

ÁRVORES E ÁRVORE BINÁRIA DE BUSCA

Prof. André Backes

Definição

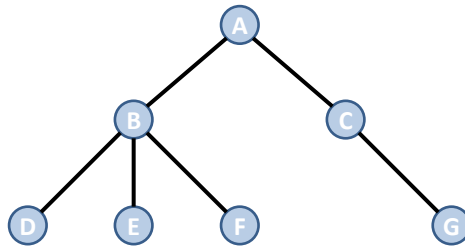
2

- Diversas aplicações necessitam que se represente um conjunto de objetos e as suas relações hierárquicas
- Uma árvore é uma abstração matemática usada para representar estruturas hierárquicas não lineares dos objetos modelados

Definição

3

- É um tipo especial de grafo
 - ▣ Definida usando um conjunto de nós (ou vértices) e arestas
 - ▣ Qualquer par de vértices está conectado a apenas uma aresta
 - Grafo não direcionado, conexo e acíclico (sem ciclos)



Definição

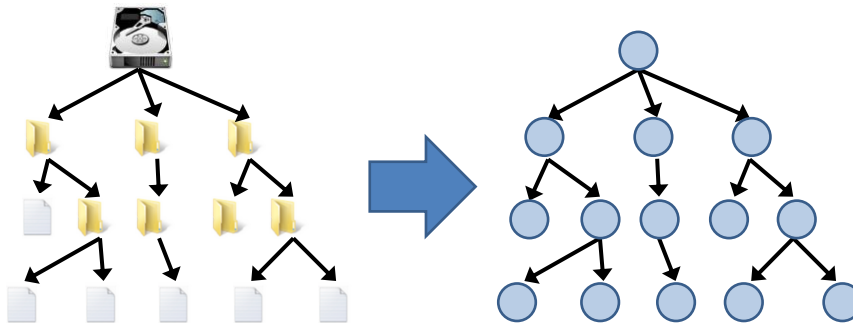
4

- Vértice
 - ▣ Cada uma das entidades representadas na árvore (depende da natureza do problema).
 - ▣ Basicamente, qualquer problema em que exista algum tipo de hierarquia pode ser representado por uma árvore

Definição

5

- Exemplo
 - ▣ Estrutura de pastas do computador



Definição

6

- Exemplos
 - ▣ relações de descendência (pai, filho, etc.)
 - ▣ diagrama hierárquico de uma organização;
 - ▣ campeonatos de modalidades desportivas;
 - ▣ taxonomia
- Em computação
 - ▣ busca de dados armazenados no computador
 - ▣ representação de espaço de soluções
 - Exemplo: jogo de xadrez;
 - ▣ modelagem de algoritmos

Conceitos básicos

7

- Principais conceitos relativos as árvores
 - ▣ Raiz
 - nó mais alto na árvore, o único que não possui pai
 - ▣ Pai ou ancestral
 - nó antecessor imediato de outro nó
 - ▣ Filho
 - é o nó sucessor imediato de outro nó

Conceitos básicos

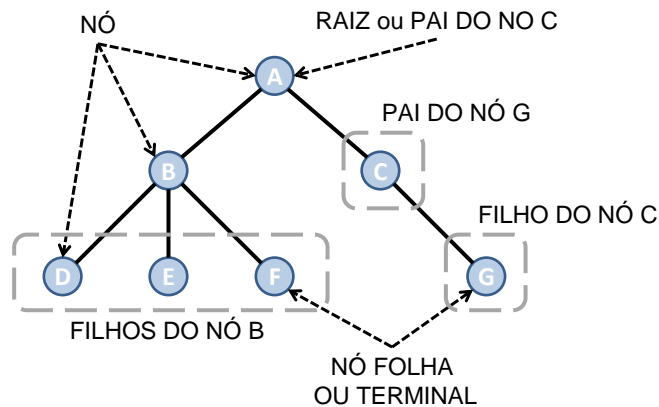
8

- Principais conceitos relativos as árvores
 - ▣ Nó folha ou terminal
 - qualquer nó que não possui filhos
 - ▣ Nó interno ou não-terminal
 - nó que possui ao menos UM filho
 - ▣ Caminho
 - sequência de nós de modo que existe sempre uma aresta ligando o nó anterior com o seguinte

Conceitos básicos

9

Exemplo



Conceitos básicos

10

Observação

- ▣ Dado um determinado nó da árvore, cada filho seu é considerado a **raiz** de uma nova **sub-árvore**
 - Qualquer nó é a **raiz** de uma **sub-árvore** consistindo dele e dos nós abaixo dele
 - Conceito recursivo

Conceitos básicos

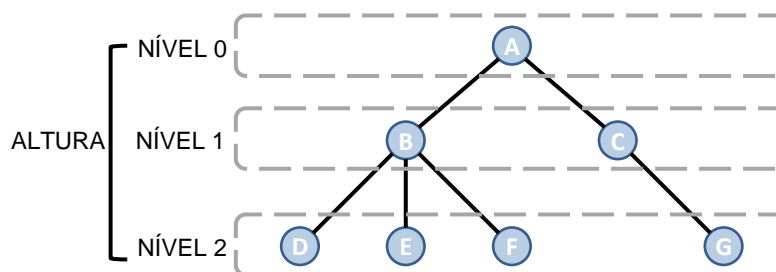
11

- Principais conceitos relativos as árvores
 - ▣ Nível
 - É dado pelo o número de nós que existem no caminho entre esse nó e a raiz (nível 0)
 - Nós são classificados em diferentes níveis
 - ▣ Altura
 - Também chamada de profundidade
 - Número total de níveis de uma árvore
 - Comprimento do caminho mais longo da raiz até uma das suas folhas

Conceitos básicos

12

- Nível e altura



Tipos de árvores

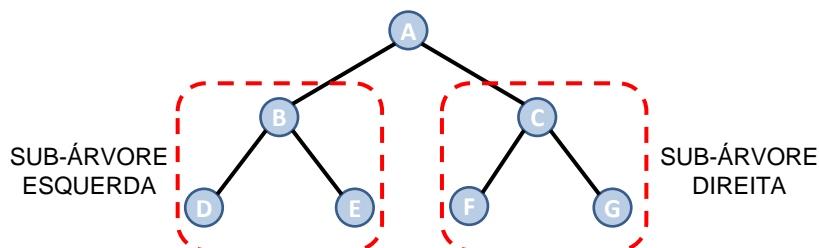
13

- Na computação, assim como na natureza, existem vários tipos diferentes de árvores.
 - ▣ Cada uma delas foi desenvolvida pensando diferentes tipos de aplicações
 - árvore binária de busca
 - árvore AVL
 - árvore Rubro-Negra
 - árvore B, B+ e B*
 - árvore 2-3
 - árvore 2-3-4
 - quadtree
 - octree

Árvore Binária

14

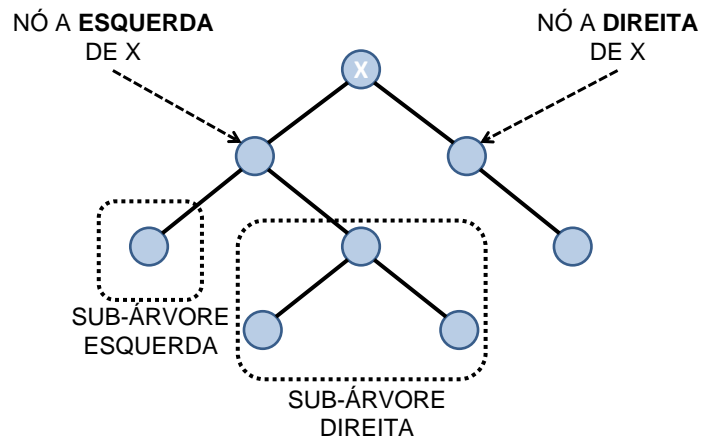
- É um tipo especial de árvore
 - ▣ Cada nó pode possuir nenhuma, uma ou no máximo duas **sub-árvores**
 - Sub-árvore da **esquerda** e a da **direita**
 - ▣ Usadas em situações onde, a cada passo, é preciso tomar uma decisão entre duas direções



Árvore Binária

15

- Exemplo de árvore binária



Árvore Binária

16

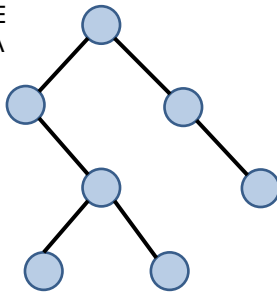
- Existem três tipos de árvores binárias
 - ▣ Estritamente binária
 - ▣ Completa
 - ▣ Cheia

Árvore Binária

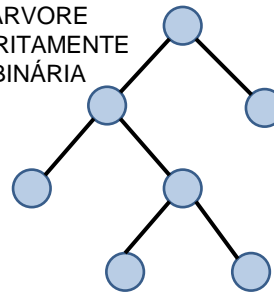
17

- Árvore estritamente binária
 - ▣ Cada nó possui sempre ou 0 (no caso de nó folha) ou 2 sub-árvores
 - ▣ Nenhum nó tem **filho único**

ÁRVORE
BINÁRIA



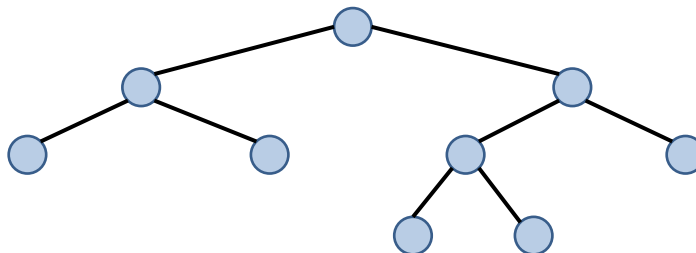
ÁRVORE
ESTRITAMENTE
BINÁRIA



Árvore Binária

18

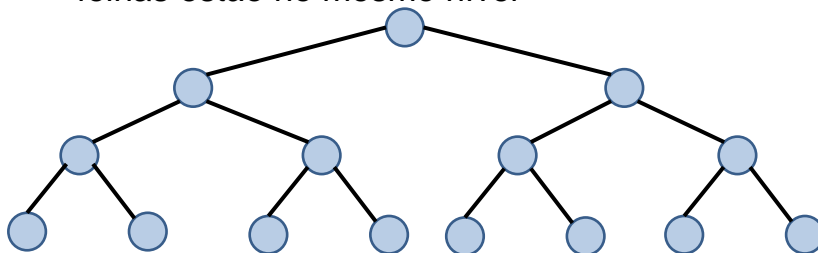
- Árvore binária completa
 - ▣ A diferença de altura entre as sub-árvores de qualquer nó é no máximo 1
 - ▣ Se a altura da árvore é **D**, cada nó folha está no nível **D** ou **D-1**



Árvore Binária

19

- Árvore binária cheia
 - ▣ Árvore estritamente binária onde todos os nós folhas estão no mesmo nível



Nível (n)	0	1	2	3
Nº de nós	1	2	4	8
Potência	2^0	2^1	2^2	2^3

Tipo de representação

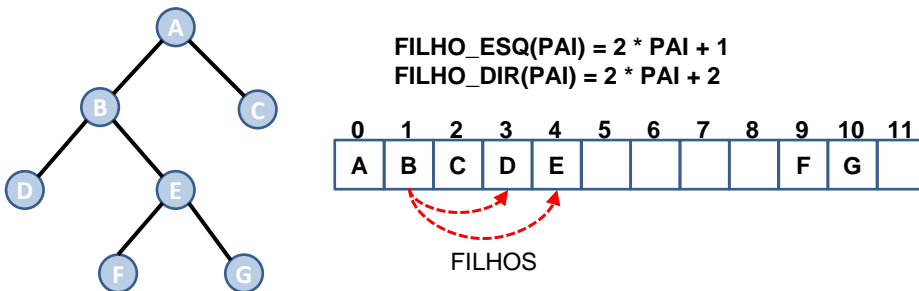
20

- Como implementar uma árvore no computador?
- Existem duas abordagens muito utilizadas
 - ▣ Usando um array (alocação estática)
 - ▣ Usando uma lista encadeada (alocação dinâmica)

Tipo de representação

21

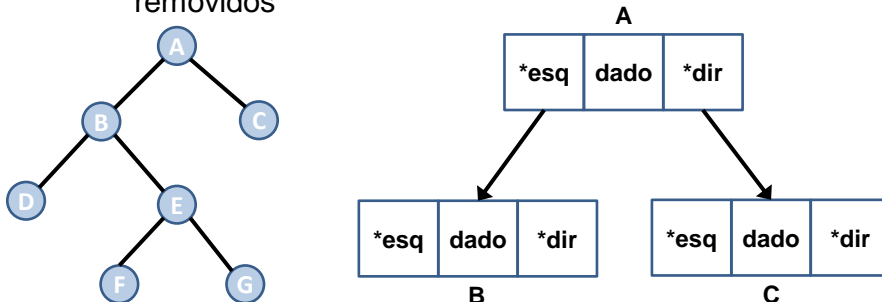
- Usando um array (alocação estática)
 - ▣ Necessário definir o número máximo de nós
 - Tamanho do array
 - ▣ Usa 2 funções para retornar a posição dos filhos à esquerda e à direita de um pai



Tipo de representação

22

- Lista encadeada (alocação dinâmica)
 - ▣ Espaço de memória alocado em tempo de execução
 - A árvore cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos



TAD Árvore Binária

23

□ Definição

▣ Uso de alocação dinâmica

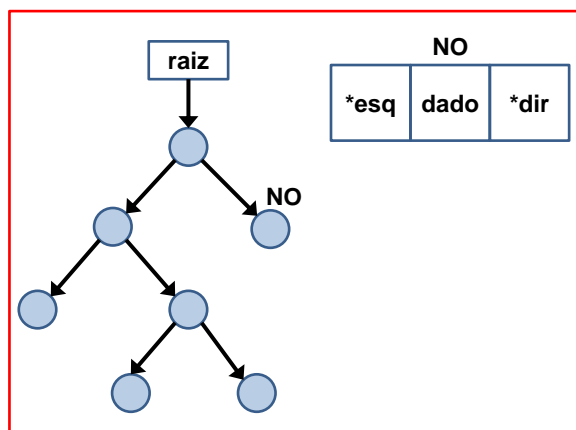
- Para guardar o primeiro nó da árvore utilizamos um **ponteiro para ponteiro**
- Um **ponteiro para ponteiro** pode guardar o endereço de um **ponteiro**
- Assim, fica fácil mudar quem é a **raiz** da árvore (se necessário)

TAD Árvore Binária

24

□ Definição

`ArvBin* raiz`



TAD Árvore Binária

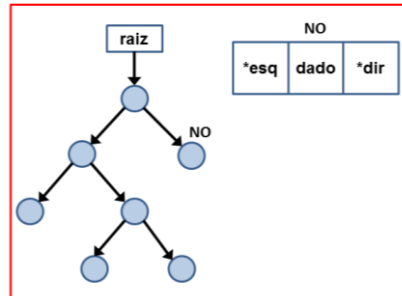
25

Definição

```
//Arquivo ArvoreBinaria.h
typedef struct NO* ArvBin;

//Arquivo ArvoreBinaria.c
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreBinaria.h" //inclui os Protótipos
struct NO{
    int info;
    struct NO *esq;
    struct NO *dir;
};
//programa principal
ArvBin* raiz; //ponteiro para ponteiro
```

ArvBin* raiz



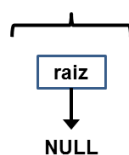
TAD Árvore Binária

26

Criando a árvore

```
1 //arquivo ArvoreBinaria.h
2 ArvBin* cria_ArvBin();
3
4 //arquivo ArvoreBinaria.c
5 ArvBin* cria_ArvBin(){
6     ArvBin* raiz = (ArvBin*) malloc(sizeof(ArvBin));
7     if(raiz != NULL)
8         *raiz = NULL;
9     return raiz;
10 }
11
12 //programa principal
13 ArvBin* raiz = cria_ArvBin();
```

ArvBin* raiz



TAD Árvore Binária

27

- Liberando a árvore
 - ▣ Uso de 2 funções: uma percorre e libera os nós, outra trata a raiz

```

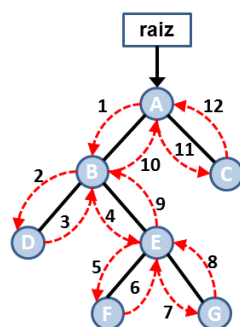
5 void libera_NO(struct NO* no) {
6     if(no == NULL)
7         return;
8     libera_NO(no->esq);
9     libera_NO(no->dir);
10    free(no);
11    no = NULL;
12 }
13 void libera_ArvBin(ArvBin* raiz) {
14     if(raiz == NULL)
15         return;
16     libera_NO(*raiz); //libera cada nó
17     free(raiz); //libera a raiz
18 }

```

TAD Árvore Binária

28

- Remoção: passo a passo



1	visita B
2	visita D
✗ 3	libera D, volta para B
4	visita E
5	visita F
✗ 6	libera F, volta para E
7	visita G
✗ 8	libera G, volta para E
✗ 9	libera E, volta para B
✗ 10	libera B, volta para A
11	visita C
✗ 12	libera C, volta para A e
✗	libera A
✗	libera raiz

TAD Árvore Binária

29

□ Informações básicas sobre a árvore

▣ Altura

■ Número total de níveis de uma árvore

```

5 int altura_ArvBin(ArvBin *raiz){
6     if (raiz == NULL)
7         return 0;
8     if (*raiz == NULL)
9         return 0;
10    int alt_esq = altura_ArvBin(&((*raiz)->esq));
11    int alt_dir = altura_ArvBin(&((*raiz)->dir));
12    if (alt_esq > alt_dir)
13        return (alt_esq + 1);
14    else
15        return(alt_dir + 1);
16 }

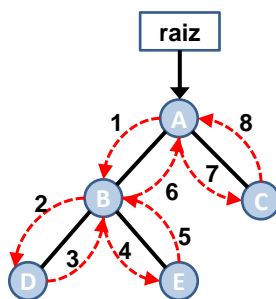
```

TAD Árvore Binária

30

□ Informações básicas sobre a árvore

▣ Altura



	inicia no nó A
1	visita B
2	visita D
3	D é nó folha: altura é 1. Volta para B
4	visita E
5	E é nó folha: altura é 1. Volta para B
6	altura de B é 2: maior altura dos filhos + 1. Volta para A
7	visita C
8	C é nó folha: altura é 1. Volta para A
	altura de A é 3: maior altura dos filhos + 1.

TAD Árvore Binária

- Informações básicas sobre a árvore
 - Número de nós
 - Quantidade de elementos na árvore

```

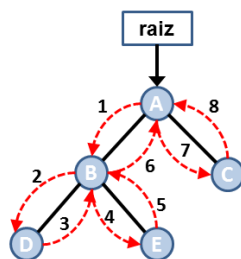
5 int totalNO_ArvBin(ArvBin *raiz){
6     if (raiz == NULL)
7         return 0;
8     if (*raiz == NULL)
9         return 0;
10    int alt_esq = totalNO_ArvBin(&((*raiz)->esq));
11    int alt_dir = totalNO_ArvBin(&((*raiz)->dir));
12    return(alt_esq + alt_dir + 1);
13 }

```

TAD Árvore Binária

32

- Informações básicas sobre a árvore
 - Número de nós



1	visita B
2	visita D
3	D é nó folha: conta como 1 nó. Volta para B
4	visita E
5	E é nó folha: conta como 1 nó. Volta para B
6	Número de nós em B é 3: total de nós a direita (1) + total de nós a esquerda (1) + 1. Volta para A
7	visita C
8	C é nó folha: conta como 1 nó. Volta para A
	Número de nós em A é 5: total de nós a direita (1) + total de nós a esquerda (3) + 1.

Percurso na árvore

33

- Percorrer todos os nós é uma operação muito comum em árvores binárias
 - ▣ Cada nó é visitado uma única vez
 - Isso gera uma sequência linear de nós, cuja ordem depende de como a árvore foi percorrida
 - ▣ Não existe uma **ordem natural** para se percorrer todos os nós de uma árvore binária
 - ▣ Isso pode ser feito para executar alguma ação em cada nó
 - Essa ação pode ser mostrar (imprimir) o valor do nó, modificar esse valor, etc.

Percurso na árvore

34

- Podemos percorrer a árvore de 3 formas
 - ▣ Percurso pré-ordem
 - visita a **raiz**, o filho da **esquerda** e o filho da **direita**
 - ▣ Percurso um-ordem
 - visita o filho da **esquerda**, a **raiz** e o filho da **direita**
 - ▣ Percurso pós-ordem
 - visita o filho da **esquerda**, o filho da **direita** e a **raiz**
- Essas são os percursos mais importantes
 - ▣ Existem outras formas de percurso

Percurso pré-ordem

35

□ Ordem de visitação

- ▣ Raiz
- ▣ Filho esquerdo
- ▣ Filho direito

```

5 void preOrdem_ArvBin(ArvBin *raiz){
6     if(raiz == NULL)
7         return;
8     if(*raiz != NULL){
9         printf("%d\n", (*raiz)->info);
10        preOrdem_ArvBin(&((*raiz)->esq));
11        preOrdem_ArvBin(&((*raiz)->dir));
12    }
13 }

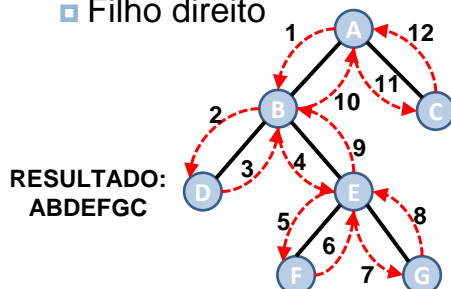
```

Percurso pré-ordem

36

□ Ordem de visitação

- ▣ Raiz
- ▣ Filho esquerdo
- ▣ Filho direito



	inicia no nó A
1	imprime A, visita B
2	imprime B, visita D
3	imprime D, volta para B
4	visita E
5	imprime E, visita F
6	imprime F, volta para E
7	visita G
8	imprime G, volta para E
9	volta para B
10	volta para A
11	visita C
12	imprime C, volta para A

Percurso em-ordem

37

- Ordem de visitação
 - ▣ Filho esquerdo
 - ▣ Raiz
 - ▣ Filho direito

```

5 void emOrdem_ArvBin(ArvBin *raiz) {
6     if(raiz == NULL)
7         return;
8     if(*raiz != NULL) {
9         emOrdem_ArvBin(&((*raiz)->esq));
10        printf("%d\n", (*raiz)->info);
11        emOrdem_ArvBin(&((*raiz)->dir));
12    }
13 }

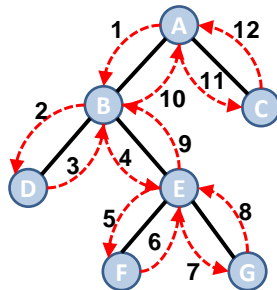
```

Percurso em-ordem

38

- Ordem de visitação
 - ▣ Filho esquerdo
 - ▣ Raiz
 - ▣ Filho direito

RESULTADO:
DBFEGAC



inicia no nó A

- | | |
|----|-------------------------|
| 1 | visita B |
| 2 | visita D |
| 3 | imprime D, volta para B |
| 4 | imprime B, visita E |
| 5 | visita F |
| 6 | imprime F, volta para E |
| 7 | imprime E, visita G |
| 8 | imprime G, volta para E |
| 9 | volta para B |
| 10 | volta para A |
| 11 | imprime A, visita C |
| 12 | imprime C, volta para A |

Percurso pós-ordem

39

- Ordem de visitação
 - ▣ Filho esquerdo
 - ▣ Filho direito
 - ▣ Raiz

```

5 void posOrdem_ArvBin(ArvBin *raiz) {
6     if(raiz == NULL)
7         return;
8     if(*raiz != NULL) {
9         posOrdem_ArvBin(&((*raiz)->esq));
10        posOrdem_ArvBin(&((*raiz)->dir));
11        printf("%d\n", (*raiz)->info);
12    }
13 }

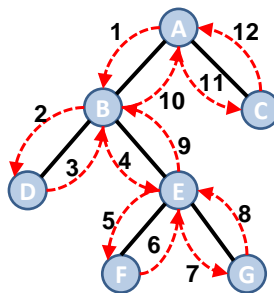
```

Percurso pós-ordem

40

- Ordem de visitação
 - ▣ Filho esquerdo
 - ▣ Filho direito
 - ▣ Raiz

RESULTADO:
DFGEBCA



	inicia no nó A
1	visita B
2	visita D
3	imprime D, volta para B
4	visita E
5	visita F
6	imprime F, volta para E
7	visita G
8	imprime G, volta para E
9	imprime E, volta para B
10	imprime B, volta para A
11	visita C
12	imprime C, volta para A e imprime A

41

Árvore Binária de Busca - ABB

Árvore Binária de Busca

42

- Definição
 - ▣ É uma árvore binária
 - Cada nó pode ter 0, 1 ou 2 filhos
 - ▣ Cada nó possui da árvore possui um valor (chave) associado a ele
 - Não existem valores repetidos
 - ▣ Esse valor determina a posição do nó na árvore

Árvore Binária de Busca

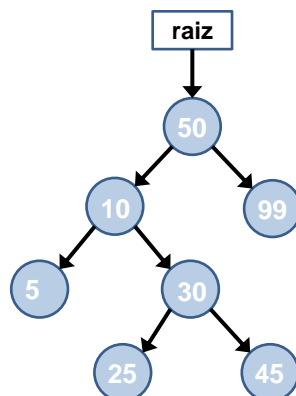
43

- Regra para posicionamento dos valores na árvore
 - ▣ Para cada nó pai
 - todos os valores da sub-árvore **esquerda são menores** do que o nó pai
 - todos os valores da sub-árvore **direita são maiores** do que o nó pai;
 - ▣ Inserção e remoção devem ser realizadas respeitando essa regra de posicionamento dos nós.

Árvore Binária de Busca

44

- Exemplo



Árvore Binária de Busca

45

- Ótima alternativa para operações de busca binária
 - ▣ Possui a vantagem de ser uma estrutura dinâmica em comparação ao array
 - ▣ É mais fácil inserir valores na árvore do que em um array ordenado
 - Array: envolve deslocamento de elementos

Árvore Binária de Busca

46

- Custo para as principais operações em uma **árvore binária de busca** contendo **N** nós.
 - ▣ O pior caso ocorre quando a árvore não está balanceada

	Melhor Caso	Pior Caso
Inserção	$O(\log N)$	$O(N)$
Remoção	$O(\log N)$	$O(N)$
Busca	$O(\log N)$	$O(N)$

Árvore Binária de Busca - Inserção

47

- Para inserir um valor **V** na árvore
 - ▣ Se a raiz é igual a **NULL**, insira o nó
 - ▣ Se **V** é menor do que a raiz: vá para a **sub-árvore esquerda**
 - ▣ Se **V** é maior do que a raiz: vá para a **sub-árvore direita**
 - ▣ Aplique o método **recursivamente**
 - pode ser feito sem recursão
- Dessa forma, percorremos um conjunto de nós da árvore até chegar ao nó folha que irá se tornar o pai do novo nó

Árvore Binária de Busca: Inserção

48

- Devemos também considerar a inserção em uma árvore que está vazia

Inserir o valor '40'
em uma árvore
vazia



```
*raiz = novo;
```


TAD Árvore Binária

49

□ Inserção em uma Árvore Binária de Busca

```
int insere_ArvBin(ArvBin* raiz, int valor){
    if(raiz == NULL)
        return 0;
    struct NO* novo;
    novo = (struct NO*) malloc(sizeof(struct NO));
    if(novo == NULL)
        return 0;
    novo->info = valor;
    novo->dir = NULL;
    novo->esq = NULL;
    //procurar onde inserir!
```

TAD Árvore Binária

50

□ Inserção em uma Árvore Binária de Busca

Navega nos
nós da árvore
até chegar em
um nó folha

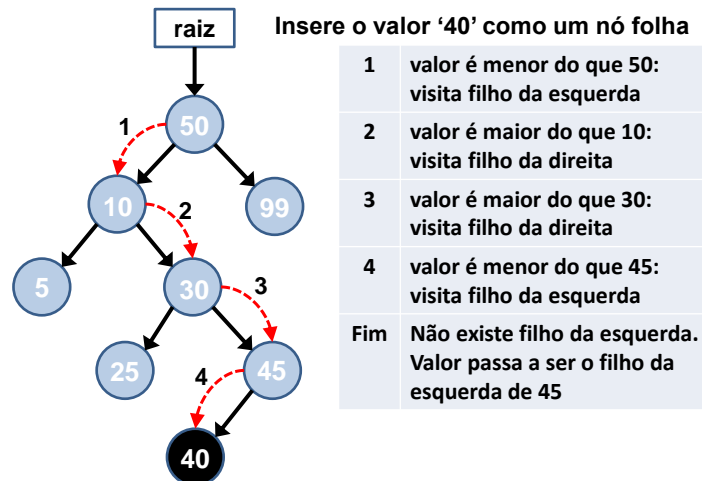
Insere como
filho desse nó
folha

```
if(*raiz == NULL)
    *raiz = novo;
else{
    struct NO* atual = *raiz;
    struct NO* ant = NULL;
    while(atual != NULL){
        ant = atual;
        if(valor == atual->info){
            free(novo);
            return 0; //elemento já existe
        }
        if(valor > atual->info)
            atual = atual->dir;
        else
            atual = atual->esq;
    }
    if(valor > ant->info)
        ant->dir = novo;
    else
        ant->esq = novo;
    return 1;
}
```

Árvore Binária de Busca: Inserção

51

Exemplo



Árvore Binária de Busca: Busca

52

- Consultar se um determinado nó **V** existe em uma árvore é similar a operação de inserção
 - ▣ primeiro compare o valor buscado com a **raiz**;
 - ▣ se **V** é menor do que a raiz: vá para a **sub-árvore da esquerda**;
 - ▣ se **V** é maior do que a raiz: vá para a **sub-árvore da direita**;
 - ▣ aplique o método recursivamente até que a raiz seja igual ao valor buscado
 - pode ser feito sem recursão

TAD Árvore Binária

53

□ Busca em uma Árvore Binária de Busca

```

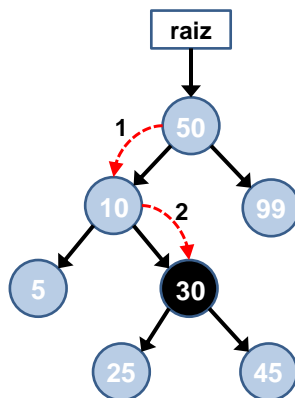
6 int consulta_ArvBin(ArvBin *raiz, int valor) {
7     if(raiz == NULL)
8         return 0;
9     struct NO* atual = *raiz;
10    while(atual != NULL) {
11        if(valor == atual->info) {
12            return 1;
13        }
14        if(valor > atual->info)
15            atual = atual->dir;
16        else
17            atual = atual->esq;
18    }
19    return 0;
20 }

```

Árvore Binária de Busca: Busca

54

□ Exemplo



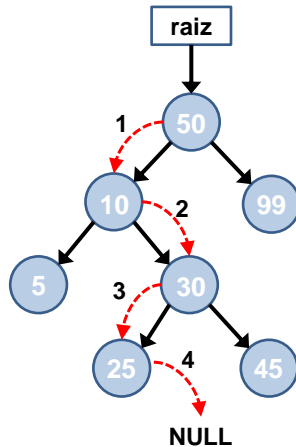
Valor procurado: 30

1	valor procurado é menor do que 50: visita filho da esquerda
2	valor procurado é maior do que 10: visita filho da direita
Fim	valor procurado é igual ao do nó: retornar dados do nó

Árvore Binária de Busca: Busca

55

- Exemplo: valor buscado não existe!



Valor procurado: 28

1	valor procurado é menor do que 50: visita filho da esquerda
2	valor procurado é maior do que 10: visita filho da direita
3	valor procurado é menor do que 30: visita filho da esquerda
4	valor procurado é maior do que 25: visita filho da direita
Fim	Filho da direita de 25 não existe: a busca falhou

Árvore Binária de Busca: Remoção

56

- Remover um nó de uma árvore binária de busca não é uma tarefa tão simples quanto a inserção.
 - Isso ocorre porque precisamos procurar o nó a ser removido da árvore o qual pode ser um
 - nó folha
 - nó interno (que pode ser a raiz), com um ou dois filhos.
 - Se for um nó interno
 - Reorganizar a árvore para que ela continue sendo uma árvore binária de busca

TAD Árvore Binária

57

- Remoção em uma Árvore Binária de Busca
 - ▣ Trabalha com 2 funções
 - Busca pelo nó
 - Tratar os 3 tipos de remoção: com 0, 1 ou 2 filhos

```

8  int remove_ArvBin(ArvBin *raiz, int valor){
9      /*
10     FUNÇÃO RESPONSÁVEL PELA BUSCA
11     DO NÓ A SER REMOVIDO
12     */
13 }
14 struct NO* remove_atual(struct NO* atual){
15     /*
16     FUNÇÃO RESPONSÁVEL POR TRATAR OS 3
17     TIPOS DE REMOÇÃO
18     */
19 }

```

TAD Árvore Binária

58

- Remoção em uma Árvore Binária de Busca

Achou o nó a ser removido. Tratar o tipo de remoção

Continua andando na árvore a procura do nó a ser removido

```

int remove_ArvBin(ArvBin *raiz, int valor){
    if(raiz == NULL) return 0;
    struct NO* ant = NULL;
    struct NO* atual = *raiz;
    while(atual != NULL){
        if(valor == atual->info){
            if(atual == *raiz)
                *raiz = remove_atual(atual);
            else{
                if(ant->dir == atual)
                    ant->dir = remove_atual(atual);
                else
                    ant->esq = remove_atual(atual);
            }
            return 1;
        }
        ant = atual;
        if(valor > atual->info)
            atual = atual->dir;
        else
            atual = atual->esq;
    }
    return 0;
}

```

TAD Árvore Binária

59

Remoção em uma Árvore Binária de Busca

```

1 struct NO* remove_atual(struct NO* atual) {
2     struct NO *no1, *no2;
3     if(atual->esq == NULL){
4         no2 = atual->dir;
5         free(atual);
6         return no2;
7     }
8     no1 = atual;
9     no2 = atual->esq;
10    while(no2->dir != NULL){
11        no1 = no2;
12        no2 = no2->dir;
13    }
14
15    if(no1 != atual){
16        no1->dir = no2->esq;
17        no2->esq = atual->esq;
18    }
19    no2->dir = atual->dir;
20    free(atual);
21    return no2;
22 }

```

Sem filho da esquerda.
Apontar para o filho da direita (trata nó folha e nó com 1 filho)

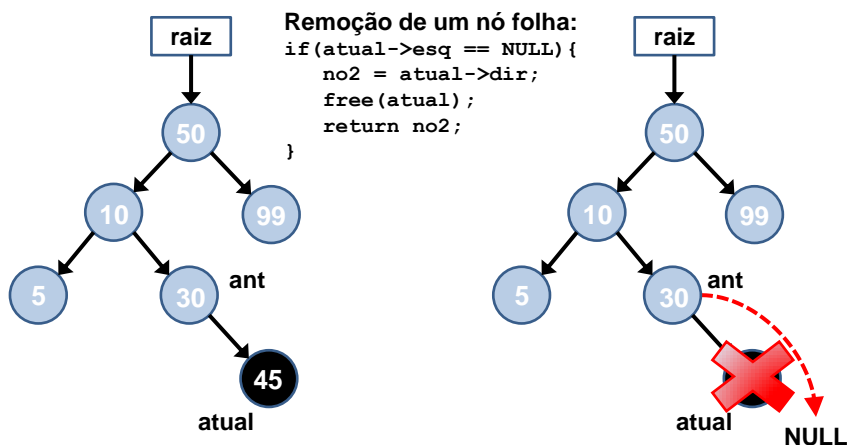
Procura filho mais a direita na sub-árvore da esquerda.

Copia o filho mais a direita na sub-árvore da esquerda para o lugar do nó removido.

Árvore Binária de Busca: Remoção

60

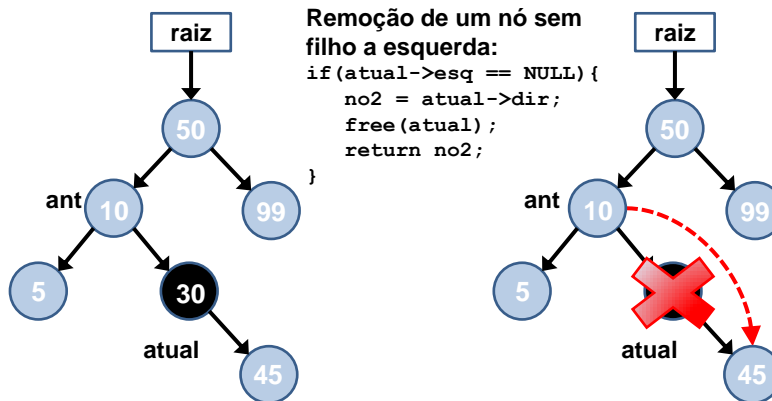
Exemplo: remoção de um nó folha



Árvore Binária de Busca: Remoção

61

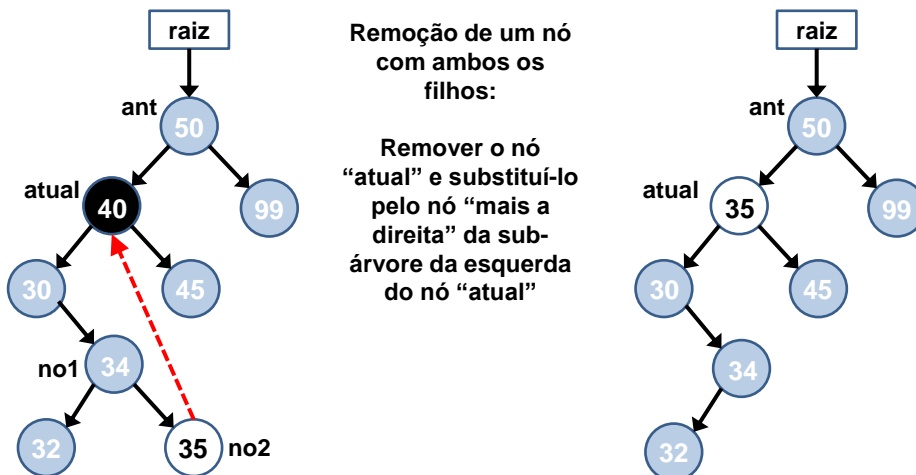
- Exemplo: remoção de um nó com 1 filho



Árvore Binária de Busca: Remoção

62

- Exemplo: remoção de um nó com 2 filhos



Material Complementar

63

- Vídeo Aulas
 - ▣ Aula 67: Árvores:
 - ▣ youtu.be/iLvpaqAoVD8
 - ▣ Aula 68: Árvores: propriedades:
 - ▣ youtu.be/U7liLJIMfnU
 - ▣ Aula 69: Árvore Binária: Definição:
 - ▣ youtu.be/9WxCeWX9qDs
 - ▣ Aula 70: Árvore Binária: Implementação:
 - ▣ youtu.be/TR8ZLUKmcPc
 - ▣ Aula 71: Criando e destruindo uma árvore binária:
 - ▣ youtu.be/QAJkoJW8bEc
 - ▣ Aula 72: Árvore Binária: informações básicas:
 - ▣ youtu.be/qVnNdmx4fOA

Material Complementar

64

- Vídeo Aulas
 - ▣ Aula 73: Percorrendo uma Árvore Binária:
 - ▣ youtu.be/z7XwVVYQRAA
 - ▣ Aula 74: Árvore Binária de Busca:
 - ▣ youtu.be/M7cb4HjePjk
 - ▣ Aula 75: Inserção em Árvore Binária de Busca:
 - ▣ youtu.be/8cdbmsPaR-k
 - ▣ Aula 76: Remoção em Árvore Binária de Busca:
 - ▣ youtu.be/_0Yu9BSYXGY
 - ▣ Aula 77: Consulta em Árvore Binária de Busca:
 - ▣ youtu.be/mw_wqqB48yY