

Estruturas de Dados

Engenharia Informática

1º Ano - 2º Semestre

Francisco Morgado

Escola Superior de Tecnologia e Gestão de Viseu

3. ESTRUTURAS DINÂMICAS

3.1 Stacks (Pilhas)

3.2 Filas de espera

3.3 Listas ligadas ordenadas

3.4 Listas bi-ligadas ordenadas

3.5 Árvores binárias

3.6 Hashing

3.5 Árvores Binárias

Conceito

Inserção de elementos

Consultas

Travessias

Remoção de um elemento

Aplicação

Árvores Binárias – Conceito

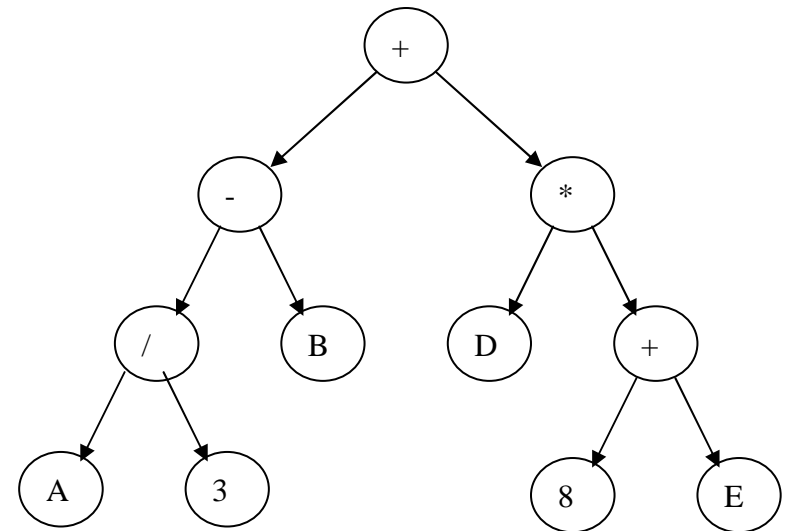
ÁRVORE é uma abstracção matemática que serve para especificar relações, descrever organizações e armazenar informação, estruturada de forma hierárquica.

Uma das aplicações informáticas mais conhecidas das árvores é o sistema de ficheiros dos sistemas operativos modernos.

Servem também para descrever sistemas de agrupamento de informação, como por exemplo relações algébricas ou lógicas.

EXEMPLO

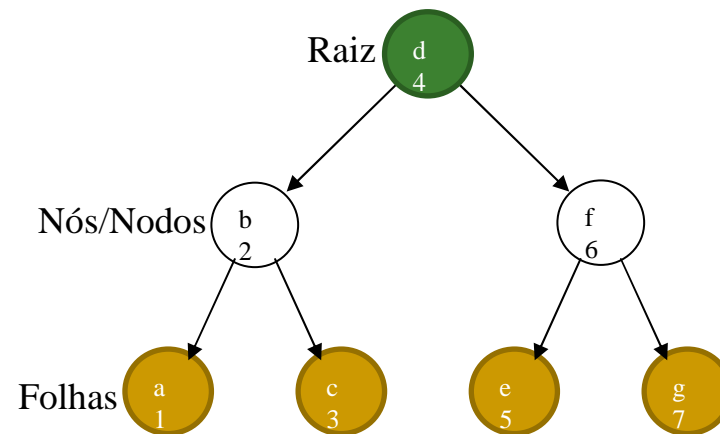
A árvore ao lado representa a expressão:
 $A/3 - B + D*(8+E)$



Uma **árvore** é uma colecção de **nós ou nodos** – os elementos que contêm a informação que se pretende armazenar – e de **arestas ou arcos** – elementos que ligam os nós.

Um **CAMINHO** numa árvore é uma lista de nós sucessivos ligados por arcos.

Numa árvore só pode existir um caminho entre dois nós.



Uma árvore com **RAIZ** é uma árvore com um nó inicial. Numa árvore com raiz existe apenas um caminho entre a raiz e qualquer nó.

Cada nó, com excepção da raiz, tem um nó acima (pai) e nós debaixo (filhos). Um nó sem filho designa-se **FOLHA**.

ALTURA de uma árvore é o número de nós internos no caminho mais longo entre a raiz e uma folha (a árvore acima tem altura dois; uma árvore vazia tem altura zero).

Estrutura

```
typedef struct
{
    char nome[80];
    char bi[12];
    int          numero;
/*chave*/
}Aluno;
```

```
typedef Aluno *Informacao;

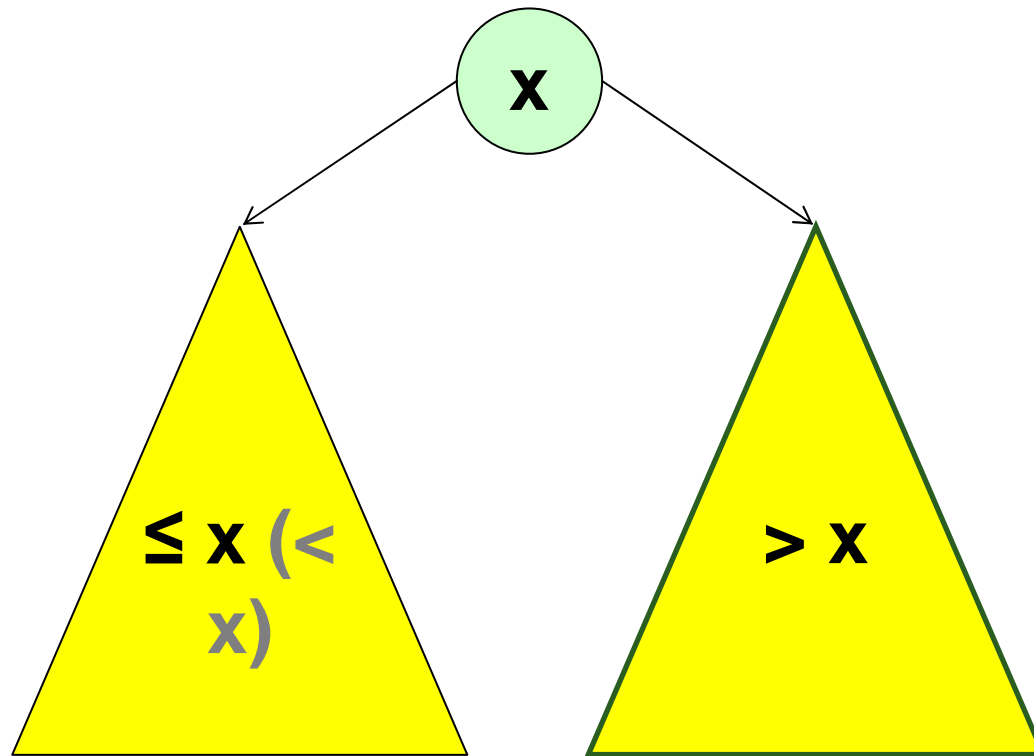
typedef struct nodo
{
    Informacao dados;
    struct nodo *esq, *dir;
}Nodo;

typedef Nodo *ArvoreBinaria;
```

Operações

- ❑ Inserir elemento na ordem correta (binária)
- ❑ Consultar/pesquisar
- ❑ Percorrer todos os nós (travessia)
- ❑ Retirar qualquer elemento

Assunções



Ordering

- Entre elementos de tipos primitivos podem fazer comparações com `==`, `!=`, `<`, `>`, `<=`, `>=`
- E com elementos de uma *struct* definida pelo programador?!!

```
enum ord{ Less = -1, Equal = 0, Great = 1};  
/*      Menor,      Igual,      Maior */  
typedef enum ord Ordering;
```

ordering.h
(no Moodle)

Para cada tipo de dados que definirmos, definimos uma função:

```
//indica se elemento1 é menor, igual ou maior que elemento2  
Ordering compara(TipoDados elemento1, TipoDados elemento2);
```

```
Ordering compara(Aluno *aluno1, Aluno *aluno2)  
{  
    if(aluno1->numero < aluno2->numero)        return Less;  
    else if(aluno1->numero > aluno2->numero)    return Great;  
    else // aluno1->numero == aluno2->numero  
                                                return Equal;  
}
```


Funções

```
bool eVazia(ArvoreBinaria raiz)
{
    return raiz == NULL;  // se a árvore está vazia
}
```

Inserção de elementos

```
void inserir(Informacao elemento, ArvoreBinaria * arvore)
{
    Nodo *nodo;
```

```
    if ( eVazia(*arvore) )
    {
        nodo = (Nodo *) malloc( sizeof(Nodo) );
        nodo->esq = NULL;
        nodo->dir = NULL;
        nodo->dados = elemento;
        *arvore = nodo;
    }
```

```
    else // se a árvore já contém elementos
    {
```

```
        nodo = *arvore;
        switch( compara(elemento, nodo->dados) )
        {
```

```
            2 case EQUAL:
              case LESS : inserir(elemento, &(amp;nodo->esq) ); break;
```

```
            3 case GREAT: inserir(elemento, &(amp;nodo->dir) ); break;
```

```
            // default: //ERRO
        }
```

```
    }
}
```

Função
Recursiva

Chamada `inserir(elemento, &raiz);`

Consultas

```
Aluno *consultar(int chave, ArvoreBinaria arvore)
```

```
{
```

```
1 if ( eVazia(arvore) )  
    return NULL;
```

```
else
```

```
    switch (comparaInt(chave, arvore->dados->numero) )
```

```
    {
```

```
2 case Equal: return arvore->dados; break;
```

```
3 case Less:  return consultar(chave, arvore->esq); break;
```

```
4 case Great: return consultar(chave, arvore->dir); break;
```

```
    // default: //ERRO
```

```
    }
```

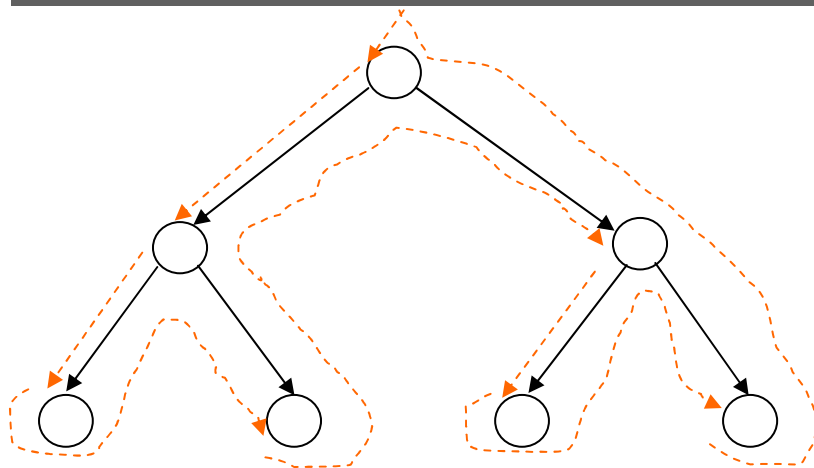
```
}
```

Função
Recursiva

Chamada

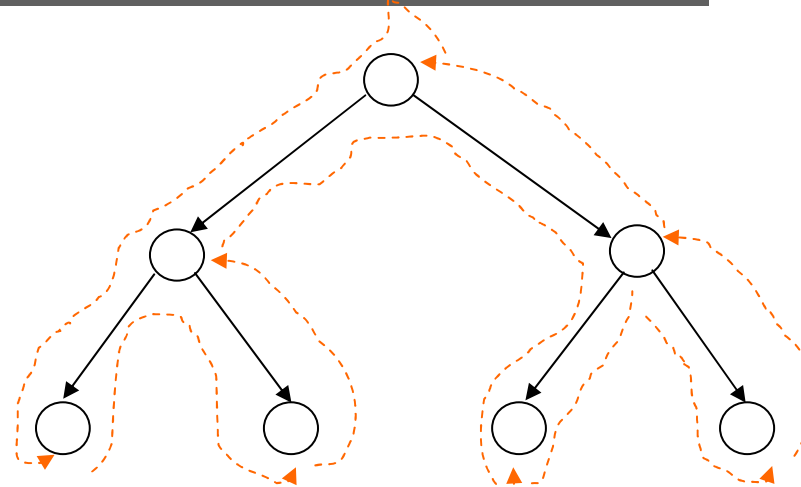
```
Aluno * aluno = consultar(nAluno, raiz);
```

Travessias



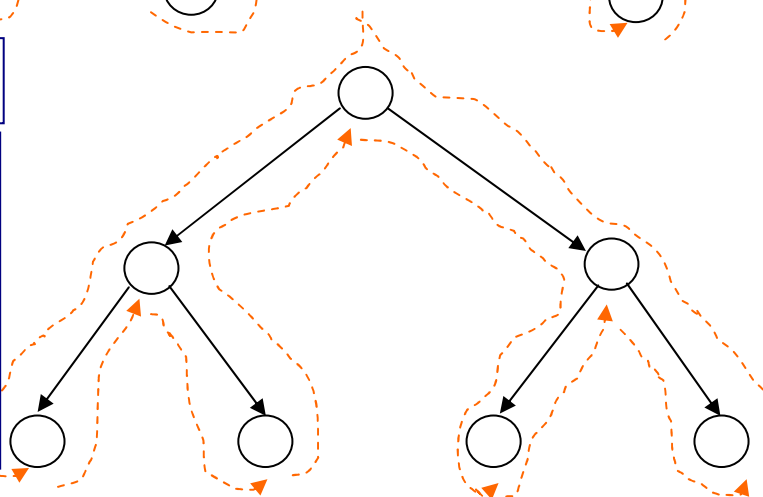
Travessia pré-ordem

1. **Visita primeiro o nó**
2. em seguida o conteúdo da sua subárvore esquerda
3. e depois o da sua subárvore direita.



Travessia pós-ordem

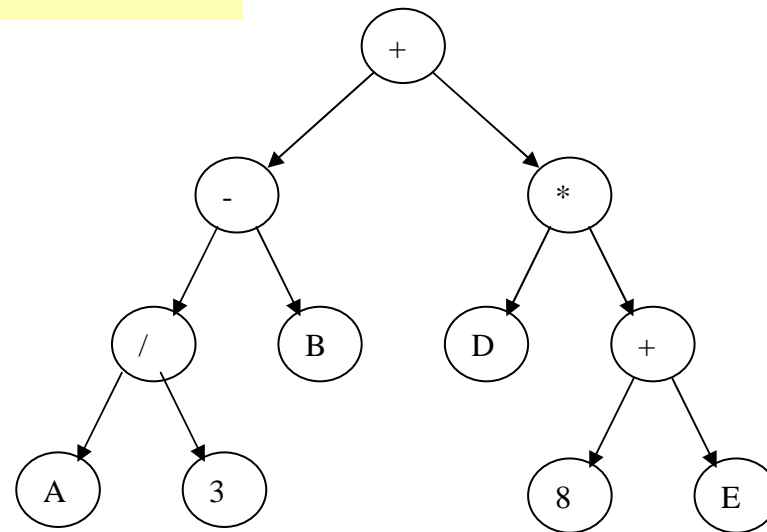
1. Visita primeiro o conteúdo da subárvore esquerda,
2. em seguida o da subárvore direita
3. **e depois o nó.**



Travessia em ordem

1. Visita primeiro o conteúdo da subárvore esquerda
2. **em seguida o nó**
3. e depois o da sua subárvore direita.

Exemplos de travessias



Travessia pré-ordem

nó -> ramo esquerdo -> ramo direito

+ - / A 3 B * D + 8 E

Travessia em ordem

ramo esquerdo -> nó -> ramo direito

A / 3 - B + D * 8 + E

*Mostra o conteúdo de
uma árvore binária por
ordem crescente*

Travessia pós-ordem

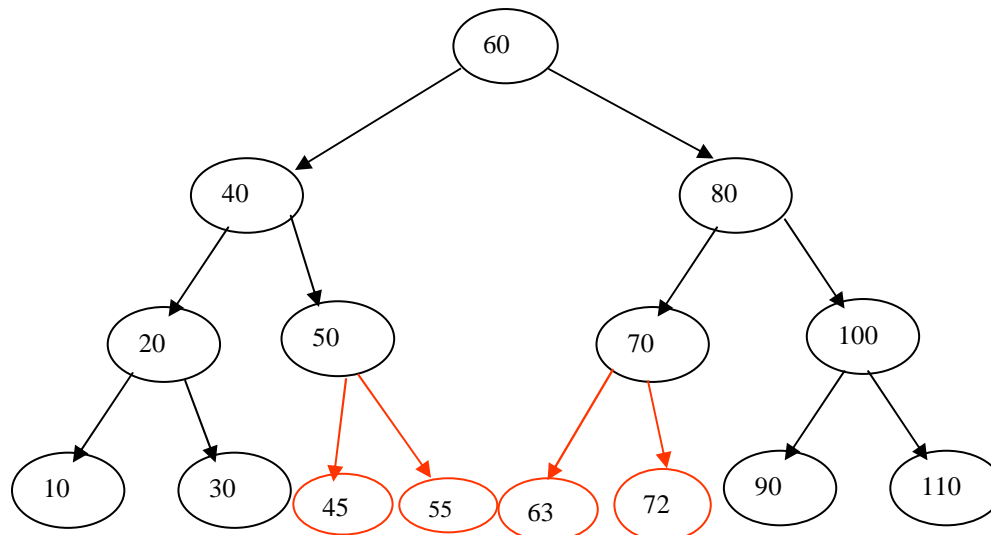
ramo esquerdo -> ramo direito -> nó

A 3 / B - D 8 E + * +

Armazenamento em ordem



10	20	30	40	45	50	55	60	63	70	72	80	90	100	110
----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----



Colocar na raiz o elemento do meio

Uma forma de balancear uma árvore, para obter uma árvore equilibrada em altura, consiste em armazenar uma sequência dos seus elementos com uma travessia em ordem e depois construir uma nova árvore fazendo uma inserção “binária” dos elementos dessa sequência.

árvore equilibrada em altura é uma árvore em que a diferença de alturas entre as subárvores esquerda e direita, em cada nó, não excede uma unidade.

A árvore binária completa tem a MELHOR altura. É a que tem o maior número de nós para uma dada altura.

Travessia pré- ordem

```
void mostraPreOrder(ArvoreBinaria arvore)
{
    if ( !eVazia(arvore) )
    {
        mostra(arvore->dados);
        mostraPreOrder(arvore->esq);
        mostraPreOrder(arvore->dir);
    }
}
```

Função
Recursiva

Chamada
mostraPreOrder(raiz);

Travessia em ordem

```
void mostraInOrder(ArvoreBinaria arvore)
{
    if ( !eVazia(arvore) )
    {
        mostraInOrder(arvore->esq);
        mostra(arvore->dados);
        mostraInOrder(arvore->dir);
    }
}
```

Função
Recursiva

Chamada
mostraInOrder(raiz);

Travessia pós-ordem

```
void mostraPosOrder(ArvoreBinaria arvore)
{
    if ( !eVazia(arvore) )
    {
        mostraPosOrder(arvore->esq);
        mostraPosOrder(arvore->dir);
        mostra(arvore->dados);
    }
}
```

Função
Recursiva

Chamada
mostraPosOrder(raiz);

Remoção de um elemento

```
Aluno * remover(int chave, ArvoreBinaria *arvore)
```

```
{
    Nodo *nodo;
    Aluno *aluno;
    if ( eVazia(*arvore) )
        return NULL;
    else
    {
        nodo = *arvore;
        switch( comparaInt(chave, nodo->dados->numero) )
        {
            case Less : return remover(chave, &(nodo->esq)); break;
            case Great: return remover(chave, &(nodo->dir)); break;
            case Equal:
                aluno = nodo->dados;
                removerNodo(arvore);
                free(nodo);
                return aluno;
                break;
            // default: //ERRO
        }
    }
}
```

Chamada

```
Aluno *aluno = remover(chave, &raiz);
```

// re-arranja a árvore
// arvore é endereço de ponteiro

Continua nos 2
diapositivos
seguintes

Remoção de um elemento – função de apoio: removeNodo()

```
void removerNodo(ArvoreBinaria *arvore) // re-arranja a árvore
{
    Nodo *nodo = *arvore; //nodo fica a apontar para o nó a remover
    Nodo *max;

    if ( eVazia(nodo->esq) )
        *arvore = nodo->dir; //nodo sobe um nível
    else if ( eVazia(nodo->dir) )
        *arvore = nodo->esq; //nodo sobe um nível
    else
    {
        max = removerNodoMax( &(nodo->esq) );
        // o máximo da subárvore esquerda é promovido
        *arvore = max;
        max->dir = nodo->dir;
        max->esq = nodo->esq;
    }
}
```

Remoção de um elemento - função removerNodoMax()

```
Nodo * removerNodoMax(ArvoreBinaria *arvore)
{
    Nodo *max = *arvore;
    Nodo *anterior = max;

    // encontrar o máximo
    while(! eVazia(max->dir) )
    {
        anterior = max;
        max = max->dir;
    }

    if(max == * arvore) // o máximo é a raiz da subárvore
        *arvore = max->esq;
    else
        anterior->dir = max->esq;
    return max;
}
```

Função main()

```
void main() {
    int nAl;
    ArvoreBinaria raiz = NULL;
    do {
        char op = menu(); // devolve a opção
        switch (op) {
            case '1':
                aluno = lerAluno(); inserir(aluno, &raiz); break;
            case '2':
                mostraInOrder(raiz); break;
            case '3':
                mostraPreOrder(raiz); break;
            case '4':
                mostraPosOrder(raiz); break;
            case '5': printf("Nº do aluno a consultar?");
                    scanf("%d", &nAl);
                    consultar(nAl, raiz);
                    if (aluno != NULL)
                        mostra(aluno);
                    break;
            case '6': printf("\nNº do aluno a retirar?"); scanf("%d", &nAl);
                    aluno = remover(nAl, &raiz);
                    if (aluno != NULL)
                        printf("Eliminou aluno %s", aluno->nome);
                    else
                        printf("\nNao existe esse aluno!");
                    break;
            case '0': exit(0); }
        }while (1);
    }
```

Exercício...

Função menu()

Função lerAluno()

Definição de Estrutura

Nó

```
typedef struct No_Arvore
{
    Aluno  *Info;
    No_Arvore *Esq, *Dir;
}No_Arv, *NO_ARV;
```

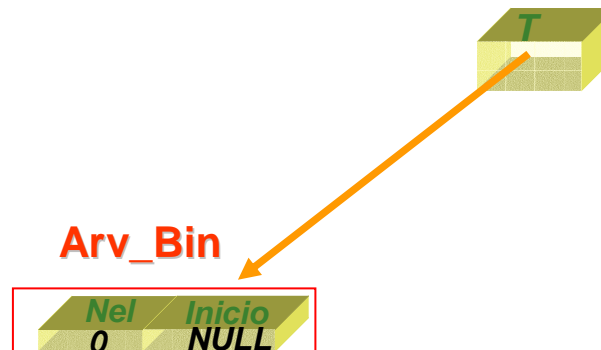
Árvore

```
typedef struct arv_bin
{
    int NEL;
    NO_ARV Inicio;
}Arv_Bin, *ARV_BIN;
```


Funções de Apoio

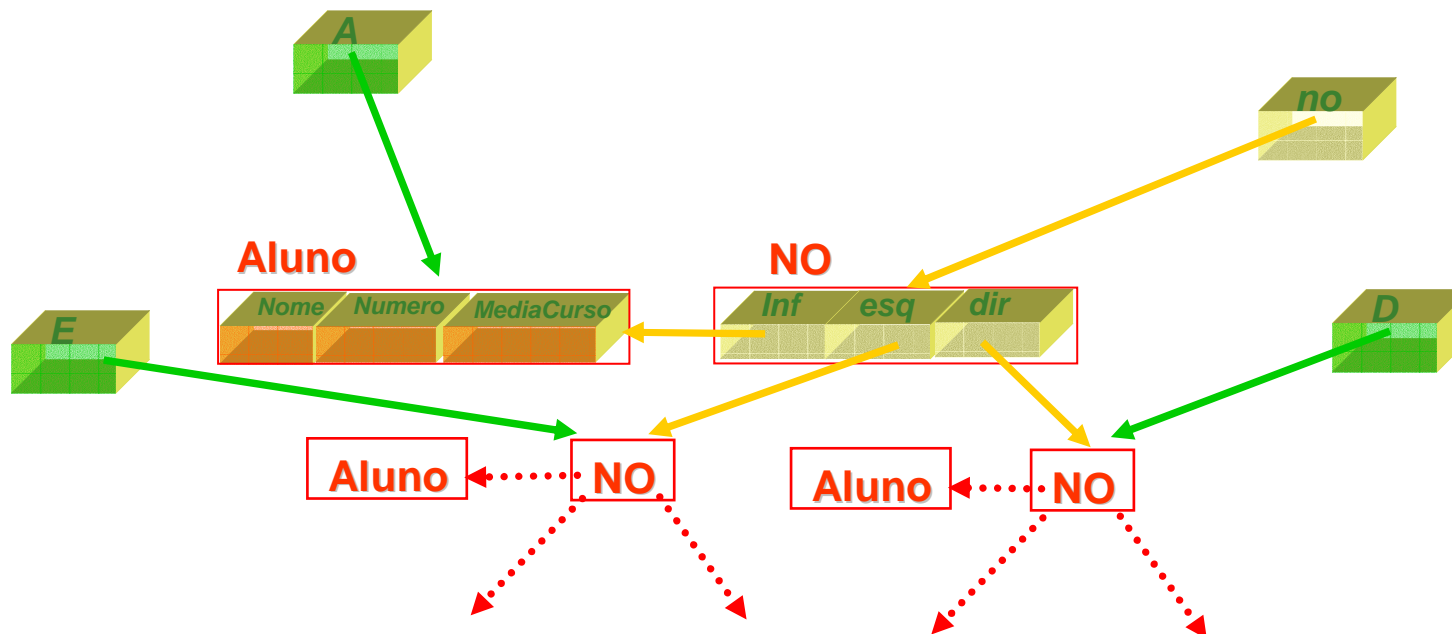
Criar uma árvore (de alunos)

```
ARV_BIN CriarArvore()  
{  
    ARV_BIN T = (ARV_BIN )malloc(sizeof(Arv_Bin));  
    T->Inicio = NULL;  
    T->NEL = 0;  
    return T;  
}
```



Criar um nó da árvore binária

```
NO_ARV Criar_No_Arvore(Aluno *A, NO_ARV E, NO_ARV D)
{
    NO_ARV no = (NO_ARV)malloc(sizeof(No_Arv));
    no->Info = A;
    no->Esq = E;
    no->Dir = D;
    return no;
}
```



Comparar dois alunos através do respectivo número

```
bool CriterioComparacao(Aluno *A, Aluno *B)
{
    return A->Numero < B->Numero;
}
```

Inserção de elementos

```
void InserirAluno(NO_ARV nt, Aluno *A)
```

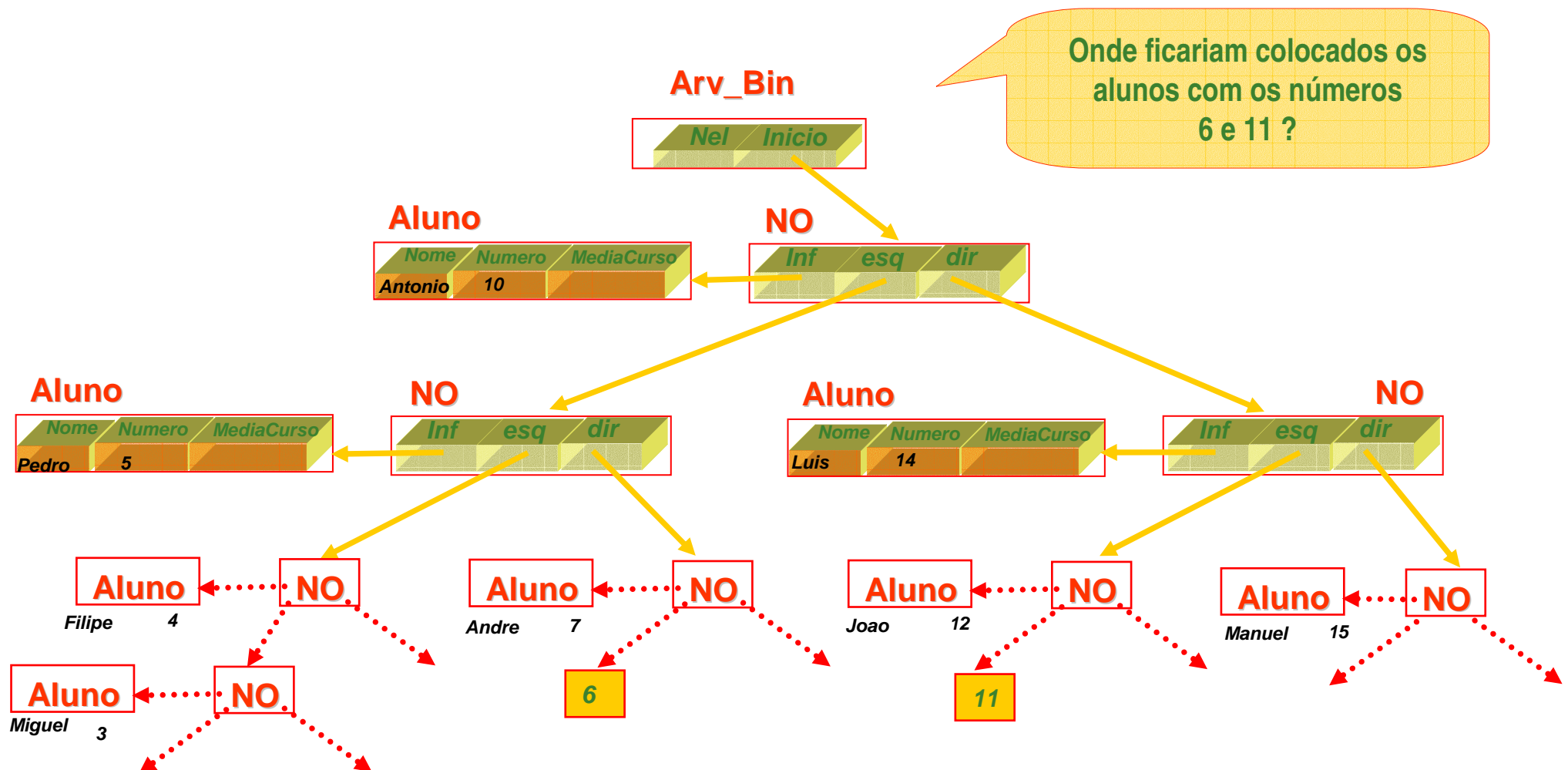
```
{
    if (CritérioComparacao(A, nt->Info))
    {
        if (nt->Esq) InserirAluno(nt->Esq, A);
        else nt->Esq = Criar_No_Arvore(A, NULL, NULL);
    }
    else
    {
        if (nt->Dir) InserirAluno(nt->Dir, A);
        else nt->Dir = Criar_No_Arvore(A, NULL, NULL);
    }
}
```

Inserção de um aluno na árvore, de acordo com o critério estabelecido (ordenando pelo campo *número*)

```
void InserirArvoreAluno(ARV_BIN T, Aluno *A)
```

```
{
    if (!T) return;
    if (!(T->Inicio))
        T->Inicio = Criar_No_Arvore(A, NULL, NULL);
    else
        InserirAluno(T->Inicio, A);
    T->NEL++;
}
```

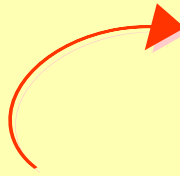
Simulação



Listar a informação de uma árvore

```
void Mostrar_Informacao(Aluno *A)
{
    printf("Nome do Aluno= %s\n",A->Nome);
    printf("Numero do Aluno= %d\n",A->Numero);
    printf("Media do Curso= %f\n",A->MediaCurso);
}
```

```
void MostrarArvore (ARV_BIN T)
{
    if (!T) return;
    if (T->Inicio)
        Mostrar_pre_ordem(T->Inicio);
        //Mostrar_em_ordem(T->Inicio);
        //Mostrar_pos_ordem(T->Inicio);
}
```



Diferentes modos de travessia de uma árvore

Travessia pré-ordem

nó -> ramo esquerdo -> ramo direito

```
void Mostrar_pre_ordem(NO_ARV nt)
{
    if (!nt) return;
    Mostrar_Informacao(nt->Info);
    if (nt->Esq) Mostrar_pre_ordem(nt->Esq);
    if (nt->Dir) Mostrar_pre_ordem(nt->Dir);
}
```

```
void Mostrar_em_ordem(NO_ARV nt)
{
    if (!nt) return;
    if (nt->Esq)
        Mostrar_em_ordem(nt->Esq);
    Mostrar_Informacao(nt->Info);
    if (nt->Dir)
        Mostrar_em_ordem(nt->Dir);
}
```

Travessia em ordem

ramo esquerdo -> nó -> ramo direito

```
void Mostrar_pos-ordem(NO_ARV nt)
{
    if (!nt) return;
    if (nt->Esq)
        Mostrar_pos_ordem(nt->Esq);
    if (nt->Dir)
        Mostrar_pos_ordem(nt->Dir);
    Mostrar_Informacao(nt->Info);
}
```

Travessia pós ordem

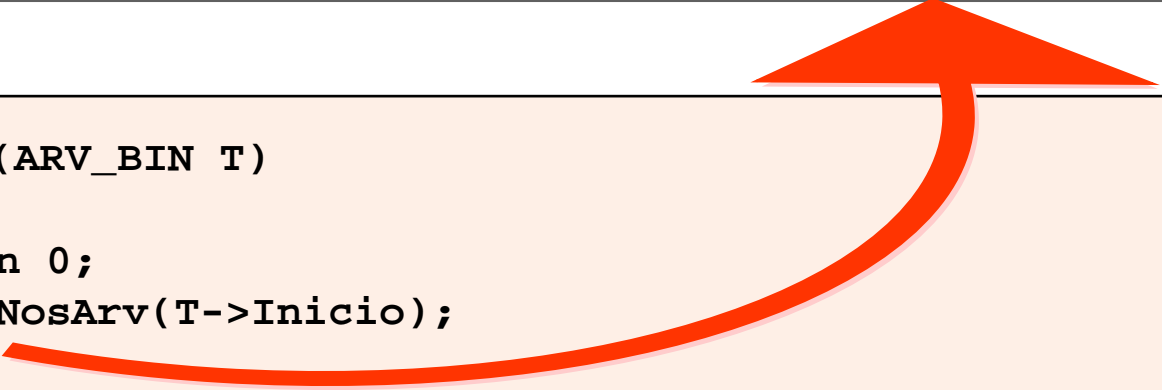
ramo esquerdo -> ramo direito -> nó

Consultas

Determinar quantos alunos estão registrados na árvore

```
int ContarNosArv(NO_ARV nt)
{
    if (!nt) return 0;
    return 1 + ContarNosArv(nt->Esq) + ContarNosArv(nt->Dir);
}
```

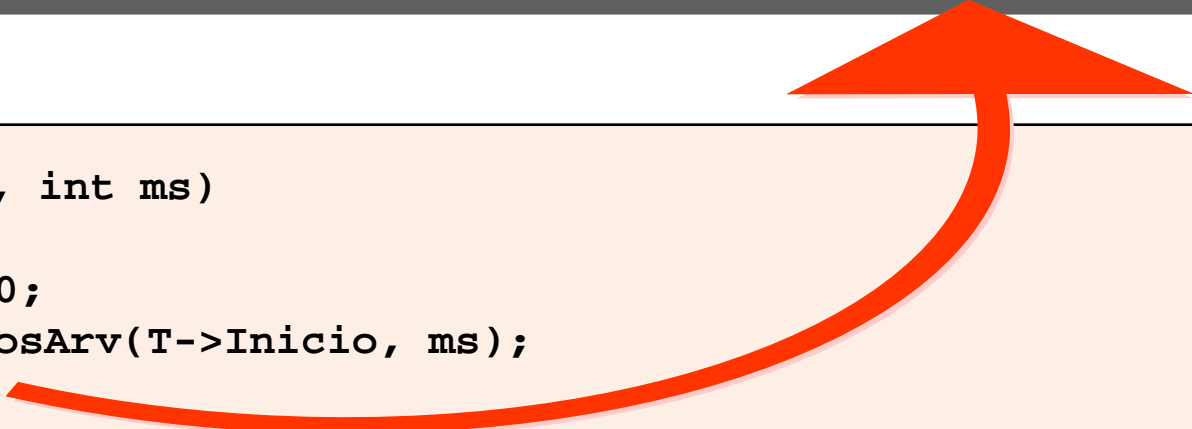
```
int ContarNosArvore(ARV_BIN T)
{
    if (!T) return 0;
    return ContarNosArv(T->Inicio);
}
```



Determinar quantos alunos têm número superior a *ms*

```
int ContSupNosArv(NO_ARV nt, int ms)
{
    if (!nt) return 0;
    if (nt->Info->Numero > ms)
        return 1 + ContSupNosArv(nt->Esq, ms) + ContSupNosArv(nt->Dir, ms);
    else
        return ContSupNosArv(nt->Esq, ms) + ContSupNosArv(nt->Dir, ms);
}
```

```
int ContSup(ARV_BIN T, int ms)
{
    if (!T) return 0;
    return ContSupNosArv(T->Inicio, ms);
}
```

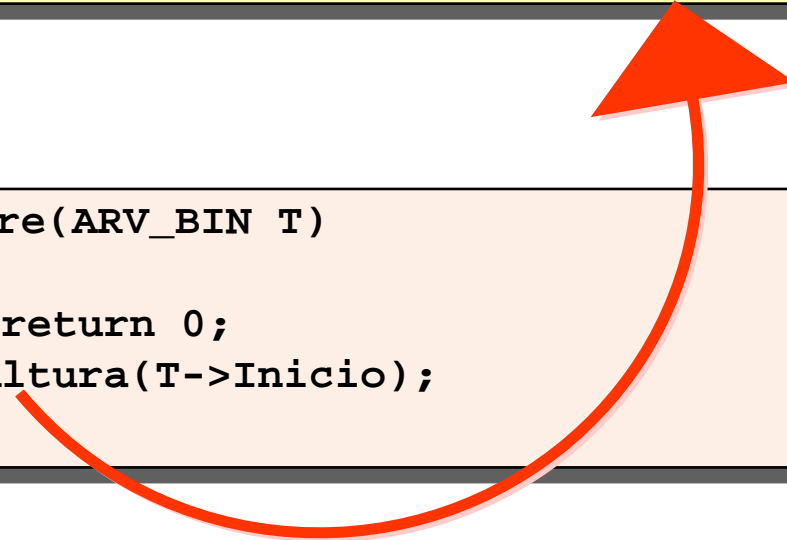


Determinar a altura de uma árvore

```
int Altura(NO_ARV nt)
{
    if (!nt) return 0;
    int alt_esq = Altura(nt->Esq);
    int alt_dir = Altura(nt->Dir);
    if (alt_esq > alt_dir)
        return alt_esq + 1;
    else
        return alt_dir + 1;
}
```

ALTURA de uma árvore é o número de nós sem filhos no caminho mais longo entre a raiz e uma folha.

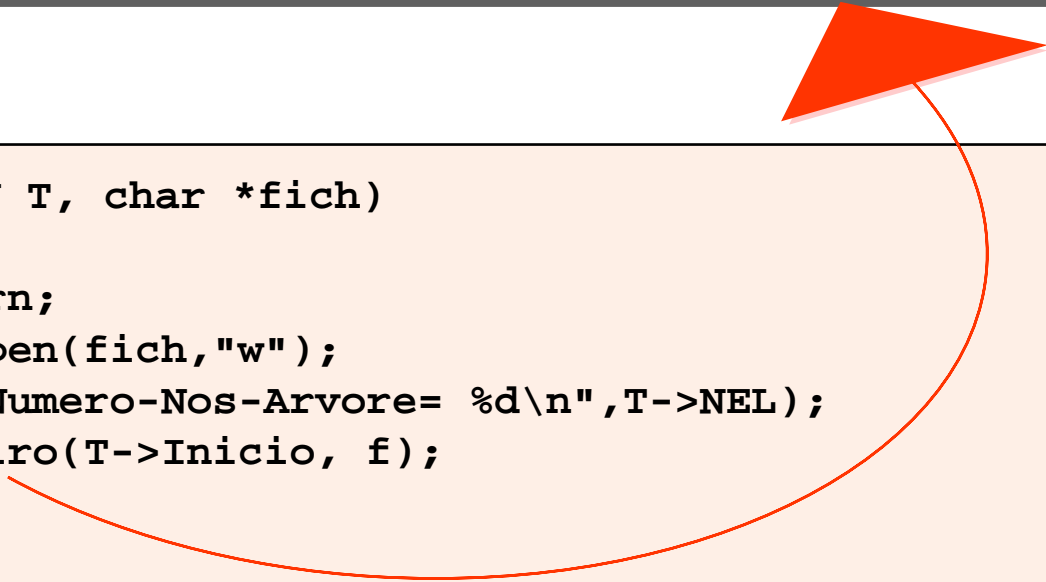
```
int AlturaArvore(ARV_BIN T)
{
    if (!T) return 0;
    return Altura(T->Inicio);
}
```



Gravar toda a estrutura para ficheiro

```
void ColocarFicheiro(NO_ARV nt, FILE *f)
{
    if (!nt) return;
    Gravar_Informacao_Ficheiro(f, nt->Info);
    if (nt->Esq) ColocarFicheiro(nt->Esq, f);
    if (nt->Dir) ColocarFicheiro(nt->Dir, f);
}
```

```
void Gravar(ARV_BIN T, char *fich)
{
    if (!T) return;
    FILE *f = fopen(fich, "w");
    fprintf(f, "Numero-Nos-Arvore= %d\n", T->NEL);
    ColocarFicheiro(T->Inicio, f);
    fclose(f);
}
```



Colocar todos os alunos da árvore numa lista

```
void ConstruirLista(NO_ARV nt, LISTA L)
{
    if (!nt) return;
    InserirInicioLista(l, CriarCopiaAluno(nt->Info));
    if (nt->Esq) ConstruirLista(nt->Esq, L);
    if (nt->Dir) ConstruirLista(nt->Dir, L);
}
```

```
Lista *ColocarLista(ARV_BIN T)
{
    if (!T) return NULL;
    Lista *la = CriarLista();
    ConstruirLista(T->Inicio, la);
    return la;
}
```

Trata-se de criar uma lista com os alunos que estão na árvore.
Por essa razão, vai sendo criada uma cópia de cada aluno a inserir na lista.


Colocar todos os alunos da árvore numa lista (funções de apoio)

```
Aluno *CriarCopiaAluno(Aluno *A)
{
    return CriarUmAluno(A->Nome, A->Numero, A->MediaCurso);
}
```

```
Aluno *CriarUmAluno(char *nome, int num, float media)
{
    Aluno *A = (Aluno *)malloc(sizeof(Aluno));
    //A fica a apontar para o bloco criado, do tipo Aluno
    strcpy(A->Nome, nome);
    A->Numero = num;
    A->MediaCurso = media;
    return A;
}
```

Remoção de elementos

```
void RetirarFolhas(ARV_BIN T)
{
    if (!T) return;
    if (NoFolha(T->Inicio))
    {
        DestruirNosArvore(T->Inicio);
        T->Inicio = NULL;
        T->NEL--;
    }
    else
        T->NEL = RetirarFolhasNos(T->Inicio);
}
```



Remover todas as
folhas da árvore

```
bool NoFolha(NO_ARV nt)
{ // Um dado No é uma folha quando Esq é Null e Dir é Null
    if (!nt) return true; // Se Nó nulo => estamos perante uma folha
    if (!nt->Dir && !nt->Esq)
        return true; // é Folha
    else return false;
}
```

Remover todas as folhas da árvore (continuação)

```
int RetirarFolhasNos(NO_ARV nt)
{
    int nfolhas_ret = 0;
    if (NoFolha(nt->Esq))
    {
        DestruirNosArvore(nt->Esq);
        nt->Esq = NULL;
        nfolhas_ret++;
    }
    else // a parte ESq não é Folha
        nfolhas_ret = nfolhas_ret + RetirarFolhasNos(nt->Esq);
    if (NoFolha(nt->Dir))
    {
        DestruirNosArvore(nt->Dir);
        nt->Dir = NULL;
        nfolhas_ret++;
    }
    else // a parte Dir não é Folha
        nfolhas_ret = nfolhas_ret + RetirarFolhasNos(nt->Dir);
    return nfolhas_ret;
}
```

Destruir uma árvore

```
void DestruirNosArvore(NO_ARV nt)
{
    if (!nt) return;
    DestruirNosArvore(nt->Dir); // destruimos o ramo Direito
    DestruirNosArvore(nt->Esq); // destruimos o ramo Esquerdo
    free (nt->Info);
    free (nt);
}
```

```
void DestruirArvore(ARV_BIN t)
{
    if (!t) return;
    if (t->Inicio)
        DestruirNosArvore(t->Inicio);
    free(t);
}
```