

SIN 211 - Algoritmos e Estruturas de Dados

(Árvores Binárias de Pesquisa)

Prof^o: Joelson Antônio dos Santos

Universidade Federal de Viçosa
Instituto de Ciências Exatas e Tecnológicas
Campus de Rio Paranaíba - MG

joelsonn.santos@gmail.com
Sala: BBT 233

14 de junho de 2018

Aula de Hoje

1 Árvores Binárias de Pesquisa

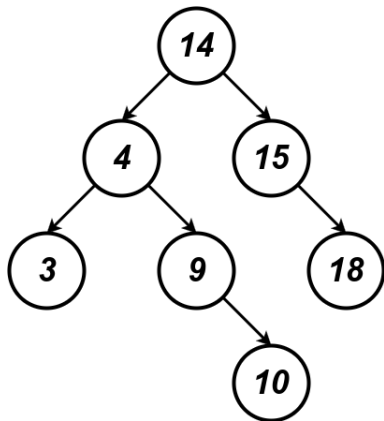
- Propriedades
- Operações

Árvores Binárias de Pesquisa

- Tipo especial de árvore binária, em que cada nó possui um **valor** (chave) associado a ele, e esse valor determina a “posição” desse nó na árvore;
- Em uma árvore, é necessário percorrê-la até que o elemento seja encontrado, caso ele exista;

Árvores Binárias de Pesquisa

- Maneira de organizar elementos na árvore:
 - Todos os itens da sub-árvore **esquerda** são menores do que a raiz;
 - Todos os itens da sub-árvore **direita** são maiores ou iguais a raiz;
 - Cada sub-árvore é também uma árvore binária de pesquisa;



Árvores Binárias de Pesquisa

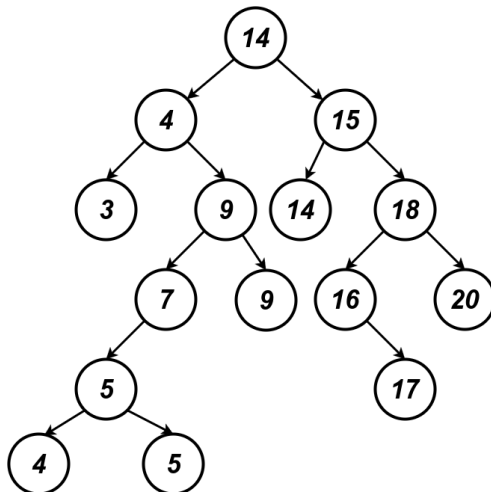
- O operador \leq ou $<$ deve ser sobrecarregado caso seja necessário processar alguma chave não numérica ou de conteúdo diferente;
- A ordem alfabética é geralmente usada quando em chaves de tipo “*string*”;
- E se a informação do nó for uma **struct**?

Árvores Binárias de Pesquisa

- Obedecendo as regras para a construção de uma árvore binária de busca/pesquisa, como ficaria a árvore após a inserção dos seguintes elementos?
 - {14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5}

Árvores Binárias de Pesquisa

- {14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5}



Árvores Binárias de Pesquisa

- Use os mesmos elementos do slide 6 em uma sequência diferente e construa novamente a árvore binária de busca;
- A construção de uma árvore binária de pesquisa é dependente da sequência de inserção dos dados. Então, um conjunto dos mesmos valores pode formar diferentes árvores;

Árvores Binárias de Pesquisa

- Qual o tipo de percurso pode ser utilizado para imprimir os elementos em ordem **crescente**?
- O que deve ser modificado no tipo de percurso escolhido no item anterior para a impressão em ordem **decrecente**?

Árvores Binárias de Pesquisa

- Principais operações de uma árvore binária de pesquisa:
 - Inicializar;
 - Inserir;
 - Pesquisar;
 - Remover;

Árvores Binárias de Pesquisa

- Definição de uma árvore binária de pesquisa:
 - Cada nó deve conter:
 - O campo da informação;
 - Um ponteiro para o nó à esquerda;
 - Um ponteiro para o nó à direita;
- Uma estrutura árvore contendo um ponteiro para a raiz da árvore pode ser definida;

Árvores Binárias de Pesquisa

- Estruturas para uma árvore binária de pesquisa;
- A primeira e a terceira estruturas são opcionais;
- Qual outro tipo de informação pode ser adicionada à estrutura **ARVBINPESQ**?

```
typedef struct sInfo{  
    <tipo> info;  
}ELEMENTO;  
  
typedef struct sNo{  
    ELEMENTO info;  
    struct sNo *esq;  
    struct sNo *dir;  
}NO;  
  
typedef struct sArv{  
    NO *raiz;  
}ARVBINPESQ;
```

Árvores Binárias de Pesquisa

- Para inicializar uma árvore, é preciso que o ponteiro externo aponte para NULL;
- Também é necessário criar a operação de verificação de árvore vazia; Se a árvore é vazia, então o ponteiro externo está apontando para NULL;

Árvores Binárias de Pesquisa

- **Inserção:**


- Para inserir um elemento na árvore binária de pesquisa, é necessário tratar os seguintes casos:
 - Se a árvore está vazia, insere o elemento como raiz;
 - Senão, insere o elemento na posição adequada (de acordo com a ordenação da árvore);

Árvores Binárias de Pesquisa

- **Inserção** - Quando a árvore não está vazia:
 - Comparar com a raiz:
 - Se o elemento é menor do que a raiz, temos que ir para a sub-árvore da esquerda;
 - Se o elemento é maior ou igual a raiz, temos que ir para sub-árvore da direita;
 - O método é aplicado recursivamente;
 - É possível desenvolver o código de maneira não recursiva;

Árvores Binárias de Pesquisa

- Inserção recursiva;



```
void inserir(NO **raiz, int elemento){
    if((*raiz) == NULL){
        (*raiz) = criarNo(); // célula
        (*raiz)->esq = NULL;
        (*raiz)->dir = NULL;
        (*raiz)->info = elemento;
    }else{
        if(elemento < (*raiz)->info){
            inserir(&(*raiz)->esq, elemento);
        }if(elemento > (*raiz)->info){
            inserir(&(*raiz)->dir, elemento);
        }
    }
}
```



Árvores Binárias de Pesquisa

- **Pesquisa:**

- Inicializa a busca por elementos a partir do nó raiz;
- Para cada nó raiz de uma sub árvore, comparar:
 - O elemento procurado é menor do que a raiz? Neste caso, continue a busca pela sub árvore da esquerda.
 - Caso contrário, realiza a busca pela sub árvore da direita.
- Se o elemento for encontrado, retorne o ponteiro para o nó com o elemento, senão, retorne NULL.

Árvores Binárias de Pesquisa

- Pesquisa:



```
NO* pesquisar(NO *raiz, int elemento){
    if(raiz == NULL){
        return NULL;
    }
    if(raiz->info == elemento){
        return raiz;
    }else if(raiz->info > elemento){
        return pesquisar(raiz->esq, elemento);
    }else{
        return pesquisar(raiz->dir, elemento);
    }
}
```

Árvores Binárias de Pesquisa

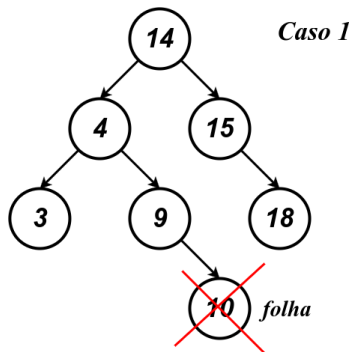
- **Remoção:**

- São considerados 3 casos distintos:
 - **Caso 1** - O nó é folha: Então, basta removê-lo sem modificação a estrutura da árvore;
 - **Caso 2** - O nó possui uma sub-árvore: Então, o nó da sub-árvore assume o lugar do nó removido;
 - **Caso 3** - O nó possui duas sub-árvores - Neste caso devemos escolher o nó que irá ocupar o nó removido, de forma a respeitar a estrutura da árvore binária de pesquisa;

Árvores Binárias de Pesquisa - Remoção

Caso 1

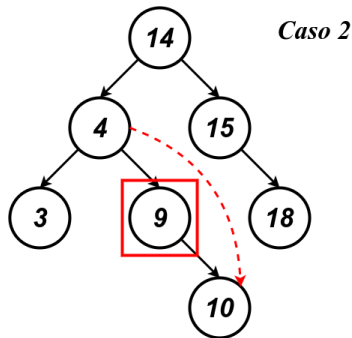
- Os nós 10, 3 e 18 podem ser removidos sem a necessidade de reajustes na árvore;
- “Quando o nó a ser removido é folha, o mesmo pode ser retirado da árvore sem outras alterações”



Árvores Binárias de Pesquisa - Remoção

Caso 2

- O elemento 9 possui uma sub-árvore direita, o nó contendo o valor 10 irá “ocupar” o lugar do nó removido;
- “O nó raiz da sub-árvore (esq. ou dir.) do nó removido ocupa o lugar do nó removido.”



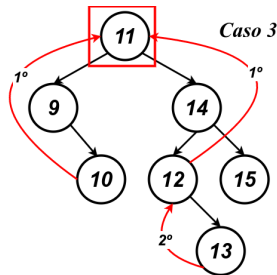
Árvores Binárias de Pesquisa - Remoção

Caso 3

- O elemento 11 possui duas sub-árvores. Então:

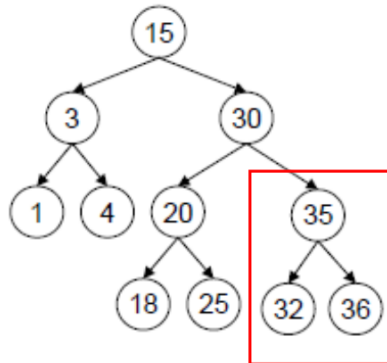
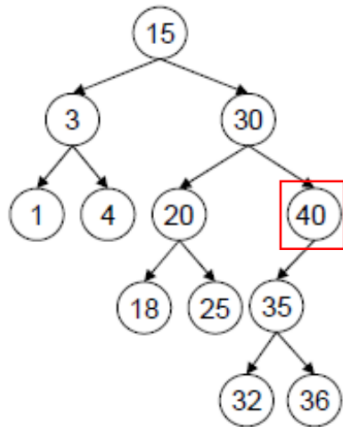
- 1 O nó com chave 10 pode “ocupar” o lugar do nó raiz;
- 2 **OU** o nó 12 pode “ocupar” o lugar do nó raiz e o 13 o lugar do 12;

- “O nó contendo o menor valor da sub árvore direita pode “ocupar” o lugar do nó removido; ou o maior valor da sub-árvore esquerda pode “ocupar” o lugar do nó removido.”



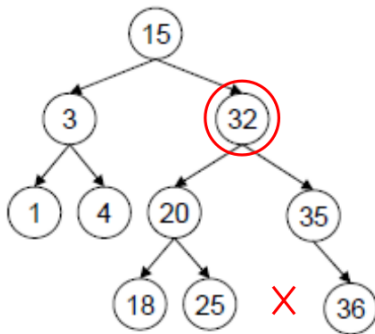
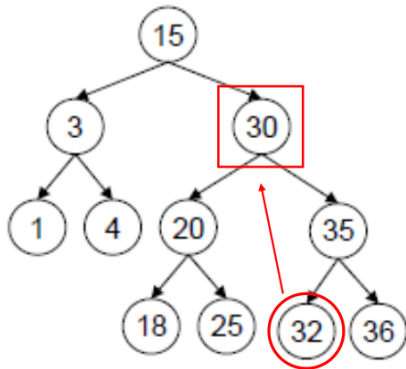
Árvores Binárias de Pesquisa - Exemplo

Remover o elemento 40



Árvores Binárias de Pesquisa - Exemplo

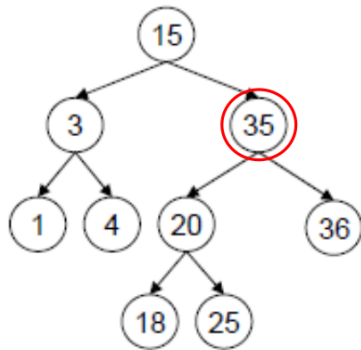
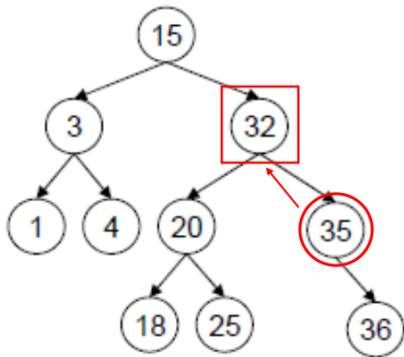
Remover o elemento 30



Qual seria a outra opção?

Árvores Binárias de Pesquisa - Exemplo

Remover o elemento 32



Qual seria a outra opção?

Árvores Binárias de Pesquisa - Remoção (Parte 1)


```
void remover(NO** raiz, int elemento){
    NO *aux;
    if(arvoreVazia(raiz)){
        printf("Arvore vazia");
        return;
    }
    // Continua na sub-árvore esquerda
    if(elemento < (*raiz)->info){
        remover(&(*raiz)->esq, elemento);
    }else{
        // Continua na sub-árvore direita
        if(elemento > (*raiz)->info){
            remover(&(*raiz)->dir, elemento);
        }else{// valor encontrado
            aux = (*raiz);
            // Folha - caso 1
            if((*raiz)->esq == NULL && ((*raiz)->dir == NULL)){
                free(aux);
                (*raiz) = NULL;
            }
        }
    }
}
```

Árvores Binárias de Pesquisa - Remoção (Parte 2)



```
else{ // caso 2 dir
    if((*raiz)->esq == NULL){
        (*raiz) = (*raiz)->dir;
        aux->dir = NULL;
        free(aux);
    }else{ // caso 2 eq
        if((*raiz)->dir == NULL){
            (*raiz) = (*raiz)->esq;
            aux->esq = NULL;
            free(aux);
        }else{ // caso 3
            aux = maiorElemento(&(*raiz)->esq);
            aux->esq = (*raiz)->esq;
            aux->dir = (*raiz)->dir;
            free((*raiz));
            (*raiz) = aux;
        }
    }
}
```

Árvores Binárias de Pesquisa - Remoção (maiorElemento)



```
NO* maiorElemento(NO** no){
    if((*no)->dir != NULL){
        return maiorElemento(&(*no)->dir);
    }else{
        NO *aux = (*no);
        if((*no)->esq != NULL){
            (*no) = (*no)->esq;
        }else{
            (*no) = NULL;
        }
        return aux;
    }
}
```

Árvores Binárias de Pesquisa

- É possível implementar a operação de remoção de maneira não recursiva; Também pode-se criar outras funções dependendo da aplicação.
 - Imprimir todos os nós;
 - Imprimir os folhas;
 - Imprimir número de ocorrências de um elemento, etc.

Árvores Binárias de Pesquisa

- Desenvolva um método não recursivo para inserir um elemento em uma árvore binária de pesquisa.
- Desenvolva um método não recursivo para remover um elemento em uma árvore binária de pesquisa.

Bibliografia Básica

- DROZDEK, Adam. Estrutura de Dados e Algoritmos em C++. Editora Pioneira Thomson Learning, 2005.
- <https://goo.gl/gk01D5> - Acessado em 7 de novembro de 2017.
- Estrutura de dados descomplicada em linguagem C, CAPÍTULO 11 - André Ricardo Backes, <https://www.evolution.com.br/epubreader/estrutura-de-dados-descomplicada-em-linguagem-c-1ed>