

**Universidade Federal Rural de Pernambuco**

**Departamento de Estatística e Informática**

**Coordenação de Graduação em Ciência da Computação**

## **Algoritmo e Estrutura de Dados**

### **Lista de Exercícios para VA 2**

**Aluno**

**Giuseppe Fiorentino Neto**

**Professor**

**Rodrigo Nonamor Pereira Mariano de Souza**

**Recife**

**setembro–2017.1**

# EXERCÍCIOS

## QUESTAO 1

// Tira um elemento da fila e devolve seu o conteudo. Supondo que a fila não está vazia.

```
int tiradafila (void) {  
    int x = fila[p++];  
    if (p == N) p = 0;  
    return x; }
```

// Coloca um novo elemento na fila e devolve o endereço da cabeça da fila resultante.

```
void colocanafila (int y) {  
    fila[u++] = y;  
    if (u == N) u = 0;  
}
```

// Retorna um inteiro que indica se a fila esta ou não vazia. 1 se a fila estiver vazia e 0 caso  
//contrario.

```
int filavazia () {  
    return (u == p);  
}
```

// Retorna um inteiro que indica se a fila esta ou não cheia. 1 se a fila estiver cheia e 0 caso  
//contrario.

```
int filacheia () {  
    return ((u+1) % N == p);  
}
```

// Retorna o comprimento da fila.

```
int filacomprimento () {  
    int tamanho = p - u;  
    if(tamanho <0 ) tamanho * = -1;  
    return (tamanho);  
}
```

## QUESTAO 2

## QUESTAO 3

```
#define N 100
typedef struct {
    int individuo[N];
    int esq;
    int dir;
} tdeque;
//cria a fila
void criaFila(tdeque *fila){
    deque->esq=0;
    deque->di=0;
}
//testa se a fila esta vazia. Caso esteja retorna 1 caso
//contrario retorna 0;
int filaVazia(tdeque* fila){
    if(fila->esq==fila->dir) return 1;
    return 0;
}
//testa se a fila da direita esta cheia. Caso esteja retorna 1
```

```

//caso contrario retorna 0;
int filaCheiaD(tdeque *fila){
    int aux=fila->dir;
    if((++aux)%N)==fila->esq) return (1);
    return(0);
}

//testa se a fila da esquerda esta cheia. Caso esteja retorna
//1 caso contrario retorna 0;
int filaCheiaE(tdeque *fila){
    int aux=fila->dir;
    if((--aux)%N)==fila->esq) return (1);
    return(0);
}

//Remove os elementos do lado esquerdo. Recebe a estrutura que
//representa a fila e um ponteiro para o valor que sera
//removido, pois a funcao retorna 0 caso nao tenha removido na
// fila e 1 caso tenha removido na fila
int filaRemoverE(tdeque *fila, int *valor){//procurar oq eh tvalor
    if(filaVazia(&fila)==1) return (0);
    fila->esq=(++fila->esq)%N;
    *valor=fila->elemento[fila->esq];
    return 1;
}

//Remove os elementos do lado direito. Recebe a estrutura que
//representa a fila e um ponteiro para o valor que sera
//removido, pois a funcao retorna 0 caso nao tenha removido na
// fila e 1 caso tenha removido na fila
int filaRemoverD(tdeque *fila, int *valor){
    if(filaVazia(&fila)==1) return (0);
    *valor=fila->elemento[fila->dir];

```

```

        fila->dir=(--fila->dir)%N;
        return 1;
    }
//adiciona um valor a fila pelo lado esquerdo
int filaInserirD(int valor, tdeque *fila){
    if(filaCheiaD(&fila,)==1) return (0);
    fila->dir=(++fila->dir)%N;
    fila->indivduo[fila->dir]=valor;
    return (1);
}
//adiciona um valor a fila pelo lado direito
int filaInserirE(int valor, tdeque *fila){
    if(filaCheiaE(&fila,)==1) return (0);
    fila->indivduo[fila->esq]=valor;
    fila->dir=(--fila->esq)%N;
    return (1);
}

```

#### QUESTAO 4

```

//Define uma função que compara quem é o menor
#define less(A, B) (A < B)

//A função recebe duas listas ligadas crescentes a e b e devolve uma lista crescente que é o resultado
//da intercalação das duas listas dadas.
celula* intercala (celula* a,celula* b)
{
    struct reg head;

```

```

celula * c = &head;
c= malloc(sizeof(celula));
while (a != NULL && b != NULL){
    if (less(a->conteudo,b->conteudo)) {
        c->prox = a;
        c = a; a = a->prox;
    }
    else {
        c->prox = b;
        c = b; b = b->prox;
    }
}
c->prox = (a == NULL) ? b : a;
return c->prox;
}

```

## QUESTAO 5

//LISTA SIMPLES

struct Node { //estrutura que define o nó de uma lista

int conteudo;

struct Node\* prox;

};

typedef struct dll; // renomeia a estrutura Node para dll

//busca um valor a partir desse valor e da cabeça da lista em que esse valor esta inserido

dll \*busca (dll \* valor, dll \*cabeca)

{

```

dll *p;
p = cabeca;
while (p != NULL && p->elemento != valor)
    p = p->prox;
return p;
}
// A função recebe 2 listas ligadas e depois troca a posição de um com o outro. E como
//é uma lista ligada para haver uma modificação de quem é o valor antecedente e o consequente
//deve-se andar pela lista até achar o valor que se quer trocar a partir do seu valor antecedente.

```

```

void troca(dll *x , dll* y ){
    dll *anteX;
    dll *anteY;
    dll *proxX=x->prox;
    dll *proxY=y->prox;
    if(proxX==NULL || proxY==NULL) return;
    anteX=(busca(&x,&head))
    anteY=(busca(&y,&head))
    anteX->prox=y;
    y->prox=x->prox;
    anteY->prox=x;
    x->prox=y->prox;
}

```

//LISTA DUPLAMENTE LIGADA

```

struct Node { //estrutura que define o nó de uma lista duplamente ligada
    int elemento;
    struct Node* prox;
    struct Node* ant;
};

```

```

typedef struct dll; // renomeia a estrutura Node para dll

```

// A função recebe 2 listas duplamente ligadas e depois troca a posição de um com o outro. E como

//é uma lista duplamente ligada deve haver uma preocupação em modificar quem é o valor

//antecedente e o consequente

```
void troca(dll *x , dll* y ){
    dll *anteX=x->ant;
    dll *anteY=y->ant;
    dll *proxX=x->prox;
    dll *proxY=y->prox;
    if(proxX==NULL || proxY==NULL) return;

    anteX->prox=y;
    y->ant=anteX;
    y->prox=x->prox;

    anteY->prox=x;
    x->ant=anteY;
    x->prox=y->prox;
}
```

## QUESTAO 6

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define troca(A, B){int t = A; A = B; B = t;}

typedef struct{
    int k[10];
```



```
}vetor;
```

```
static void constroiHeap (int m, int v[])
```

```
{  
    int k;  
    for (k = 0; k < m; ++k){  
        int f = k + 1;  
        while (f > 0 && v[f/2] > v[f]){  
            troca (v[f/2],v[f] );  
            f /= 2;  
        }  
    }  
}
```

```
static void peneira (int m, int v[])
```

```
{  
    int p = 0, f = 1, t = v[0];  
    while (f <= m) {  
        if (f < m && v[f] > v[f+1]) ++f;  
        if (t <= v[f]) break;  
        v[p] = v[f];  
        p = f, f = 2*p;  
    }  
    v[p] = t;  
}
```

```
void heapsort(int n, int v[])
```

```
{  
    int m;  
    constroiHeap (n, v);
```

```

        for (m = n; m >= 1; --m) {
            troca (v[0], v[m]);
            peneira (m-1, v);
        }
    }

int main()
{
    srand(time(NULL));
    vetor A[10];
    int t[10];
    int i, j;
    for(i=0;i<10;i++){
        t[i] = 0;
    }
    for(i=0;i<10;i++){
        for(j=0;j<10;j++){
            A[i].k[j] = rand()%10;
            t[i] += A[i].k[j];
            printf("%i ", A[i].k[j]);
        }
        printf("\n");
    }
    for(i=0;i<10; i++){
        printf("\n%i", t[i]);
    }

    printf("\n");
    heapsort(10, t);
    for(i=0;i<10; i++){

```

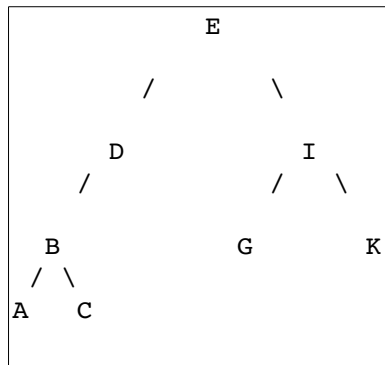
```

        printf("\n%i", t[i]);
    }
    return 0;
}

```

### QUESTAO 7

Não, pois se tomarmos a seguinte árvore binária como exemplo de altura 3, e somarmos a profundidade com a altura, por exemplo, do nó I, logo teremos 2 como valor da soma e isso faz com que a afirmação não seja verdade.



### QUESTAO 8

```

// Recebe uma árvore binária não vazia r
// e devolve o primeiro nó da árvore
// na ordem e-r-d.
noh *primeiro(arvore r){
    if(r->esq == NULL) return r;
    return primeiro(r->esq);
}

```

### QUESTAO 9

```
// Recebe uma árvore binária não vazia r
// e devolve o ultimo nó da árvore
// na ordem e-r-d.
noh *ultimo (arvore r) {
    while (r->dir != NULL) r = r->dir;
    return r;
}
```

### QUESTAO 10

```
// Recebe um nó x, este que é diferente de NULL, e devolve o nó anterior na ordem e-r-d.
noh *anterior (noh *x) {
    if (x->esq != NULL) {
        noh *y = x->esq;
        while (y->dir != NULL) y = y->dir;
        return y;
    }
    while (x->pai != NULL && x->pai->esq == x)
        x = x->pai;
    return x->pai;
}
```

### QUESTAO 11

//A função recebe um vetor que representa a expressão pos-fixa cria-se uma pilha que empilhara os  
//elementos dessa expressao. Desloca-se pelo vetor e toda vez que encontrar um numero este é  
//transformado em uma arvore que não tem filhos e que o numero é a raiz dessa arvore.  
//Esta arvore é empilhada na pilha de arvores e ao achar no vetor de expressao um sinal operatorio  
//desempilha duas arvores da pilha e as insere como subarvore na arvore em que tem como raiz um  
//sinal operatorio. E assim é feito até o fim do vetor criando-se assim uma arvore.

```
void subArvores(char v[]){  
    int i;  
    noh *a;  
    noh *e,*d;  
    criapilha();  
    for(i=0;i<strlen(v);i++){  
        if(v[i]=='+'){  
            d=desempilha();  
            e=desempilha();  
            a=arv_cria(v[i],e,d);  
            empilha(a);  
        }  
        if(v[i]=='*'){  
            d=desempilha();  
            e=desempilha();  
            a=arv_cria(v[i],e,d);  
            empilha(a);  
        }  
    }  
}
```

```

    }
    if(v[i]=='-'){
        d=desempilha();
        e=desempilha();
        a=arv_cria(v[i],e,d);
        empilha(a);

    }
    if(v[i]=='/'){
        d=desempilha();
        e=desempilha();
        a=arv_cria(v[i],e,d);
        empilha(a);

    }
    else if(v[i]>='0' && v[i]<='9'){
        empilha(arv_cria(0,inicializarArvore(),inicializarArvore()));
    }
    while (v[i] >= '0' && (v[i] <= '9')){

        valor=10 * desempilha()->conteudo + (v[i++] - '0')+'0';
        empilha(arv_cria(valor,inicializarArvore(),inicializarArvore()));
    }
}
}

```

### QUESTAO 12

// Recebe a raiz de uma árvore binária e retorna true caso a árvore seja de busca.

//Considerando que foi definido o tipo bool.

```
bool Busca (arvore r) {  
    if (r != NULL) {  
        bool e = Busca (r->esq);  
        bool d = Busca (r->dir);  
        if(r->dir->chave >= r->chave && r->esq->chave <= r->chave && e && d)  
            return true;  
        else return false;  
    }  
    return true;  
}
```

### QUESTAO 13

//Transforma um vetor que é crescente em uma árvore binária de busca balanceada.

void balancearArvoreBinaria (arvore r, int vetor[], int inicio, int fim)

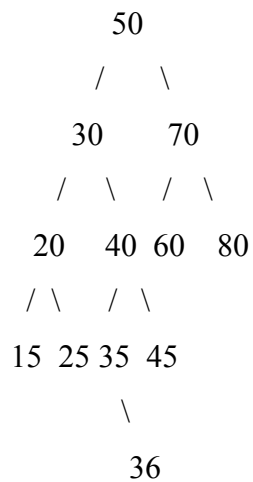
```
{  
    if (inicio <= fim)  
    {  
        int meio = (inicio+fim)/2;  
        noh *novo;  
        novo = malloc (sizeof (noh));  
        novo->chave = vetor[meio];  
        novo->esq = novo->dir = NULL;  
        inserir( r, novo);  
    }  
}
```

```

        balancearArvoreBinaria ( r, vetor, inicio, meio-1);
        balancearArvoreBinaria ( r, vetor, meio+1, fim);
    }
}

```

#### QUESTAO 14



#### QUESTAO 15

```

#include<stdio.h>

#include<stdlib.h>

#define troca (A, B) { int t = A; A = B; B = t; }

#define N 100

//ARVORE

```



```

//define a estrutura que representa o no da arvore
typedef struct reg {
    int conteudo;
    struct reg *pai;
    struct reg *esq, *dir;
} noh;

noh fila[N]; //cria uma fila de prioridades em um vetor

noh* inicializarArvore() //inicializa a arvore
{
    return NULL;
}

int arvoreVazia(noh *raiz) { //testa se a arvore ta vazia
    if(!raiz) return (1);
    else return (0);
}

noh* arv_cria(int c, noh* sae, noh* sad) //cria a subarvore
{
    noh* p=(noh*)malloc(sizeof(noh));
    if(p==NULL) exit(1);
    p->conteudo = c;
    p->esq = sae;
    p->dir = sad;
    sae->pai=p;
    sad->pai=p;
    return p;
}

noh* arv_libera (noh* a) //liberar a subarvore de acordo com o nó
if (!arv_vazia(a)) {
    arv_libera(a->esq); /* libera sae */
    arv_libera(a->dir); /* libera sad */
}

```

```

        free(a); /* libera raiz */
    }
    return NULL;
}

int arv_vazia (noh* a)
{
    return a==NULL;
}

//FILA DE PRIORIDADES

// A função recebe um vetor A[1..m-1] e rearranja este de modo a transformar o vetor em um max-heap.

void corrige-subindo(int A[], int m){//
    int i=m;
    if (r != NULL) {
        erd (r->esq);
        erd (r->dir);
        if(r->dir->conteudo < r->esq->conteudo){
            if(r->conteudo<r->esq->conteudo){
                int tmp=r->conteudo;
                r->conteduo=r->esq->conteduo;
                r->esq->conteduo=tmp;
            }
        }
        else if(r->dir->conteudo > r->esq->conteudo){
            if(r->conteudo < r->dir->conteudo){
                int tmp=r->conteudo;
                r->conteduo=r->dir->conteduo;
                r->dir->conteduo=tmp;
            }
        }
    }
}

```

```
}
```

```
}
```

//A ideia da função é: se A[i] é maior ou igual que seus filhos então não é preciso fazer nada; senão, troque A[i] com o maior dos filhos e repita o

//processo para o filho envolvido na troca.

```
void corrige-descendo(int A[], int n, int i){
```

```
    int j=i;
```

```
    if (r != NULL) {
```

```
        if(r->dir->conteudo < r->esq->conteudo){
```

```
            if(r->conteudo < r->esq->conteudo){
```

```
                int tmp=r->conteudo;
```

```
                r->conteudo=r->esq->conteudo;
```

```
                r->esq->conteudo=tmp;
```

```
            }
```

```
        }
```

```
    else if(r->dir->conteudo > r->esq->conteudo){
```

```
        if(r->conteudo < r->dir->conteudo){
```

```
            int tmp=r->conteudo;
```

```
            r->conteudo=r->dir->conteudo;
```

```
            r->dir->conteudo=tmp;
```

```
        }
```

```
    }
```

```
    erd (r->esq);
```

```
    erd (r->dir);
```

```
}
```

```
}
```

//cria a fila de prioridade

```
void criafila (void) {
```

```

    p = 0; u = 0;
}

//testa pra saber se a fila esta vazia
int filavazia (void) {
    return p >= u;
}

//remove da fila e corrige a fila de prioridade
noh* tiradafila (void) {
    noh i= fila[p++];
    corrige-descendo(A,n,i);
    return i;
}

//coloca na fial e corrige a fila de prioridade
void colocanafila (noh y) {
    fila[u++] = y;
    corrige-subindo(A,i);
}

void movimentarNaFila (noh r) {
    criafila (); // cria a fila
    colocanafila (r); //coloca na fila
    while (1) {
        x = tiradafila (); //remove da fila
        if (x != NULL) {
            colocanafila(x);
            colocanafila (x->esq);
        }
        else {
            if (filavazia ()) break;
            x = tiradafila (); //remover da fila

```

```

        colocanafila (x->dir);
    }
}

}

```

### QUESTAO 16

```

// Devolve a altura da árvore binária
// cuja raiz é r.
int altura (arvore r) {
    if (r == NULL)
        return -1; // altura da árvore vazia
    else {
        int he = altura (r->esq);
        int hd = altura (r->dir);
        if (he < hd) return hd + 1;
        else return he + 1;
    }
}

//Preenche o campo fb de cada nó da arvore
void preencherFB(arvore r){
    if (r != NULL) {
        preencherFB (r->esq);
        r->fb = altura(r->esq) - altura(r->dir);
        preencherFB (r->dir);
    }
}

```

```
}
```

### QUESTAO 17

```
//Transforma um vetor crescente em uma árvore
```

```
//binária de busca balanceada.
```

```
void balancearArvoreBinaria (arvore r, int vetor[], int inicio, int fim)
```

```
{
```

```
    if (inicio <= fim)
```

```
    {
```

```
        int meio = (inicio+fim)/2;
```

```
        noh *novo;
```

```
        novo = malloc (sizeof (noh));
```

```
        novo->chave = vetor[meio];
```

```
        novo->esq = novo->dir = NULL;
```

```
        inserir( r, novo);
```

```
        balancearArvoreBinaria ( r, vetor, inicio, meio-1);
```

```
        balancearArvoreBinaria ( r, vetor, meio+1, fim);
```

```
    }
```

```
}
```

```
// Recebe a raiz r de uma árvore binária de busca
```

```
// e transforma em uma arvore de busca balanceada
```

```
arvore transformarArvore(arvore r){
```

```
    int *vetor = malloc(sizeof(int));
```

```
    arvoreVetor( r, vetor);
```

```
    arvore nova;
```

```
    balancearArvoreBinaria ( nova, vetor, 0, sizeof(vetor)/sizeof(int));
```

```

}

//Recebe a raiz de uma arvore binaria

//e a transforma em um vetor de ordem crescente

void arvoreVetor (arvore r, int* vetor) {
    if (r != NULL) {
        arvoreVetor (r->esq);
        vetor = realloc(vetor,((sizeof(vetor)/sizeof(int))+ 1) *sizeof(int));
        vetor[(sizeof(vetor)/sizeof(int))-1] = r->chave;
        arvoreVetor (r->dir);
    }
}

```

## QUESTAO 18

```

#include<stdio.h>

#include<stdlib.h>

#define M 13

// Tamanho da tabela.

#define hash(v, M) (v % M)

// Transforma uma chave v em um índice no intervalo 0..M-1.

//struct que define o objeto

typedef struct {
    int chave;
    int valor;
} tipoObjeto;

```

```
tipoObjeto objetonulo;
```

```
//contagem
```

```
struct reg {
```

```
    int    chave, ocorr;
```

```
    struct reg *prox;
```

```
};
```

```
typedef struct reg celula;
```

```
celula **tb;
```

```
//contagem por sondagem linear
```

```
struct reg {
```

```
    int    chave, ocorr;
```

```
    struct reg *prox;
```

```
};
```

```
typedef struct reg celula;
```

```
//struct que define o nó
```

```
struct STnode {
```

```
    tipoObjeto obj;
```

```
    struct STnode * next;
```

```
} ;
```

```
typedef struct STnode *link;
```

```
//criação da tabela
```

```
link *tab;
```

```
//iniciar tabela
```

```
void STinit()
```

```
{
```

```
    int h;
```

```
    tab = malloc(M * sizeof (link));
```



```

    for (h = 0; h < M; h++)
        tab[h] = NULL;
}

//inserir novo obj na tabela
void STinsert(tipoObjeto obj)
{
    int h, v;
    v = obj.chave;
    h = hash(v, M);
    link novo = malloc(sizeof (link));
    novo->obj = obj;
    novo->next = tab[h];
    tab[h] = novo;
}

//pesquisar pelo valor que foi enviado para a função. Caso exista retorna o objeto desse valor
//caso não exista retornar o objetonulo
tipoObjeto STsearch(int v)
{
    link t;
    int h;
    h = hash(v, M);
    for (t = tab[h]; t != NULL; t = t->next)
        if (t->obj.chave == v) break;
    if (t != NULL) return t->obj;
    return objetonulo;
}

//contabilizador
void contabiliza (int ch) {
    int h = hash (ch, M);
    celula *p = tb[h];

```

```

while (p != NULL && p->chave != ch)
    p = p->prox;
if (p != NULL)
    p->ocorr += 1;
else {
    p = malloc (sizeof (celula));
    p->chave = ch;
    p->ocorr = 1;
    p->prox = tb[h];
    tb[h] = p;
}
}

```

//contabilizador para sondagem linear

```

void contabilizaSond (int ch) {
    int h = hash (ch, M);
    cell *p = tb[h];
    while (p != NULL && p->chave != ch)
        p = p->prox;
    if (p != NULL)
        p->ocorr += 1;
    else {
        p = malloc (sizeof (cell));
        p->chave = ch;
        p->ocorr = 1;
        p->prox = tb[h];
        tb[h] = p;
    }
}

```

//printar na tela os valores

```

void display() {

```

```
int i = 0;
```

```
for(i = 0; i<M; i++) {
```

```
    if(tab[i] != NULL)
```

```
        printf(" (%d,%d)",tab[i]->obj.chave,tab[i]->obj.valor);
```

```
    else
```

```
        printf(" ~~ ");
```

```
}
```

```
printf("\n");
```

```
}
```

```
void main(void){
```

```
    objetonulo.chave=0;
```

```
    tb = (celula *)malloc (M * sizeof (celula *));
```

```
    int v[]={17,21,19,4,26,30,37};
```

```
    tipoObjeto novo;
```

```
    int i;
```

```
    STinit();
```

```
    for(i=0;i<7;i++) {
```

```
        novo.chave=10*i;
```

```
        novo.valor=v[i];
```

```
        STinsert(novo);
```

```
    }
```

```
    display();
```

```
}
```

## QUESTAO 19

```
#include<stdio.h>

#include<stdlib.h>

#define M 8191

// Tamanho da tabela.

#define hash(v, M) (v % M)

// Transforma uma chave v em um índice no intervalo 0..M-1.

typedef struct {
    int chave;
    int ocorr;
} tipoObjeto;

tipoObjeto objetonulo;

// Todas as chaves "válidas" são estritamente positivas.

// Definição de um nó das listas de colisões.
typedef struct STnode *link;
struct STnode {
    tipoObjeto obj;
    link    next;
} ;

// Tabela que aponta para as M listas de colisões.
link *tab;

// Inicializa uma tabela de simbolos que armazena
// objetos. A espinha dorsal da tabela será um
```

```

// vetor tab[0..M-1].
void salvar(int valor){
    FILE*arq;
    arq=fopen("numeros1.txt","r");
    if(arq==NULL) arq=fopen("numeros1.txt","w");
    else arq=fopen("numeros1.txt","a");
    printf("%i",valor);
    fprintf(arq,"%i\n",valor);

}

void STinit()
{
    int h;
    tab = malloc(M * sizeof (link));
    for (h = 0; h < M; h++)
        tab[h] = NULL;
}

// Insere obj na tabela de símbolos.
//
void STinsert(tipoObjeto obj)
{
    int h, v;
    v = obj.chave;
    h = hash(v, M);
    link t=tab[h];
    for (t = tab[h]; t != NULL; t = t->next) {
        if (t->obj.chave == v) break;
    }
    if (t != NULL)

```

```

        t->obj.ocorr++;
    if(t==NULL){
        obj.ocorr=1;
        link novo = malloc(sizeof (link));
        novo->obj = obj;
        novo->next = tab[h];
        tab[h] = novo;
        salvar(obj.chave);
    }
}

```

// Devolve um objeto cuja chave é v. Se tal objeto não existe,  
// a função devolve um objeto fictício com chave nula.

//

```

tipoObjeto STsearch(int v)
{
    link t;
    int h;
    h = hash(v, M);
    for (t = tab[h]; t != NULL; t = t->next) {
        if (t->obj.chave == v) break;
    }
}

```

```

    if (t != NULL) return t->obj;

```

```

    return objetonulo;
}

```

```

static void imprime ()
{

```

```

link p;
int i=0;
for (i=0; i<M; i++)
    for (p=tab[i]; p!=NULL; p=p->next)
        printf("key%i-ocorr%i\n",p->obj.chave,p->obj.ocorr);
}

```

```

int main (void)
{
    objetonulo.chave = 0;
    STinit();
    FILE* fp;
    fp=fopen("numeros.txt","rt");
    if(fp==NULL) {
        printf("ERRO na abertura do arquivo.\n");
        return 0;
    }
    int c;
    fscanf(fp,"%d",&c);
    tipoObjeto novo;
    while(!feof(fp)){
        novo.chave=c;
        STinsert(novo);
        fscanf(fp,"%d",&c);
    }
    imprime();
    return 0;
}

```

**QUESTAO 20**

A questão pede a implementação de um algoritmo que exiba o número de vezes cada palavra ocorre em um dado texto. A forma implementada aqui usa a estrutura de dados Hash para deixar o algoritmo mais eficiente e mais ágil.

Deve-se entender a priori que uma palavra é uma sequência de uma ou mais letras (maiúsculas ou minúsculas) e dessa forma foi criado um tipo que representa cada palavra. O tipo chama-se “Palavra” e possui um contador de ocorrência, a colisão com listas e um vetor que representa o tamanho da palavra. Como na língua portuguesa a maior palavra possui 46 letras (segundo o site <https://www.normaculta.com.br/maior-palavra-da-lingua-portuguesa/>) definiu-se como padrão de tamanho de uma palavra 67 letras dando uma margem de erro segura.

Para usar a tabela hash criou-se um vetor de palavras, que possui o tamanho máximo de 127 palavras, para armazenar cada palavra que foi encontrada no texto.

O pré-processamento é feito pela leitura de palavras. Captura-se uma sequência de letras do arquivo texto, ou seja uma palavra, pulando os caracteres que não são letras e armazenando a sequência de letras a partir da posição do cursor do arquivo.

O processo de armazenamento das palavras lidas e da sua frequência na tabela hash é por meio de uma tabela de dispersão (que mapeia a chave de busca em um índice da tabela, que é soma dos códigos dos caracteres que compõem a cadeia e tira o módulo dessa soma para se obter o índice da tabela) e a própria palavra como chave de busca. Assim, a partir de uma função para obter a frequência das palavras, dada uma palavra tenta-se encontrá-la na tabela e se não existir é armazenada a palavra na tabela e se existir, incrementa o número de ocorrências da palavra. Isso é feito a partir de uma função para acessar os elementos na tabela. Dada uma palavra (chave de busca) é retornado o ponteiro da estrutura Palavra associada. Se a palavra ainda não existir na tabela, cria-se uma nova palavra e é fornecido como retorno essa nova palavra criada.

Para finalizar o que foi pedido na questão foi criada uma função para exibir as ocorrências em ordem decrescente. Foi criado um vetor armazenando as palavras da tabela de dispersão e ordenando o vetor para facilitar a visualização quando exibir o conteúdo do vetor. Além disso também foi criada uma função, baseada na função anterior, que mostra apenas as palavras para um dado número de ocorrências que o usuário define.