Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Algoritmos e Estruturas de Dados II (Turmas M, N, W, F)
1º Semestre de 2012
Profs. Camilo Oliveira, Gisele Pappa, Ítalo Cunha, Loïc Cerf, William Schwartz

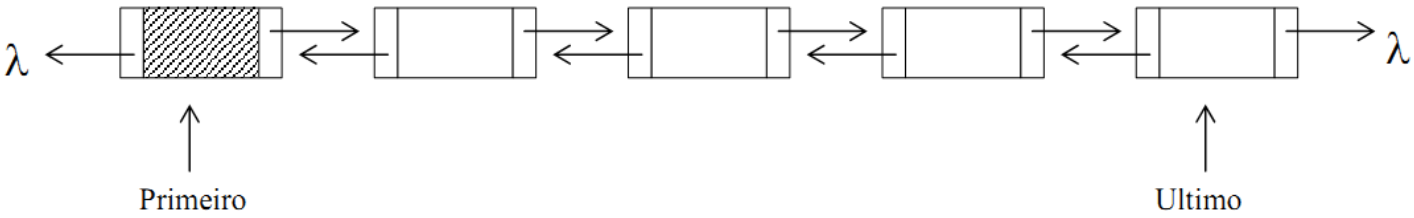
2ª Lista de Exercícios

1. Utilizando a implementação de listas com alocação sequencial (arranjos), implemente um procedimento para inserir um item em uma determinada posição da lista. Qual é a ordem de complexidade do seu procedimento?

```
#define MaxTam 1000
typedef int Apontador;

typedef struct {
   /* dados */
} TipoItem;

typedef struct {
   TipoItem Item[MaxTam];
   Apontador Primeiro, Ultimo;
} TipoLista;

void Insere(TipoLista *lista, TipoItem dado, int posicao) {
int i;

    if (lista->Ultimo >= MaxTam) {
       printf("Lista está cheia\n");
   }
   else {
       for (i = lista->Ultimo; i >= posicao; i--)
           lista->Item[i] = lista->Item[i-1];
       lista->Item[posição] = dado;
   }
}
```

A complexidade é O(MaxTam).

2. Um problema que surge com frequência na manipulação de listas lineares implementadas através de apontadores é "andar para trás" na lista, ou seja, percorrê-la no sentido inverso dos apontadores. Uma solução para isso é implementar uma lista duplamente encadeada, em que cada célula também possui um apontador para a sua antecessora, como mostrado na figura abaixo:



a) Declare os tipos necessários para a implementação dessa lista

b) Escreva um procedimento para retirar uma célula apontada por um apontador p

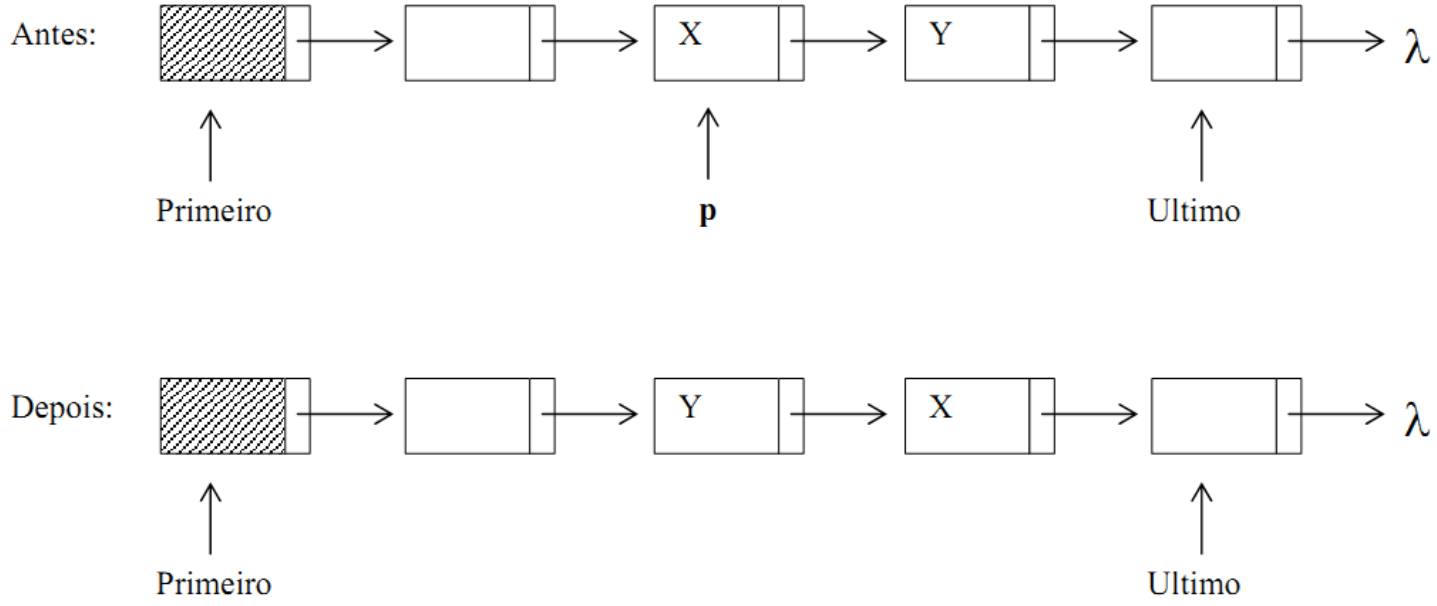
c) Escreva um procedimento para inserir um elemento na primeira posição da lista (logo após a célula cabeça).

```
struct lista {
    struct no *cabeca;
    struct no *ultimo;
};

struct no {
    struct no *ante;
    struct no *prox;
    int dado;
};

int remove_elemento(struct lista *L, struct no *P)
{
    struct no *ante = P->ante;
    struct no *prox = P->prox;
    ante->prox = prox; /* ante != NULL por causa do cabeca */
    if(prox != NULL) { prox->ante = ante; }
    if(L->ultimo == P) { L->ultimo = ante; }
    int dado = P->dado;
    free(P);
    return dado;
}

void insere_inicio(struct lista *L, int i)
{
    struct no *primeiro = L->cabeca->prox;
    struct no *novo = malloc(sizeof(struct no));
    if(novo == NULL) { perror(NULL); exit(EXIT_FAILURE); }
    L->cabeca->prox = novo;
    if(primeiro != NULL) { primeiro->ante = novo; }
    novo->ante = L->cabeca;
    novo->prox = primeiro;
    novo->dado = i;
    if(L->ultimo == L->cabeca) { L->ultimo = novo; }
}
```

3. Considere uma pilha P vazia e uma fila F não vazia. Utilizando apenas os testes de fila e pilha vazias, as operações *Enfileira*, *Desenfileira*, *Empilha*, *Desempilha*, e uma variável *aux* do *TipoItem*, escreva um programa que inverte a ordem dos elementos da fila.

```
void InverteFila(TipoFila *fila) {
TipoPilha *pilha;

   pilha = (TipoPilha *) malloc(sizeof(TipoPilha));
   FPVazia(pilha);

   while (Vazia(fila) == 0)
      Empilha(Desenfileira(fila), pilha);
   while (Vazia(pilha) == 0)
      Enfileira(Desempilha(pilha), fila);
}
```

4. Considere a implementação de listas encadeadas utilizando apontadores vista em sala. Escreva um procedimento *Troca(TipoLista *Lista, Apontador *p)* que, dado um apontador para uma célula qualquer (p), troca de posição essa célula com a sua célula seguinte da lista, como mostrado na figura abaixo. (Obs. Não vale trocar apenas o campo item! Você deverá fazer a manipulação dos apontadores para trocar as duas células de posição). Não esqueça de tratar os casos especiais.




```
struct lista {
    struct no *cabeca;
    struct no *ultimo;
};
struct no {
    struct no *ante;
    struct no *prox;
    int dado;
};
typedef struct lista TipoLista;
typedef struct no Apontador;
```

```
/* void Troca(struct lista *L, struct no *P) ou */
void Troca(TipoLista *L, Apondador *P)
{
    struct no *ante;
    struct no *prox = P->prox;
    if(prox == NULL) { return; } /* impossível trocar posições. */
    ante = L->cabeca;
    while(ante->prox != P) {
        ante = ante->prox;
    }
    /* ante é o nó anterior ao nó P */
    ante->prox = prox;
    P->prox = prox->prox;
    prox->prox = P;
    if(L->ultimo == prox) { L->ultimo = ante; }
}
```

5. Utilizando as operações de manipulação de pilhas vistas em sala, uma pilha auxiliar e uma variável do tipo TipoItem, escreva um procedimento que remove um item com chave **c** de uma posição qualquer de uma pilha. Note que você não tem acesso à estrutura interna da pilha (topo, item, etc), apenas às operações de manipulação.

```
typedef struct {
   TipoChave chave;
   /* dados */
} TipoItem;

void RemoveChaveC(TipoPilha *pilha, TipoChave c) {
TipoPilha *pilha2;
TipoItem aux;

   pilha2 = (TipoPilha *) malloc(sizeof(TipoPilha));
   FPVazia(pilha2);

   while (Vazia(pilha) == 0) {
      aux = Desempilha(pilha);
      if (aux.chave != c)
         Empilha(aux, pilha2);
      else
         break;
   }
   while (Vazia(pilha2) == 0)
      Empilha(Desempilha(pilha2), pilha);
}
```

6. Escreva dois procedimentos (um usando a implementação por arranjos (alocação sequencial) e outro usando a implementação por apontadores) para remover de uma lista encadeada um elemento com uma chave específica (passada como parâmetro): *Remove(TipoLista *Lista, TipoChave c)*. Qual é a ordem de complexidade dos seus procedimentos?

Implementação para lista de inteiros:

```
#define MAX_ELEMENTOS 65535

struct lista_arranjo {
    int dados[MAX_ELEMENTOS];
    int ultimo;
};

int remove_elemento_arranjo(struct lista_arranjo *L, int dado)
{
    int i;
    int j;
    for(i = 0; i < L->ultimo && L->dados[i] != chave; i++);
    /* se o dado está na lista, i contém o índice do elemento a
     * remover. se o dado não está na lista, i == L->ultimo. */
    if(i == L->ultimo) { return 0; }
    else {
        /* o dado está no elemento no índice i, que vamos remover: */
        while(i < L->ultimo - 1) {
            L->dados[i] = L->dados[i+1];
            i++;
        }
        return 1;
    }
}
```

Esta função possui complexidade O(N) no melhor e no pior caso, onde N é o número de elementos na lista.

```
struct lista_encadeada {
    struct no *cabeca;
    struct no *ultimo;
};

struct no {
    struct no *ante;
    struct no *prox;
    int dado;
}
```

```
int remove_elemento_encadeado(struct lista_encadeada *L, int dado)
{
    struct no *ante;
    ante = L->cabeca;
    while(ante->prox && ante->prox->dado != dado) {
        ante = ante->prox;
    }
    /* se a chave está na lista, então ante->prox->dado == chave.
     * se a chave não está na lista, então ante->prox == NULL. */
    if(ante->prox == NULL) { return 0; }
    else {
        /* chave está no elemento ante->prox, que vamos remover.
         * guardamos um ponteiro pro nó ancessor ao que queremos remover
         * para facilitar a implementação da remoção. */
        struct no *prox = ante->prox;
        if(L->ultimo == prox) { L->ultimo = ante; }
        ante->prox = prox->prox;
        free(prox);
        return 1;
    }
}
```

Esta função possui complexidade O(1) no melhor caso (dado está no primeiro elemento da lista) e O(N) no pior caso (dado está no último elemento da lista), onde N é o número de elementos na lista.

Implementação para listas de itens que contém estruturas que armazenam chaves e dados distintos:

Na implementação a seguir, os elementos na lista são do tipo `struct item`*, que possui uma chave do tipo inteiro mais um número arbitrário de campos para armazdenar dados (nome, idade, endereço, telefone, etc). As funções de remover pela chave inspecionam a chave de cada elemento.*

Comparada à implementação anterior, esta implementação permite o armazenamento de uma quantidade de dados arbitrária em cada elemento. Ela também permite identificar os dados por uma chave.

Uma limitação desta implementação é que as chaves são comparadas com o operador de igualdade (==). Logo, só podemos utilizar chaves de tipos básicos (short, int, long, float, double); pois tipos complexos (arranjos, strings, structs) não podem ser comparados diretamente usando o operador de igualdade.

```
#define MAX_ELEMENTOS 65535

struct item {
    int chave;
    /* campos de dados. por exemplo: char nome[80]; int telefone; etc */
};

struct lista_arranjo {
    struct item dados[MAX_ELEMENTOS];
    int ultimo;
};
```

```
int remove_elemento_arranjo(struct lista_arranjo *L, int chave)
{
    int i;
    int j;
    for(i = 0; i < L->ultimo && L->dados[i].chave != chave; i++);
    if(i == L->ultimo) { return 0; }
    else {
        while(i < L->ultimo - 1) {
            L->dados[i] = L->dados[i+1];
            i++;
        }
        return 1;
    }
}
```

Esta função possui complexidade O(N) no melhor e no pior caso, onde N é o número de elementos na lista.

```
struct lista_encadeada {
    struct no *cabeca;
    struct no *ultimo;
};

struct no {
    struct no *ante;
    struct no *prox; /* prox e ante são ponteiros, acessamos com seta (->) */
    struct item dado; /* note que dado não é ponteiro, então acessamos com ponto (.) */
}

int remove_elemento_encadeado(struct lista_encadeada *L, int chave)
{
    struct no *ante;
    ante = L->cabeca;
    while(ante->prox && ante->prox->dado.chave != chave) {
        ante = ante->prox;
    }
    if(ante->prox == NULL) { return 0; }
    else {
        struct no *prox = ante->prox;
        if(L->ultimo == prox) { L->ultimo = ante; }
        ante->prox = prox->prox;
        free(prox);
        return 1;
    }
}
```

Esta função possui complexidade O(1) no melhor caso (dado está no primeiro elemento da lista) e O(N) no pior caso (dado está no último elemento da lista), onde N é o número de elementos na lista.

Implementação para listas de itens que contém estruturas que armazenam chaves e dados usando `typedef`:

A implementação a seguir é similar à anterior, porém definimos um tipo para a chave e outro tipo para os items. Isto torna mais fácil mudar o tipo da chave que identifica os itens na lista.

```
#define MAX_ELEMENTOS 65535

typedef int TipoChave;

struct item {
    TipoChave chave;
    /* campos de dados. por exemplo: char nome[80]; int telefone; etc */
};
```

```
typedef struct item TipoItem;

struct lista_arranjo {
    TipoItem dados[MAX_ELEMENTOS];
    int ultimo;
};

int remove_elemento_arranjo(struct lista_arranjo *L, TipoChave chave)
{
    int i;
    int j;
    for(i = 0; i < L->ultimo && L->dados[i].chave != chave; i++);
    if(i == L->ultimo) { return 0; }
    else {
        while(i < L->ultimo - 1) {
            L->dados[i] = L->dados[i+1];
            i++;
        }
        return 1;
    }
}
```

Esta função possui complexidade O(N) no melhor e no pior caso, onde N é o número de elementos na lista.

```
struct lista_encadeada {
    struct no *cabeca;
    struct no *ultimo;
};

struct no {
    struct no *ante;
    struct no *prox;
    TipoItem dado;
}

int remove_elemento_encadeado(struct lista_encadeada *L, TipoChave chave)
{
    struct no *ante;
    ante = L->cabeca;
    while(ante->prox && ante->prox->dado.chave != chave) {
        ante = ante->prox;
    }
    if(ante->prox == NULL) { return 0; }
    else {
        struct no *prox = ante->prox;
        if(L->ultimo == prox) { L->ultimo = ante; }
        ante->prox = prox->prox;
        free(prox);
        return 1;
    }
}
```

Esta função possui complexidade O(1) no melhor caso (dado está no primeiro elemento da lista) e O(N) no pior caso (dado está no último elemento da lista), onde N é o número de elementos na lista.

Implementação genérica usando ponteiro para `void`:

A implementação genérica usando ponteiro para void é similar às anteriores.

A diferença é que o compilador não sabe comparar elementos do tipo void, então precisamos informar ao compilador como comprar nossos elementos. Fazemos isso passando para a função de remover elementos uma outra função para

comparar items. Esta função de comparação retorna zero se os elementos forem iguais e um se os elementos forem diferentes.

Comparada às implementações anteriores, esta implementação é mais genérica em dois sentidos. Primeiro, ela permite o armazenamento de qualquer tipo de dado. Segundo, ela permite qualquer tipo de chave (strings, structs,etc).

```
struct item {
    int chave;
    /* dados. */
};

int compara_items(void *i1, void *i2)
{
    /* só funciona se a lista conter elementos do tipo struct item: */
    struct item *item1 = i1;
    struct item *item2 = i2;
    /* o resto da função pode ser adaptado para qualquer tipo de chave,
     * por exemplo chaves complexas como structs ou strings. neste exercício a
     * comparação é simples por que a chave é um inteiro: */
    if(item1->chave != item2->chave) { return 1; }
    else { return 0; }
}

#define MAX_ELEMENTOS 65535

struct lista_arranjo {
    void * dados[MAX_ELEMENTOS];
    int ultimo;
};
```

O último parâmetro da função abaixo é um ponteiro para a função para comparar os elementos.

```
int remove_elemento_arranjo(struct lista_arranjo *L, void *chave,
        int (*compara_items)(void *i1, void *i2))
{
    int i;
    int j;
    for(i = 0; i < L->ultimo && compara_items(L->dados[i], chave); i++);
    if(i == L->ultimo) { return 0; }
    else {
        while(i < L->ultimo - 1) {
            L->dados[i] = L->dados[i+1];
            i++;
        }
        return 1;
    }
}
```

Esta função possui complexidade O(N) no melhor e no pior caso, onde N é o número de elementos na lista.

```
struct lista_encadeada {
    struct no *cabeca;
    struct no *ultimo;
};

struct no {
    struct no *ante;
    struct no *prox;
    void * dado;
}
```

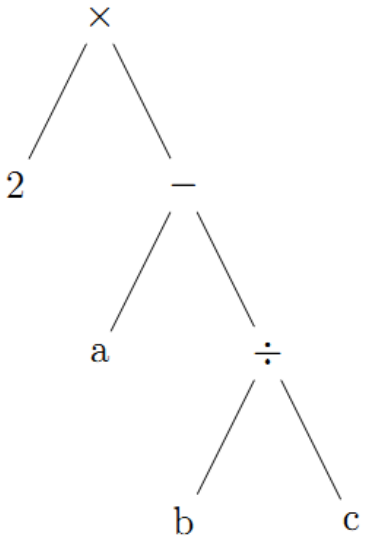
```
int remove_elemento_encadeado(struct lista_encadeada *L, void *chave,
        int (*compara_items)(void *i1, void *i2))
{
    struct no *ante;
    ante = L->cabeca;
    while(ante->prox && compara_items(ante->prox->chave, chave)) {
        ante = ante->prox;
    }
    if(ante->prox == NULL) { return 0; }
    else {
        struct no *prox = ante->prox;
        if(L->ultimo == prox) { L->ultimo = ante; }
        ante->prox = prox->prox;
        free(prox);
        return 1;
    }
}
```

Esta função possui complexidade O(1) no melhor caso (dado está no primeiro elemento da lista) e O(N) no pior caso (dado está no último elemento da lista), onde N é o número de elementos na lista.

7. Considere a implementação de filas usando arranjos "circulares" (alocação sequencial). Escreva um procedimento FuraFila(TipoFila *fila, TipoItem x) que insere um item na primeira posição da fila. O detalhe é que seu procedimento deve ser **O(1)**, ou seja, não pode movimentar os outros itens da fila.

```
#define MaxTam  1000
typedef int Apontador;

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Frente, Tras;
} TipoFila;


void FuraFila(TipoFila *fila, TipoItem x) {

  fila->Frente--;
  if (fila->Frente < 0)
     fila->Frente = MaxTam-1;
  fila->Item[fila->Frente] = x;
}
```

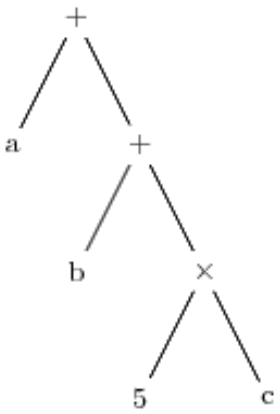
8. Expresse as seguintes equações usando árvores. Cada operador deve ser nó interno que terá os operadores como filhos. Por exemplo, a equação 2(a-b/c) é representada pela seguinte árvore:

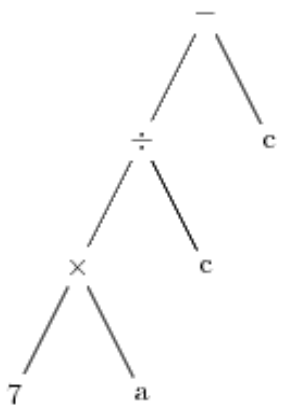


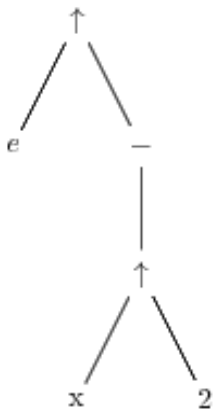
a) a + b + 5c

b) 7a/c - c

c) e^{-x^2}

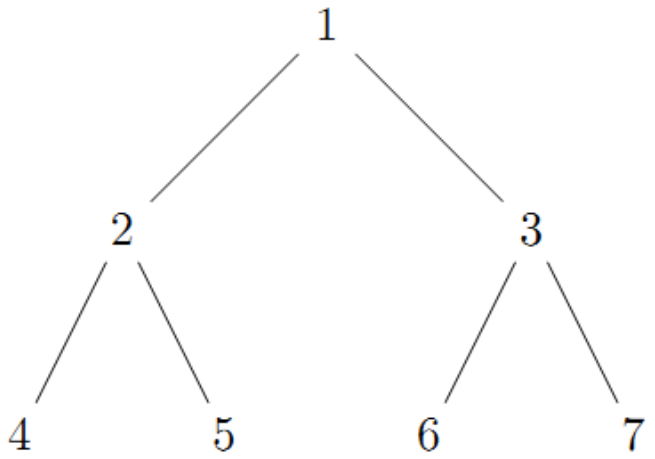






Note que os operadores de subtração, divisão e exponenciação (seta para cima) não são associativos. Nesses operadores, a ordem dos filhos é relevante.

9. Mostre o resultado de caminhamento pré-ordem, central e pós-ordem na árvore abaixo.



Pré-ordem: 1 2 4 5 3 6 7
Central: 4 2 5 1 6 3 7
Pós-ordem: 4 5 2 6 7 3 1

10. No máximo quantas chamadas recursivas podem ser feitas à função pré-ordem numa árvore binária? Quantas chamadas recursivas à função pré-ordem podem ser feitas numa árvore binária completa?

O número máximo de chamadas recursivas à função pré-ordem *é igual à altura da árvore. Em outras palavras, o número de chamadas recursivas à função* pré-ordem *é O(h), onde h é a altura da árvore. A altura máxima de uma árvore binária é N, onde N é o número de nós na árvore (isto acontece quando cada nó tem apenas um filho). Logo o número de chamadas recursivas no pior caso é O(N).*

Numa árvore binária completa, o número de chamadas recursivas à função pré-ordem *continua sendo igual à altura da árvore. Porém, uma árvore binária completa tem* $2^h - 1$ *nós, onde h é a altura da árvore. Logo, numa árvore binária completa o número de chamadas recursivas à função* pré-ordem *é O(h) = O(log(N)), onde N é o número de nós da árvore.*

11. Implemente um método para inserir um elemento numa árvore binária de pesquisa onde cada nó contém um ponteiro para seu pai, como a seguir:

```
struct arvore {
   struct arvore *pai;
   struct arvore *esquerda;
   struct arvore *direita;
   int dado;
};
```

Seu método deve ter o seguinte protótipo:

```
void insere(struct arvore *raiz, int *dado);
```

```
struct arvore {
   struct arvore *pai;
   struct arvore *esquerda;
   struct arvore *direita;
   int dado;
};

void insere(struct arvore *raiz, int *dado) {
    if(*dado < raiz->dado) {
        if (raiz->esq) {
            insere_elemento(raiz->esq, dado);
        }
        else { /* achou local de inserção */
            struct arvore *novo = cria_arvore();
            novo->dado = *dado;
            raiz->esq = novo;
            novo->pai = raiz;
        }
    } else if(*dado > raiz->dado) {
        if(raiz->dir) {
            insere_elemento(raiz->dir, dado);
        }
        else {
           struct arvore *novo = cria_arvore();
           novo->dado = *dado;
           raiz->dir = novo;
            novo->pai = raiz;
       }
    } else {
        printf("elemento já existe na árvore\n");
    }
}

struct arvore *cria_arvore(void) {
   struct arvore *novo;
   novo = malloc(sizeof(struct arvore));
   novo->esq = NULL;
   novo->dir = NULL;
   novo->pai = NULL;
   novo->dado = 0;
}
```