



MC3305

Algoritmos e Estruturas de Dados II

Aula 07 – Árvores

Prof. Jesús P. Mena-Chalco
jesus.mena@ufabc.edu.br

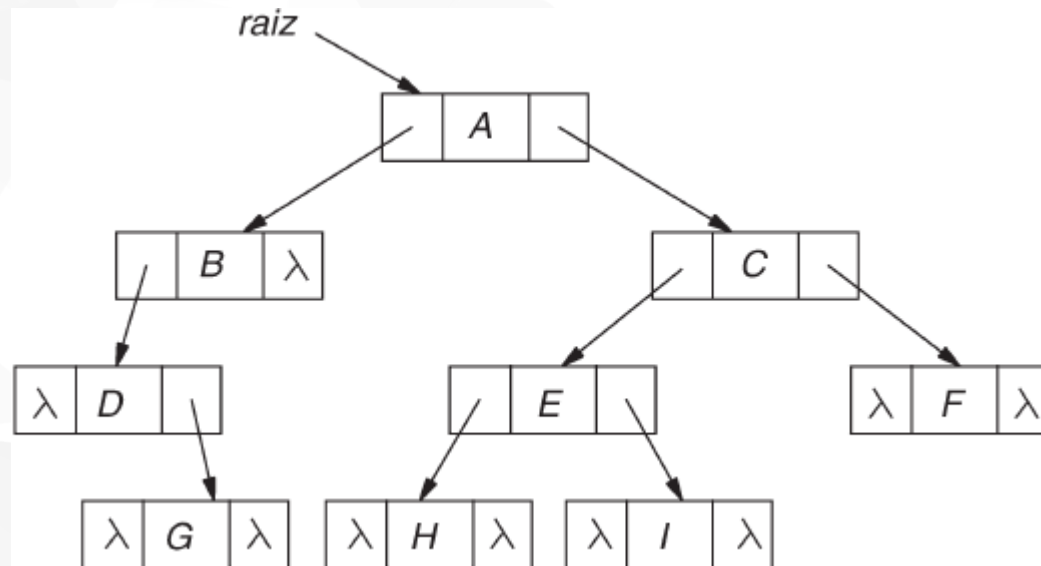
2Q-2015

Uma árvore binária



Representação de uma árvore binária

$\lambda = \text{NULL}$



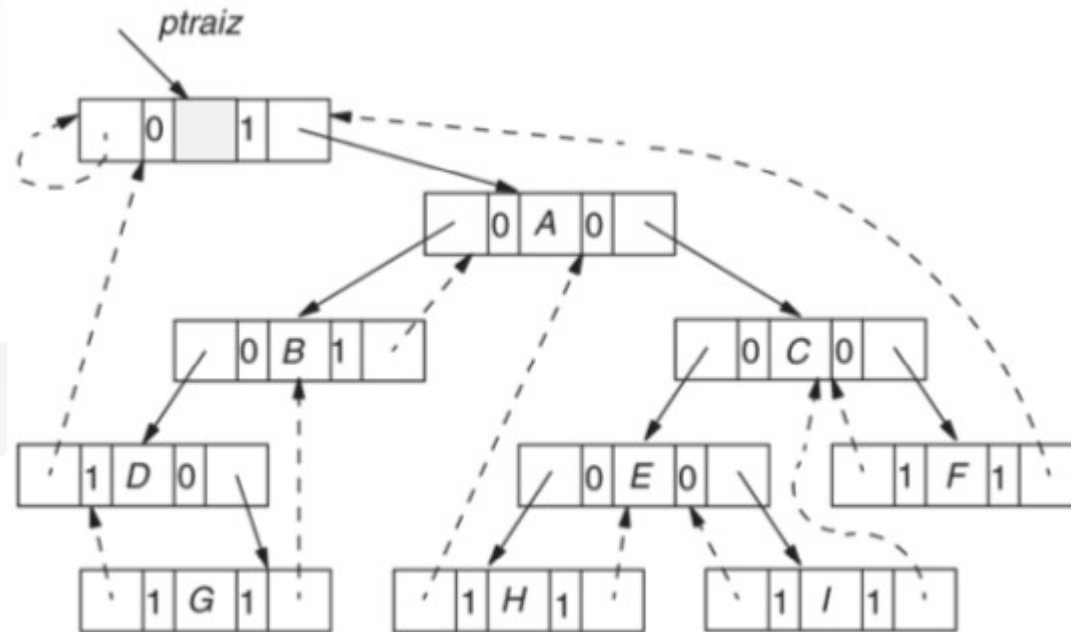
Para uma árvore binária de n vértices:

São requeridas $2n+1$ unidades de memória para sua representação

$n+1$ unidades de memória são iguais a NULL.

Por que não aproveitar esse espaço de memória?

Árvore binária com costura



Sem uso de uma pilha é facil percorrer a árvore.

Nós, filhos e pais

```
struct cel {  
    int      conteudo; //  
    struct cel *esq;  
    struct cel *dir;  
};  
typedef struct cel no; //
```

conteudo

999



esq dir

```
struct cel {  
    int      conteudo;  
    struct cel *pai;  
    struct cel *esq;  
    struct cel *dir;  
};  
typedef struct cel no;
```

pai



999



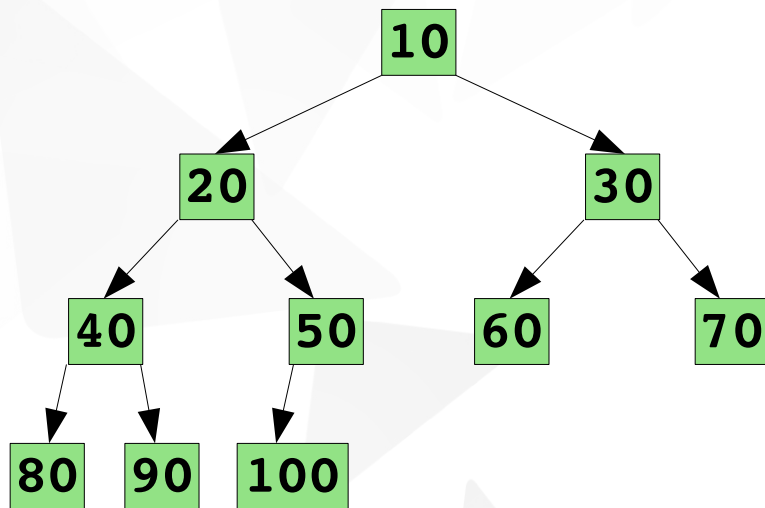
esq dir



Teste01.c

teste01.c

-1 10 20 30 40 50 60 70 80 90 100



```
void imprimirFolhas (no *r) {  
    if (r!=NULL) {  
        imprimirFolhas(r->esq);  
        if (r->esq==NULL & r->dir==NULL)  
            printf("%d\t", r->conteudo);  
        imprimirFolhas(r->dir);  
    }  
}
```

Varredura e-r-d:

80 40 90 20 100 50 10 60 30 70

Folhas da arvore:

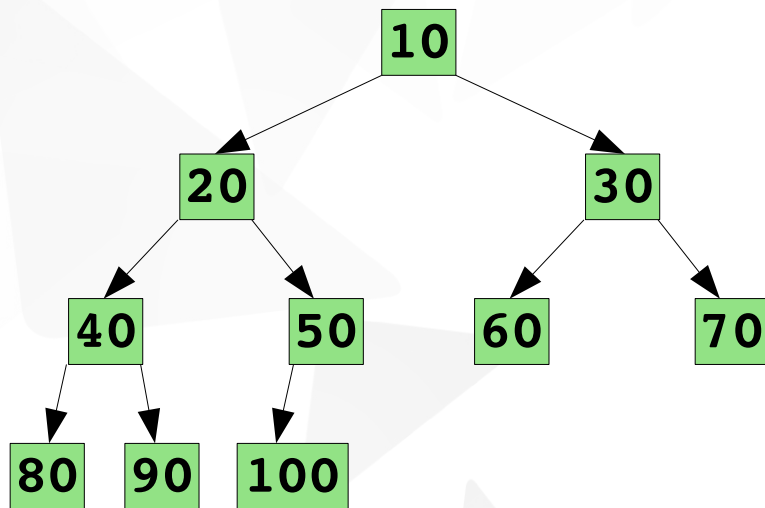
80 90 100 60 70

Altura da arvore:

3

teste01.c (atividade)

-1	10	20	30	40	50	60	70	80	90	100
----	----	----	----	----	----	----	----	----	----	-----

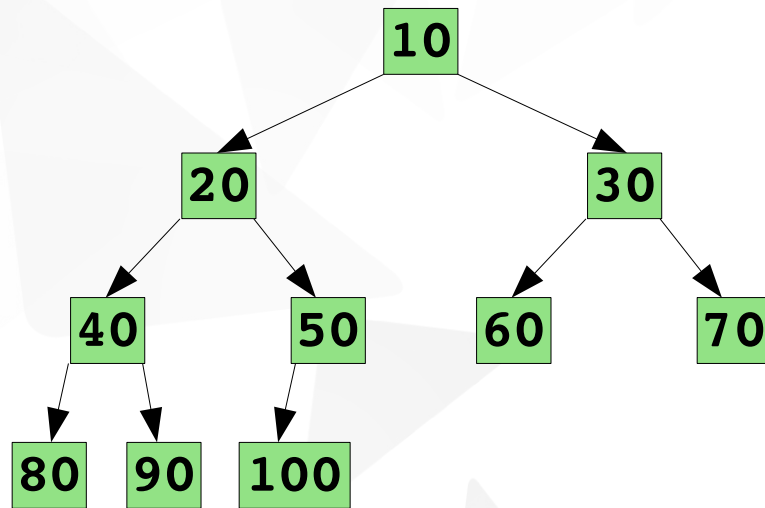


Crie uma função que permita preencher corretamente o ponteiro para o pai de cada nó.

```
// Recebe a raiz r de uma árvore binária.  
// Preenche corretamente o pai de cada nó  
void preenchePai(no *r) {  
    //...  
}
```


teste01.c (solução)

-1	10	20	30	40	50	60	70	80	90	100
----	----	----	----	----	----	----	----	----	----	-----



Crie uma função que permita preencher corretamente o ponteiro para o pai de cada nó.

```
void preenchePaiDadoFilho(no *pai, no *filho) {
    if (filho!=NULL) {
        filho->pai = pai;
        preenchePaiDadoFilho(filho, filho->esq);
        preenchePaiDadoFilho(filho, filho->dir);
    }
}

void preenchePai(no *r) {
    preenchePaiDadoFilho(r, r);
}
```

```
preenchePai(raiz);
no* ultimo = ultimoErd(raiz);
printf("\nPai do ultimo no e-r-d:\n%d", ultimo->pai->conteudo);
```

Pai do ultimo no e-r-d:

30

teste01.c (outras respostas)

// Bruna Gomes

```
void preenchePai(no *r) {
    if (r!=NULL) {
        if (r->esq!=NULL){
            r->esq->pai = r;
            preenchePai(r->esq);
        }
        if(r->dir != NULL){
            r->dir->pai = r;
            preenchePai(r->dir);
        }
    }
    if(r->pai == NULL){
        r->pai = r;
    }
}
```

// Augusto Abreu

```
void preenchePai(no *r) {
    r->pai = r;
    preenche(r);
}

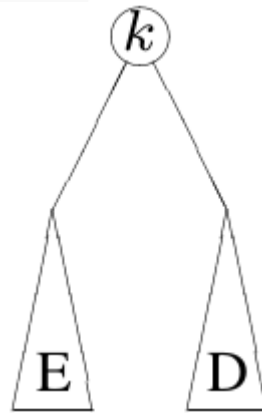
void preenche(no *r) {
    if (r->esq != NULL) {
        (r->esq)->pai = r;
        preenche(r->esq);
    }
    if (r->dir != NULL) {
        (r->dir)->pai = r;
        preenche(r->dir);
    }
}
```



Árvore binária de busca Árvore de busca binária

Árvores binárias de busca

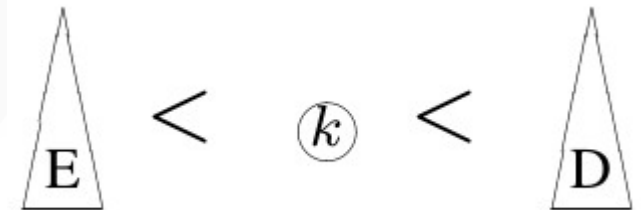
Para qualquer nó que contenha um registro:



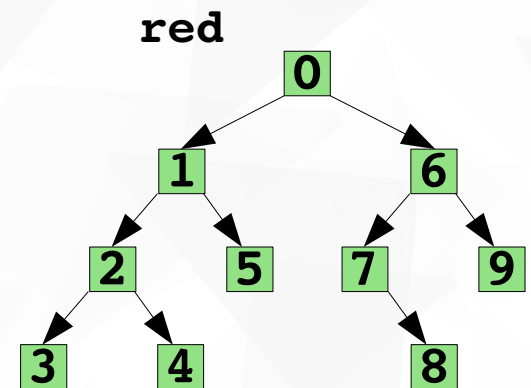
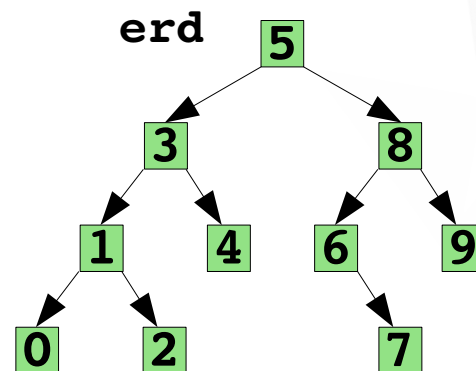
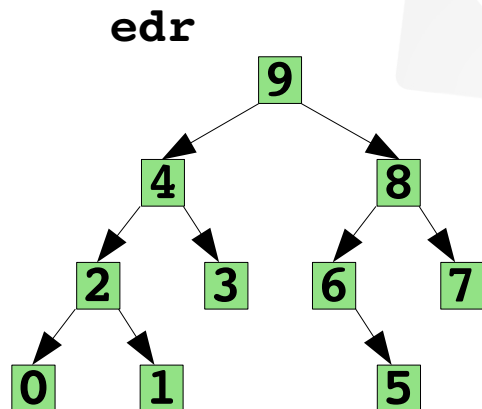
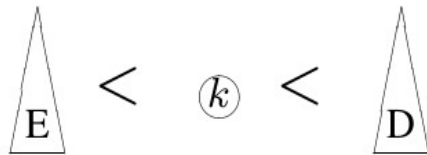
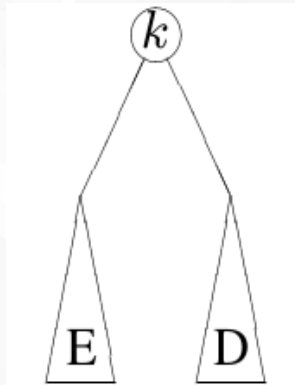
Chaves únicas!

Temos a relação invariante:

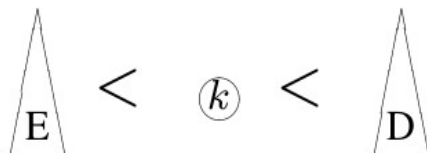
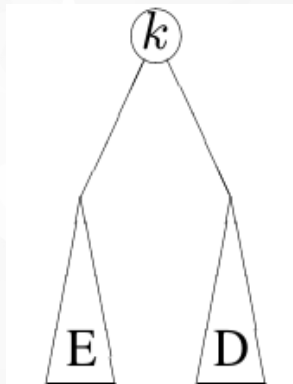
1. Todos os registros com chaves menores estão na sub-árvore à esquerda.
2. Todos os registros com chaves maiores estão na sub-árvore à direita.



Árvores binárias de busca

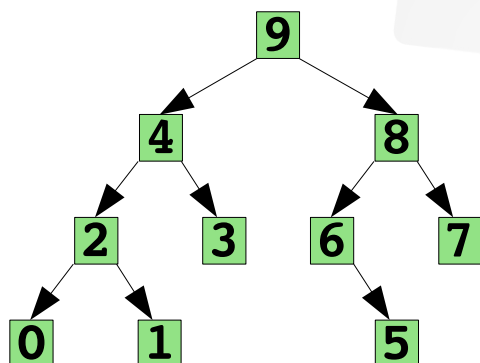


Árvores binárias de busca

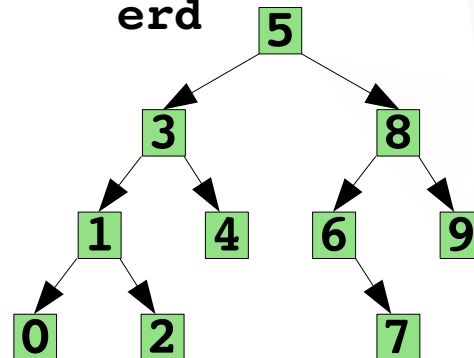


Em uma ABB
a ordem e-r-d das chaves
é crescente!

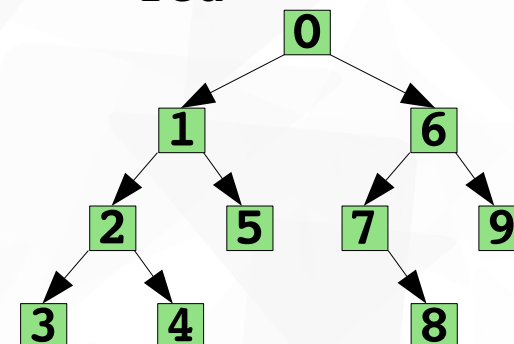
edr



erd

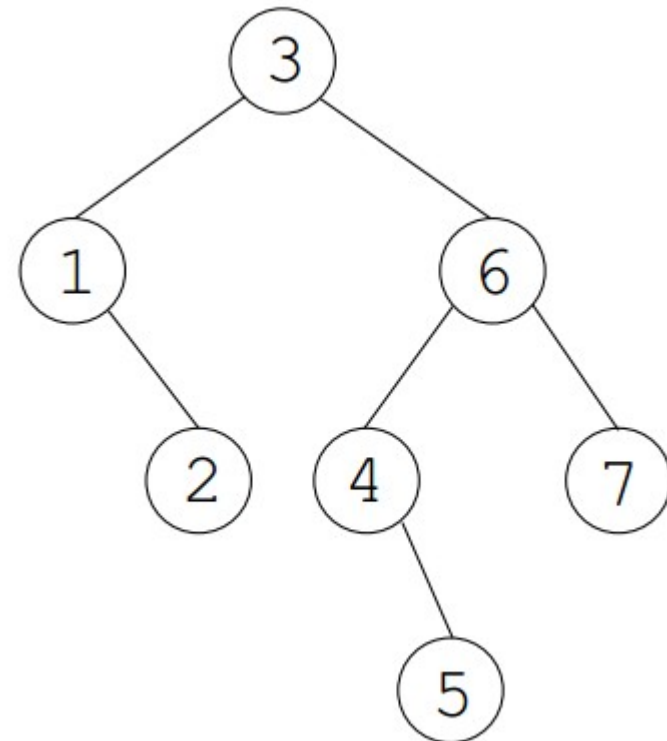


red

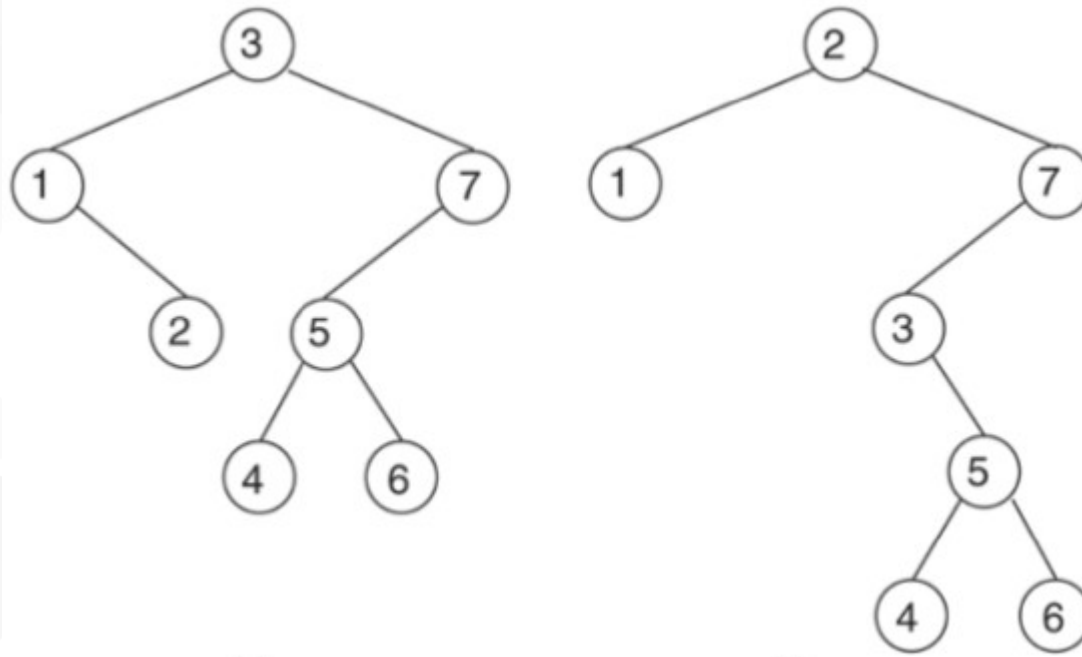


Árvores binárias de busca

- As ABB permitem minimizar o tempo de acesso no pior caso.
- Para cada chave, separe as restantes em **maiores** ou **menores**.
- A estrutura hierárquica com divisão binária: **uma árvore binária**.



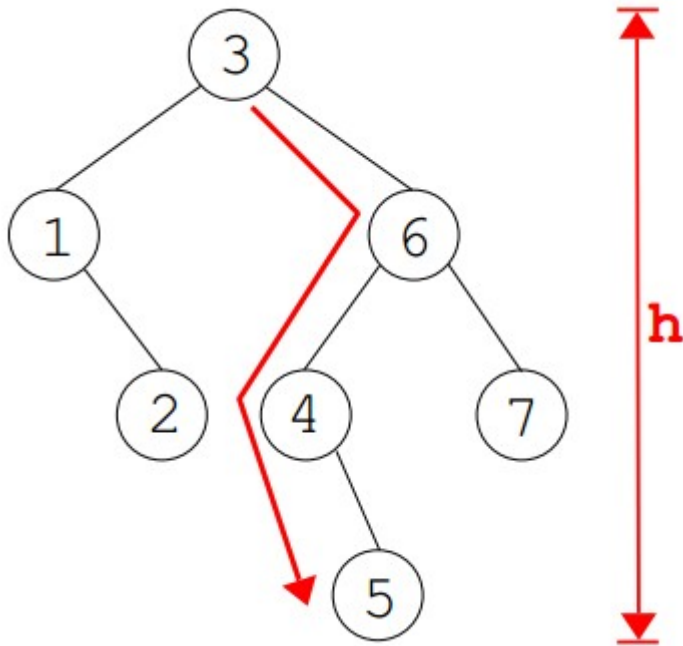
Árvores binárias de busca



Ambas árvores binárias contêm as mesmas chaves, mas sua estrutura é diferente

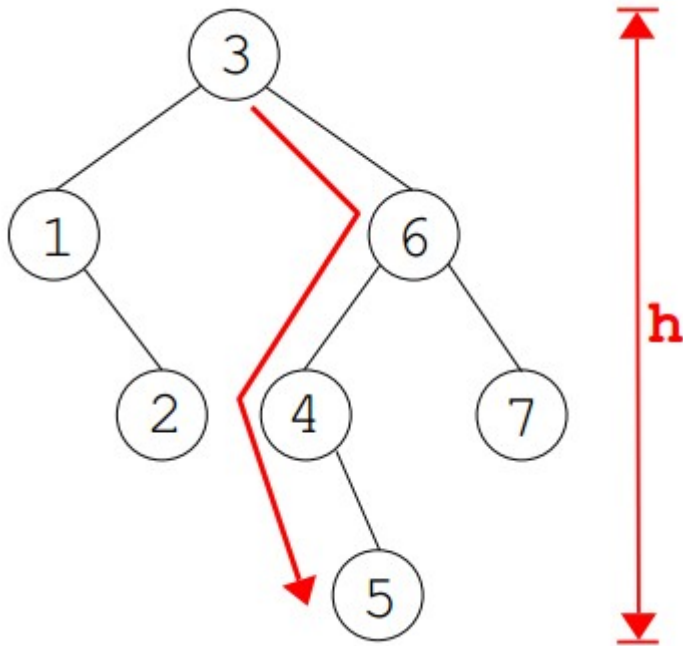
Complexidade de busca em uma ABB

- Busca em ABB = **caminho da raiz até a chave desejada**
(ou até a folha, caso a chave não exista)



Complexidade de busca em uma ABB

- Busca em ABB = **caminho da raiz até a chave desejada**
(ou até a folha, caso a chave não exista)



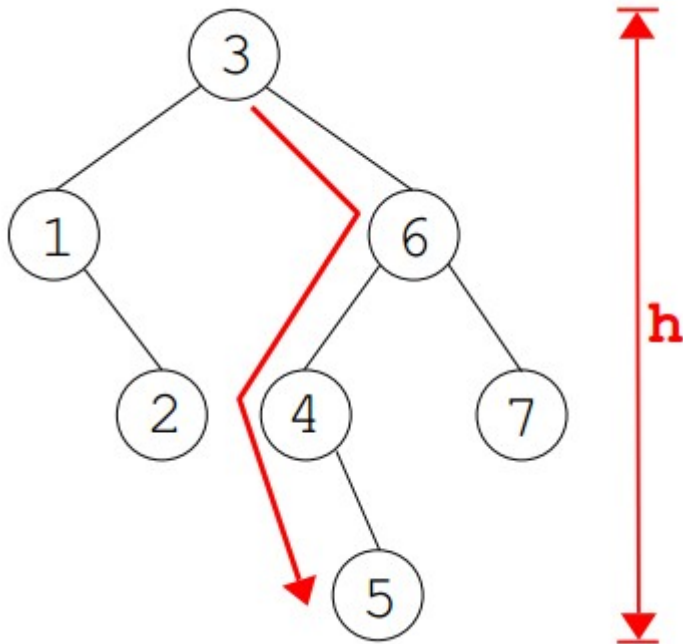
Pior caso:

Maior caminho até a folha = altura da árvore

Complexidade: $O(h)$

Complexidade de busca em uma ABB

- Busca em ABB = **caminho da raiz até a chave desejada**
(ou até a folha, caso a chave não exista)



Pior caso:

Maior caminho até a folha = altura da árvore

Complexidade: $O(h)$

*Uma árvore binária balanceada é aquela
com altura $O(\lg n)$*

Complexidade de busca em uma ABB

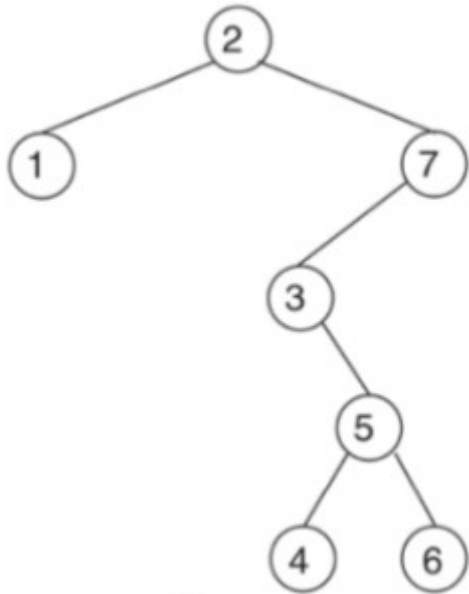
n	lg(n)
2	1
32	5
512	9
8192	13
131072	17
2097152	21
33554432	25
536870912	29
8589934592	33
137438953472	37
2199023255552	41
35184372088832	45
562949953421312	49
9007199254740990	53
144115188075856000	57
2305843009213690000	61
36893488147419100000	65
5.9029581035871E+020	69
9.4447329657393E+021	73
1.5111572745183E+023	77
2.4178516392293E+024	81
3.8685626227668E+025	85
6.1897001964269E+026	89
9.9035203142831E+027	93
1.5845632502853E+029	97
2.5353012004565E+030	101
4.0564819207303E+031	105
6.4903710731685E+032	109
1.0384593717070E+034	113
8.3076749736557E+034	116

*Uma árvore binária balanceada é aquela
com altura* $O(\lg n)$

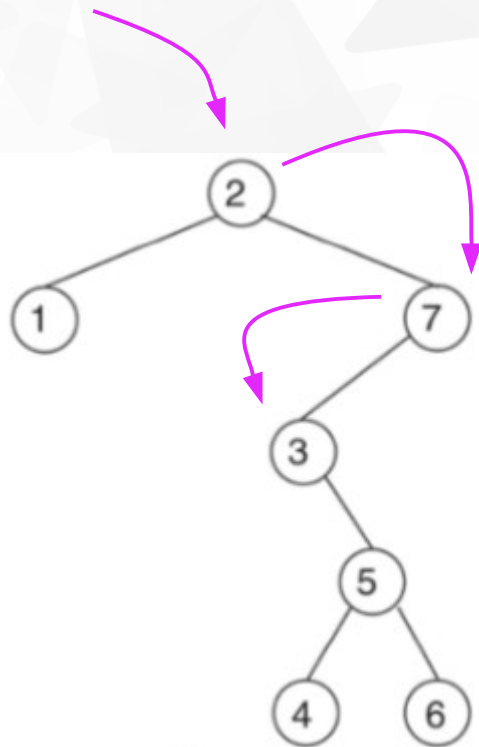
Uma árvore binária é balanceada (ou equilibrada) se,
em cada um de seus nós, as subárvores esquerda e direita
tiverem aproximadamente a mesma altura.

Busca de uma chave em uma ABB

Chave = 3



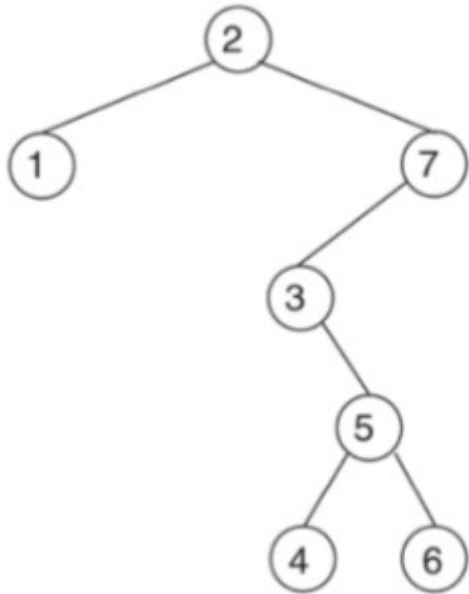
Busca de uma chave em uma ABB



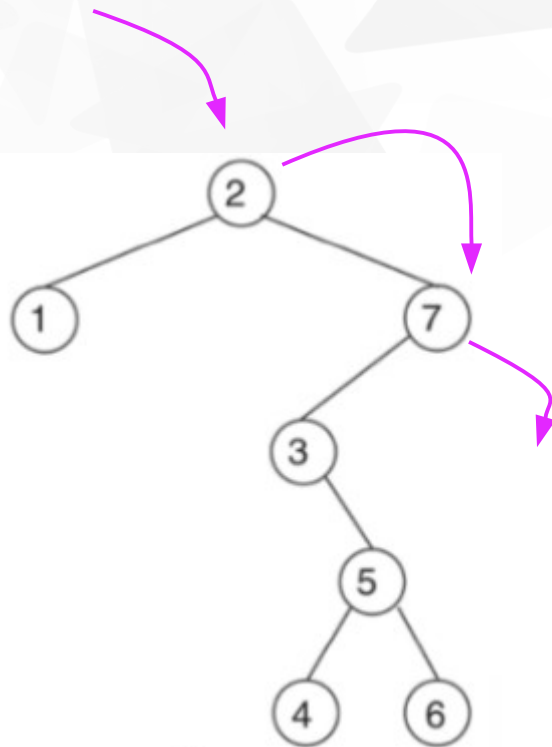
Chave =3

Busca de uma chave em uma ABB

Chave = 30

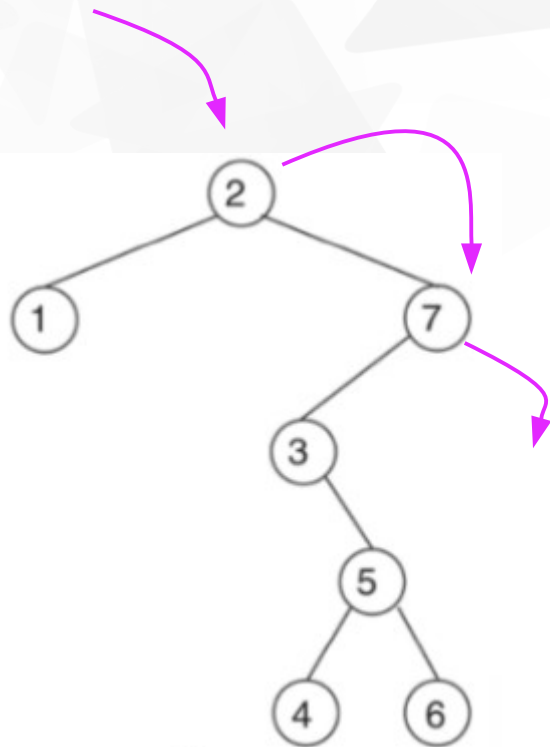


Busca de uma chave em uma ABB



Chave = 30

Busca de uma chave em uma ABB



Chave = 30

```
no* busca(no* r, int chave) {  
    if (r==NULL || r->conteudo==chave)  
        return r;  
    if (r->conteudo > chave)  
        return busca(r->esq, chave);  
    else  
        return busca(r->dir, chave);  
}
```

Função que recebe uma chave e a raiz da árvore e devolve o nó cujo conteúdo for igual a chave. Se o nó não existir



Teste02.c

Teste02.c (implemente)

```
no* inserirNoNaArvore(no* r, int valor) {  
    //..  
}  
  
// -----  
int main(int argc, char *argv[])  
{  
    int i, valor;  
    int n = atoi(argv[1]);  
    no *raiz = NULL;  
  
    for (i=0; i<n; i++) {  
        scanf("%d", &valor);  
        raiz = inserirNoNaArvore(raiz, valor);  
    }  
  
    // Altura da arvore  
    printf("\nAltura da arvore:%d", altura(raiz));  
}
```

```
$ gcc teste02.c -o teste02.exe  
$ ./teste02.exe 10  
1 2 3 4 5 6 7 8 9 0
```

Teste02.c (solução)

```
no* inserirNoNaArvore(no* r, int valor) {
    no *filho, *pai;

    if (busca(r,valor)==NULL){
        // criacao do novo no
        no* novoNo = (no*) malloc(sizeof(no));
        novoNo->conteudo = valor;
        novoNo->esq = novoNo->dir = NULL;

        if (r==NULL)
            return novoNo;
        else {
            filho = r;
            while(filho!=NULL) {
                pai = filho;
                if (filho->conteudo > novoNo->conteudo)
                    filho = filho->esq;
                else
                    filho = filho->dir;
            }
            if (pai->conteudo > novoNo->conteudo)
                pai->esq = novoNo;
            else
                pai->dir = novoNo;
        }
    }
    else {
        printf("\nChave %d ja presente na arvore!\n", valor);
    }
    return r;
}
```

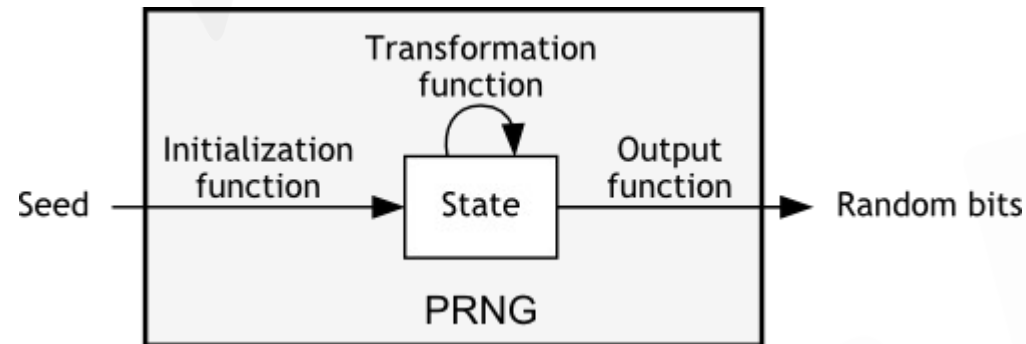
Teste02.c (solução)

```
$ gcc teste02.c -o teste02.exe  
$ ./teste02.exe 40000 < vetor4.dat
```

```
Altura da arvore:37
```

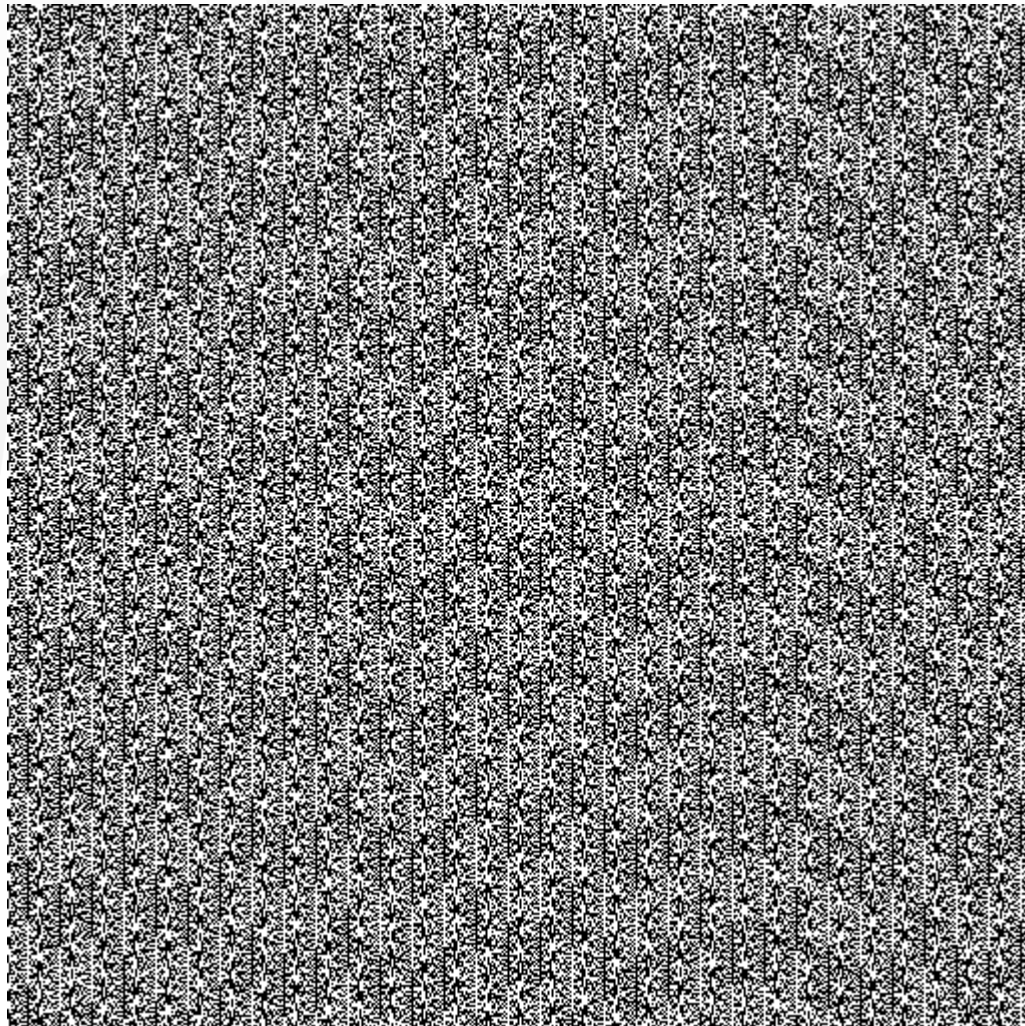
Vetor4.dat (contem números aleatórios)

1	1999865247
2	1642492079
3	1578754864
4	526388116
5	1202816126
6	1081054700
7	1412170679
8	349667423
9	908334402
10	182300537
11	369035264
12	632798115
13	634971847
14	1564533593
15	575255984
16	2043958983
17	149010144
18	311799603
19	512011104
20	321680977
21	1266852553
22	471588485
23	202818486
24	764059859
25	1963523066
26	643592132
27	1491412607
28	589261762
29	1516859391
30	1581697502
31	1600792033
32	1369240991
33	1076705934
34	1032063249
35	1805620107

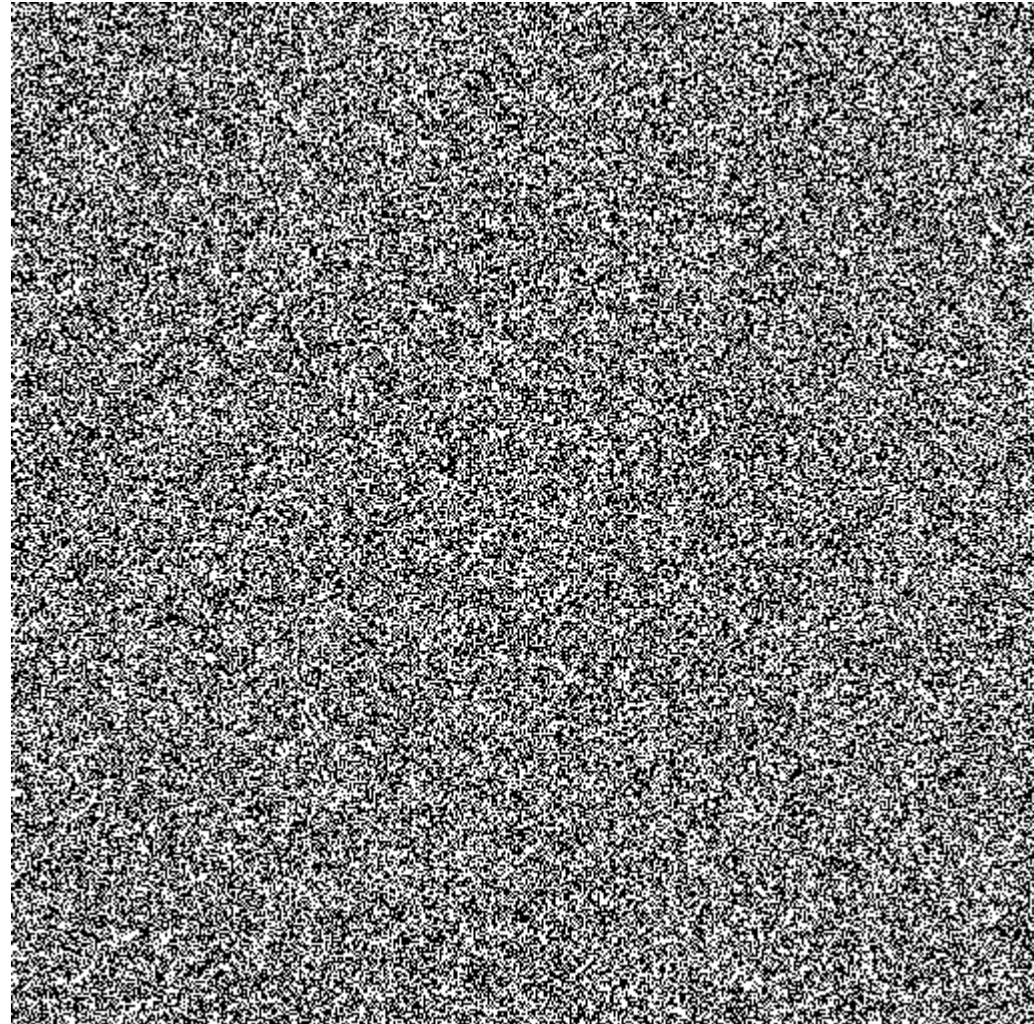


Vetor4.dat (contem números aleatórios)

Pseudo-random

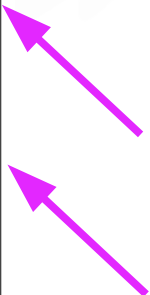


True-random



Teste02.c (solução)

n	lg(n)
2	1
32	5
512	9
8192	13
131072	17
2097152	21
33554432	25
536870912	29
8589934592	33
137438953472	37
2199023255552	41
35184372088832	45
562949953421312	49
9007199254740990	53
144115188075856000	57
2305843009213690000	61
36893488147419100000	65
5.9029581035871E+020	69
9.4447329657393E+021	73
1.5111572745183E+023	77
2.4178516392293E+024	81
3.8685626227668E+025	85
6.1897001964269E+026	89
9.9035203142831E+027	93
1.5845632502853E+029	97
2.5353012004565E+030	101
4.0564819207303E+031	105
6.4903710731685E+032	109
1.0384593717070E+034	113
8.3076749736557E+034	116



```
$ gcc teste02.c -o teste02.exe  
$ ./teste02.exe 40000 < vetor4.dat
```

Altura da arvore:37