
		printf("class: %s",(*raiz)->infor.classificacao);
		printf("signi: %s",(*raiz)->infor.significado);

		exibir_AVL_Tree (&((*raiz)->dir));
	}
}// in ordem
```

**void limpa (struct AVL_Node *node)**

```
{
	if(node != NULL)
	{
		limpa(node->esq);
		limpa(node->dir);
		free(node);
		node = NULL;
	}
}// valgrind okay
```

**void descructor_AVL_Tree (AVL_Tree *raiz)**

```
{
	if(raiz != NULL)
	{
		limpa(*raiz);
		free(raiz);
		raiz = NULL;
	}
}// valgrind okay
```

## 2.3 Códigos estrutura tipo lista

### 2.3.1 construtor, destrutor e estruturas

```
struct Node_List
{
	struct DADO info;
	struct Node_List *prox;
};


struct TADList
{
	struct Node_List *head;
	unsigned int size;
};


struct Node_List* constructor_Node_List (struct DADO info)
{
	struct Node_List* newnode = (struct Node_List*) malloc (sizeof(struct Node_List));

	if(newnode != NULL )
	{
		newnode->info = info;
		newnode->prox = NULL;
	}

	return newnode;
}


struct TADList* constructor_TADList (void)
{
	struct TADList *list = (struct TADList*) malloc (sizeof(struct TADList));

	if(list != NULL )
	{
		list->head = NULL;
		list->size = 0;
	}

	return list;
}


void clear_List (struct TADList *list)
{
	if(list != NULL)
	{
		struct Node_List *ant = NULL, *atual = list->head;

		while (atual != NULL)
```

```
		{
			ant = atual;
			atual = atual->prox;

			free(ant);
		}
		list->size = 0;
		list->head = NULL;
	}
}

void desctructor_TADList (struct TADList *list)
{
	clear_List(list);

	if(list != NULL)
	{
		free(list);
		list = NULL;
	}
}
```

### 2.3.2 função de inserção

Basicamente o que temos que fazer é procurar em que lugar da lista será inserido o novo elemento (no caso, iremos ordenar pelo campo palavra), percorrendo os ponteiros e assim que que encontrar o local corretos ajustar os ponteiros afim de encaixar o novo nó;

```
int insert_TADList (struct TADList *list, struct DADO info)
{
	if(list == NULL) return INVALID_POINTER;

	struct Node_List *newnode = constructor_Node_List(info);

	if(newnode == NULL) return OUT_NOT_MEMORY;
	else if(list->head == NULL) // ainda esta vazia
	{
		list->head = newnode;
	}
	else
	{
		struct Node_List *ant = NULL, *atual = list->head;

		while(atual != NULL)
		{
			if(strcasecmp(info.palavra, atual->info.palavra) < 0)
				break;
			else if(strcasecmp(info.palavra, atual->info.palavra) == 0)
			{
```

```
                        free(newnode);
                        return INVALID_OPERATION;
                }
                else
                {
                        ant = atual;
                        atual = atual->prox;
                }
        }

        if(atual == list->head)
        {
                newnode->prox = list->head;
                list->head = newnode;
        }
        else
        {
                newnode->prox = atual;
                ant->prox = newnode;
        }
}

list->size++;
return SUCCESS;
}
```

### 2.3.3 função de busca

```
struct DADO busca_list (const TADList *list, char chave[])
{
        struct DADO aux;
        strcpy(aux.palavra , "NULL\n\0");
        strcpy(aux.classificacao , "NULL\n\0");
        strcpy(aux.significado , "NULL\n\0");

        if(list != NULL)
        {
                struct Node_List *node = list->head;

                while(node != NULL)
                {
                        if(strcasecmp(chave, node->info.palavra) == 0)
                        {
                                aux = node->info;
                                break;
                        }
                        else if(strcasecmp(chave , node->info.palavra) > 0)
                                break;
                        else
```

```
                {
                        node = node->prox;
                }
        }
    }
    return aux;
}
```

### 2.3.4 função de remoção de um elemento indicado pela chave

Basicamente, o que temos que fazer é procurar esse elemento na lista e mudar os valores de alguns ponteiros:

```
int remove_TADList(struct TADList *list, char chave[])
{
    if(list == NULL) return INVALID_POINTER;
    else if(list->head == NULL)
        return INVALID_OPERATION;

    struct Node_List *ant = NULL, *aux = list->head;

    while(strcasecmp(chave, aux->info.palavra) != 0 && aux != NULL)
    {
        ant = aux;
        aux = aux->prox;
    }

    if(aux == NULL)return ELEMENT_NOT_FOUND;
    if(aux == list->head)
    {
        list->head = list->head->prox;
    }
    else
    {
        ant->prox = aux->prox;
    }

    free(aux);
    list->size--;
    return SUCCESS;
}// testar
```

### 2.3.4 **funções de tamanho e exibição**

```
int empty_TADList(struct TADList* list)
{
    if(list == NULL)
        return INVALID_POINTER;
    else if(list->size > 0)
        return FALSE;
    else
```

```
		return SUCCESS;
}

int size_TADList (struct TADList *list)
{
	if(list == NULL)
		return INVALID_POINTER;
	else
		return list->size;
}

void print_TADList (struct TADList *list)
{
	if(list != NULL)
	{
		struct Node_List *aux = list->head; // var auxiliares
		unsigned int contt = 1;
		printf("============================ List dictionary ====================\n");
		while(aux != NULL)
		{
			printf("\n+++++++++++++++++++++++++++++++->%d<-
++++++++++++++++++++++++++\n", contt++);
			printf("chave: %s",aux->info.palavra);
			printf("class: %s",aux->info.classificacao);
			printf("signi: %s",aux->info.significado);

			aux = aux->prox;
		}
	printf("================================= |end| ========================\n");
	}
}
```

## 2.4 Estrutura auxiliar tipo fila

### 2.4.1 Estruturas e Construtores

**struct FILA_Node**

```
{
	struct DADO info;
	struct FILA_Node *next;
};
```

**struct FILA**

```
{
	struct FILA_Node *begin;
	struct FILA_Node *end;
	unsigned int size;
};
```

```
struct FILA_Node* constructor_FILA_Node (struct DADO dado)
{
	struct FILA_Node *node = (struct FILA_Node*) malloc (sizeof(struct FILA_Node));

	if(node != NULL)
	{
		node->info = dado;
		node->next = NULL;
	}
	return node;
}

FILA* constructor_FILA (void)
{
	FILA *Fila = (FILA*) malloc (sizeof(FILA));

	if(Fila != NULL)
	{
		Fila->begin = NULL;
		Fila->end = NULL;
		Fila->size = 0;
	}
	return Fila;
}
```

### 2.4.2 **Destrutores**

```
void clear (struct FILA_Node *node)
{
	if(node != NULL)
	{
		clear(node->next);
		free(node);
		node = NULL;
	}
}

void clear_FILA (FILA *Fila)
{
	if(Fila != NULL)
	{
		clear(Fila->begin);
		Fila->begin = NULL;
		Fila->end = NULL;
		Fila->size = 0;
	}
}

void destructor_FILA (FILA *Fila)
{
```

```
	if(Fila != NULL)
	{
		clear_FILA(Fila);
		Fila = NULL;
	}
}
```

### 2.4.3 **Inserção**

Cada elemento e inserido no final:

```
int push_FILA (FILA *Fila, struct DADO dado)
{
	if(Fila == NULL) return INVALID_POINTER;

	struct FILA_Node *new = constructor_FILA_Node(dado);

	if(new == NULL) return OUT_NOT_MEMORY;

	if(Fila->begin == NULL && Fila->end == NULL)
	{
		Fila->begin = new;
		Fila->end = new;
	}
	else
	{
		Fila->end->next = new;
		Fila->end = new;
	}

	Fila->size++;
	return SUCCESS;
}
```

### 2.4.4 **Remoção**

```
int remove_FILA (FILA *Fila)
{
	if(Fila == NULL) return INVALID_POINTER;
	else if (Fila->begin == NULL && Fila->end == NULL)
	{
		return OUT_NOT_RANGE;
	}
	struct FILA_Node *aux = Fila->begin;

	if(Fila->size == 1)
	{
		Fila->begin = NULL;
		Fila->end = NULL;
	}
	else
```

```
	{
		Fila->begin = Fila->begin->next;
	}

	free(aux);
	Fila->size--;
	return SUCCESS;
}
```

#### 2.4.5 **função de busca**

```
struct DADO front_FILA (FILA *Fila)
{
	struct DADO dado;

	strcpy(dado.palavra, "\0");
	strcpy(dado.classificacao, "\0" );
	strcpy(dado.significado, "\0");

	if(Fila == NULL || (Fila->begin == NULL && Fila->end == NULL)) return dado;

	dado = Fila->begin->info;

	return dado;
}
```

#### 2.4.6 **funções de tamanho**

```
int empty_FILA (FILA *Fila)
{
	if(Fila == NULL) return INVALID_POINTER;
	if(Fila->size > 0)
		return 0;
	else
		return SUCCESS;
}

int size_FILA (FILA *Fila)
{
	if(Fila == NULL) return INVALID_POINTER;
	else
		return Fila->size;
}
```

#### **2.4.7 função de exibição**

Essa tem como intuito acompanhar o que tá acontecendo na fila:

```
void exibir_FilA(FILA *Fila)
{
	if(Fila == NULL) return;
	struct FILA_Node *aux = Fila->begin;
```

```
	printf("====================== | exibindo fila | ======================\n\n");
	while(aux != NULL)
	{
		printf("_____________________________________________________________________\n");
		printf("chave -> %s\n",aux->info.palavra);
		printf("classificacão -> %s\n",aux->info.classificacao);
		printf("significado -> %s\n",aux->info.significado);

		aux = aux->next;
	}
	printf("========================== | end | ============================\n\n");
}
```

## 2.5 arquivo dados.c

Neste arquivo ficam todas as funções de leitura e escrita em arquivos .txt:

### 2.5.1 A função atualiza a avl e lista a partir de um arquivo

Exercício 1 pedido no trabalho : 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).

```
int read_dictionary (AVL_Tree *Tree, TADList *list)
{
	if(Tree == NULL || list == NULL) return INVALID_POINTER;

	FILE *arg = fopen("dados/dados.txt", "r");

	if(arg == NULL) return ERROR;

	struct DADO aux;
	char lixo[3];

	while(!feof(arg))
	{
		fgets(aux.palavra, SIZE_CHAR, arg);
		fgets(aux.classificacao, SIZE_CHAR, arg);
		fgets(aux.significado, SIZE_CHAR, arg);
		fgets(lixo, 3, arg);

		insere_AVL_Tree(Tree, aux.palavra, aux);
		insert_TADList(list, aux);
	}

	fclose(arg);
	return SUCCESS;
}
```

2.5.2 Funções de escrita em arquivos

É usada as funções normais de percurso só que ao invés de imprimir elas colocam os elementos em uma fila em pré ordem, in ordem ou pós ordem.

Exercício 2 do trabalho: Salvar o dicionário em um arquivo texto (percurso escolhido pelo usuário: em, pré ou pós-ordem) a partir da AVL:

```
int write_dictionary_pre (const AVL_Tree *tree)
{
	if(tree == NULL) return INVALID_POINTER;

	FILA *fila = constructor_FILA();

	if(fila == NULL ) return ERROR;

	FILE *arg = fopen("dados/preOrdem.txt", "w");

	if(arg == NULL) return ERROR;

	pre_ordem_AVL (tree, fila);

	while(!empty_FILA(fila))
	{

		fprintf(arg, "%s", front_FILA(fila).palavra);
		fprintf(arg, "%s", front_FILA(fila).classificacao);
		fprintf(arg, "%s", front_FILA(fila).significado);

		fprintf(arg, "%s\n", "&");

		remove_FILA(fila);
	}
	fprintf(arg, "%s", "FIM");

	destructor_FILA(fila);
	fclose(arg);
	return SUCCESS;
}



int write_dictionary_in (const AVL_Tree *tree)
{
	if(tree == NULL) return INVALID_POINTER;

	FILA *fila = constructor_FILA();

	if(fila == NULL ) return ERROR;
```

```
	FILE *arg = fopen("dados/inOrdem.txt", "w");

	if(arg == NULL) return ERROR;

	in_ordem_AVL(tree, fila);

	while(!empty_FILA(fila))
	{

		fprintf(arg, "%s", front_FILA(fila).palavra);
		fprintf(arg, "%s", front_FILA(fila).classificacao);
		fprintf(arg, "%s", front_FILA(fila).significado);

		fprintf(arg, "%s\n", "&");

		remove_FILA(fila);
	}
	fprintf(arg, "%s", "FIM");

	destructor_FILA(fila);
	fclose(arg);
	return SUCCESS;
}


int write_dictionary_pos (const AVL_Tree *tree)
{
	if(tree == NULL) return INVALID_POINTER;

	FILA *fila = constructor_FILA();

	if(fila == NULL ) return ERROR;

	FILE *arg = fopen("dados/posOrdem.txt", "w");

	if(arg == NULL) return ERROR;

	pos_ordem_AVL (tree, fila);

	while(!empty_FILA(fila))
	{

		fprintf(arg, "%s", front_FILA(fila).palavra);
		fprintf(arg, "%s", front_FILA(fila).classificacao);
		fprintf(arg, "%s", front_FILA(fila).significado);

		fprintf(arg, "%s\n", "&");

		remove_FILA(fila);
	}
	fprintf(arg, "%s", "FIM");
```

```
	destructor_FILA(fila);
	fclose(arg);
	return SUCCESS;
}
```

## 2.6 Retornos.h

Essa é só uma estrutura compartilhada pela avl, lista , fila:

```
#include <stdio.h>

#ifndef TRABRETORNOS_H
#define TRABRETORNOS_H

#define SIZE_CHAR 70

struct DADO
{
	char palavra[SIZE_CHAR];
	char classificacao[SIZE_CHAR];
	char significado[SIZE_CHAR];
};


#define INVALID_OPERATION -2
#define INVALID_POINTER -1
#define ERROR 0
#define FALSE 0
#define SUCCESS 1
#define OUT_NOT_MEMORY 2
#define OUT_NOT_RANGE 3
#define ELEMENT_NOT_FOUND 4

#endif
```

## 2.7 main.c

### 2.7.1 Menu principal

```
int menu_principal (void)
{
	system("clear");
	printf(">>>>>>>>>>>>>>>>>>>>>>>>>>>>>> MENU <<<<<<<<<<<<<<<<<<<<<<<<<<\n");
	printf("1 - Atualizer dicionarios (lista & avl)\n");
	printf("2 - busca (avl)\n");
	printf("3 - coparar tempos de busca da avl e da lista\n");
	printf("4 - Insert nova paravla (lista & avl)\n");
```

```
	printf("5 - remover uma palavra (lista & avl)\n");
	printf("6 - exibir todos os significados de uma frase (avl)\n");
	printf("7 - exibir relátorio\n");
	printf("8 - salvar em txt\n");
	printf("9 - imprime (avl & list)\n");
	printf("0 - sair\n");
	printf("entre com uma opção: ");

	int opc = 0;
	scanf("%d",&opc);
	return opc;
}
```

2.7.2 função de tratamento de erros usada para identificar o que esta acontecendo na nossa estrutura:

```
void tratamento(int resposta)
{
	switch (resposta)
	{
	case -2:
		printf("Operação invalida não existe ou uso incorreto\n");
		break;
	case -1:
		printf("Ponteiro para TAD invalido!\n");
		break;
	case 0:
		printf("False ou error!\n");
		break;
	case 1:
		printf("Operação finalizada com sucesso\n");
		break;
	case 2:
		printf("Falta de memoria\n");
		break;
	case 3:
		printf("Não encontrado!\n");
		break;
	default:
		break;
	}
}
```

2.7.3 **função pedida no exercício 2:** - Buscar uma entrada na AVL: exibir o significado de uma palavra e sua classificação

```
void func2 (const AVL_Tree *tree)
{
	struct DADO dado;
```

```
	printf("Entre com a palavra que deseja buscar: ");
	setbuf(stdin, NULL);
	fgets(dado.palavra, SIZE_CHAR, stdin);
	dado = busca_AVL_Tree(tree, dado.palavra);
	exibir_dado(&dado);
}
```

2.7.4 **função pedida do exercício** 2ª : Comparar a busca na AVL com a busca na lista exibindo os tempos

**Atenção: essa função esta com um bug por algum motivo alguns elementos não estão sendo encontrados pela função de busca da lista**

```
void func3 (const AVL_Tree *tree, const TADList *list)
{
	struct DADO dado;

	printf("entre com a chave que deseja buscar: ");
	setbuf(stdin, NULL);
	fgets(dado.palavra, SIZE_CHAR, stdin);

	clock_t inicio = clock();
	dado = busca_AVL_Tree(tree, dado.palavra);
	clock_t fim = clock();

	double tempo_tree = (double) (fim - inicio) / CLOCKS_PER_SEC;
	printf("Tempo gasto na busca pela arvore: %lf\n",tempo_tree);
	exibir_dado(&dado);

	inicio = clock();
	dado = busca_list(list, dado.palavra);
	fim = clock();

	double tempo_list = (double) (fim - inicio) / CLOCKS_PER_SEC;
	printf("Tempo gasto na busca pela arvore: %lf\n",tempo_list);
	exibir_dado(&dado);
}
```

2.7.5 **Função pedida no exercício : 3 - Inserir uma entrada (*palavra, classificação, significado*) na AVL / lista**

```
void func4 (AVL_Tree *tree, TADList *list)
{
	struct DADO dado;

	printf("Entre com a palavra que deseja inserir: ");
	setbuf(stdin, NULL);
	fgets(dado.palavra, SIZE_CHAR, stdin);
```

```
        printf("Entre com a classificação: ");
        setbuf(stdin, NULL);
        fgets(dado.classificacao, SIZE_CHAR, stdin);

        printf("Entre com o signicado: ");
        setbuf(stdin, NULL);
        fgets(dado.significado, SIZE_CHAR, stdin);

        tratamento(insere_AVL_Tree(tree, dado.palavra, dado));
        tratamento(insert_TADList(list, dado));
}
```

2.7.6 **função pedida do exercício : Remover uma entrada (*palavra, classificação, significado*) na AVL / lista**

**Atenção: essa função esta com um bug quando tento remover varias chaves repetidas em seguencia eu recebo segmentátion found da minha lista como mostrado no testelist3 em teste.h**

```
void func5 (AVL_Tree *tree, TADList *list)
{
        struct DADO dado;

        printf("entre com a chave que deseja remover: ");
        setbuf(stdin, NULL);
        fgets(dado.palavra, SIZE_CHAR, stdin);

        tratamento( remove_AVL_Tree(tree, dado.palavra));

        tratamento( remove_TADList(list, dado.palavra));
}
```

2.7.7 função pedida no exercício: 5 - Dada uma frase, exibir as palavras/significados que foram encontrados no dicionário a partir da AVL

```
void func6 (AVL_Tree *tree)
{
        printf("enre com a frase: ");
        setbuf(stdin, NULL);

        char frase[100], palavra[SIZE_CHAR];
        struct DADO aux;

        fgets(frase, 100, stdin);
```

```
	for(int i = 0, j = 0; i < strlen(frase); i++)
	{
		if(isalpha(frase[i]))
		{
			palavra[j++] = frase[i];
		}
		else if(frase[i] == '\0')
			break;
		else if(frase[i] == ' ' || frase[i] == '\n')
		{
			palavra[j++] = '\n';
			palavra[j] = '\0';
			j = 0;

			aux = busca_AVL_Tree(tree, palavra);

			printf("\n\nResultado para %s",palavra);
			exibir_dado(&aux);
		}
	}
}// okay
```

2.7.8 função pedida no exercício: Exibir um relatório consolidado sobre o dicionário com a contagem das palavras para cada letra do alfabeto a partir da AVL.

2.7.9 função pedida no exercício : Salvar o dicionário em um arquivo texto (percurso escolhido pelo usuário: em, pré ou pós-ordem) a partir da AVL

```
void func8 (const AVL_Tree *tree)
{
	int resposta = 0, temp = 0;
	printf("(1-pre| 2-in| 3-pos) entre com a forma que deseja salvar: ");
	scanf("%d",&temp);

	if(temp == 1)
	{
		resposta = write_dictionary_pre(tree);
	}
	else if(temp == 2)
	{
		resposta = write_dictionary_in(tree);
	}
	else if(temp == 3)
	{
		resposta = write_dictionary_pos(tree);
```

```
	}
	else
	{
		printf("Opção invalida!\n");
		return;
	}

	tratamento(resposta);
}
```

### 2.7.10 função main

```
int main (void)
{
	AVL_Tree *tree = constructor_AVL_Tree();
	TADList *list = constructor_TADList();

	if(tree != NULL && list != NULL)
	{
		int resposta = 0, opc = 0;
		do
		{
			opc = menu_principal();

			switch (opc)
			{
			case 1:
				resposta = read_dictionary(tree, list);
				tratamento(resposta);
				break;

			case 2:
				func2 (tree);
				break;

			case 3:
				func3(tree, list);
				break;

			case 4:
				func4(tree, list);
				break;

			case 5:
				func5 (tree, list);
				break;

			case 6:
				func6(tree);
```

```
                    break;

                case 7:
                    printf("ainda falta\n");
                    break;

                case 8:
                    func8 (tree);
                    break;
                case 9:
                    exibir_AVL_Tree(tree);
                    print_TADList(list);
                break;

                case 0:
                    continue;
                default:
                    printf("Entre com uma opção valida!\n");
                    break;
                }

                setbuf(stdin, NULL);
                printf("press to continue... ");
                getchar();

            }while (opc != 0);
            descructor_AVL_Tree(tree);
            desctructor_TADList(list);
        }
        return 0;
}
```

## Conclusão

É visível que para resolução de problemas cuja inserção e busca de elementos são muito usadas como neste trabalho envolvendo implementação de um programa para simular um dicionário de palavras o mais apropriado seria uma estrutura do tipo árvore AVL visto que tem todas as funções que precisamos para para solucionar nosso problema em um custo de tempo inferior a qualquer outra estrutura e é de fácil adaptação.

No desenvolvimento desse trabalho foi necessário um estudo aprofundado em estrutura de dados especialmente árvores avl nas funções de inserção e remoção pesquisando métodos de implementação mais simples como o recursivo para deixar o código mais legível.