

Estruturas de Dados

Árvores

Departamento de Computação
Prof. Martín Vigil
Adaptado de prof. Jean Martina e Aldo Wangenheim

2020.1



UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Introdução

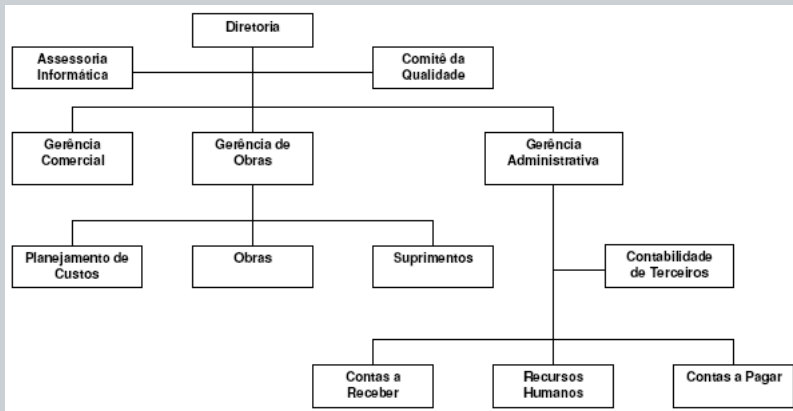
Árvores

são estruturas de dados que se caracterizam por uma organização hierárquica entre seus elementos. Essa organização permite a definição de algoritmos relativamente simples, recursivos e de eficiência bastante razoável.

Introdução

- ▶ No cotidiano, diversas informações são organizadas de forma hierárquica;
- ▶ Como exemplo, podem ser citados:
 - ▶ O organograma de uma empresa;
 - ▶ A divisão de um livro em capítulos, seções, tópicos;
 - ▶ A árvore genealógica de uma pessoa.

Introdução

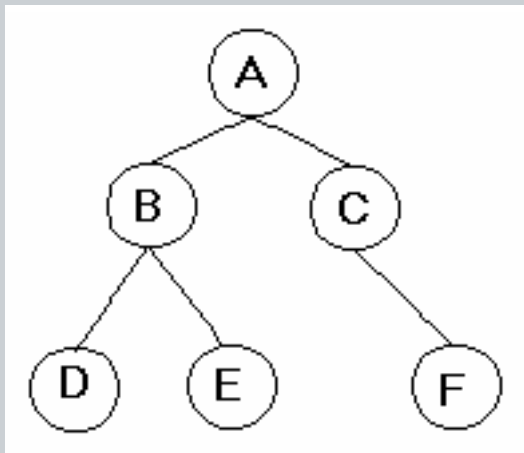


Introdução

- ▶ De um modo mais formal, podemos dizer que uma árvore é um conjunto finito de um ou mais nodos, nós ou vértices, tais que:
 - ▶ Existe um nodo denominado raiz da árvore;
 - ▶ os demais nodos formam $n \geq 0$ conjuntos disjuntos c_1, c_2, \dots, c_n , sendo que cada um desses conjuntos também é uma árvore (denominada subárvore).

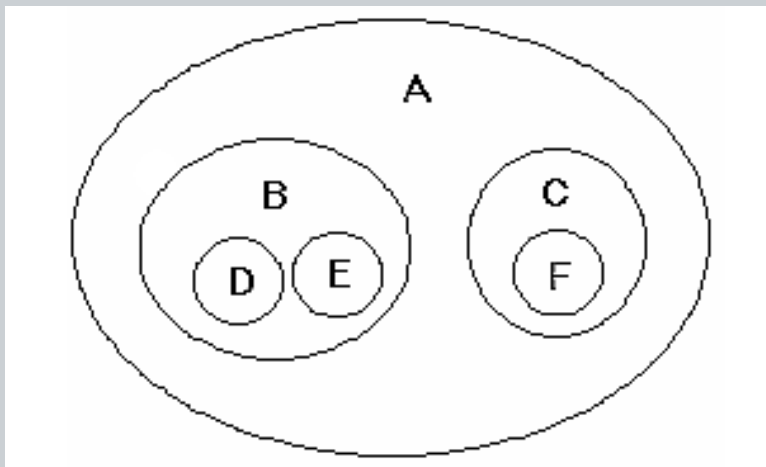
Representações

- Representação hierárquica



Representações

- Representação por conjuntos (diagrama de inclusão)



Representações

- ▶ Representação por expressão parentetizada (parênteses aninhados)
 - ▶ Cada conjunto de parênteses correspondentes contém um nodo e seus filhos. Se um nodo não tem filhos, ele é seguido por um par de parênteses sem conteúdo.

(A (B (D () E ())) (C (F ())))

Representações

- ▶ Representação por expressão não parentetizada
 - ▶ Cada nodo é seguido por um número que indica sua quantidade de filhos, e em seguida por cada um de seus filhos, representados do mesmo modo.

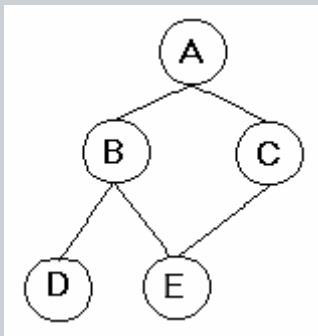
A 2 B 2 D 0 E 0 C 1 F 0

Representações

- ▶ As duas primeiras representações permitem uma melhor visualização das árvores;
- ▶ As duas últimas, por sua vez, facilitam a persistência dos nodos das árvores (em arquivos, por exemplo), possibilitando assim a sua reconstituição.

Representações

- Como, por definição, os subconjuntos c_1, c_2, \dots, c_n são disjuntos, cada nodo pode ter apenas um pai. A representação a seguir, por exemplo, não corresponde a uma árvore.



Definições

- ▶ A linha que liga dois nodos da árvore denomina-se aresta;
- ▶ Existe um caminho entre dois nodos A e B da árvore, se a partir do nodo A é possível chegar ao nodo B percorrendo as arestas que ligam os nodos entre A e B;
- ▶ Existe sempre um caminho entre a raiz e qualquer nodo da árvore.

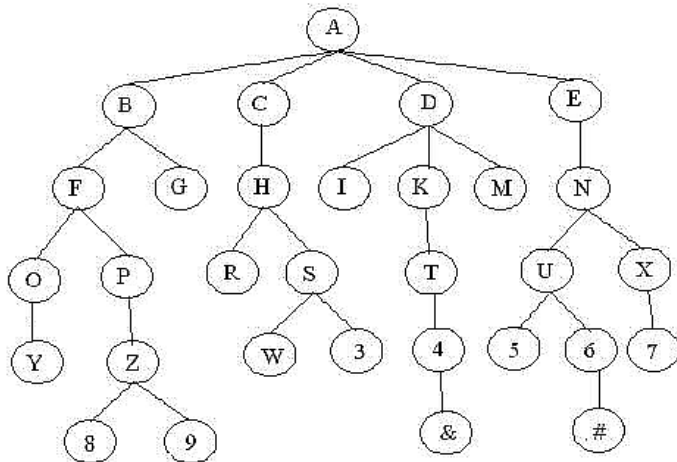
Definições

- ▶ Se houver um caminho entre A e B, começando em A diz-se que A é um nodo ancestral de B e B é um nodo descendente de A
- ▶ Se este caminho contiver uma única aresta, diz-se que A é o nodo pai de B e que B é um nodo filho de A;
- ▶ Dois nodos que são filhos do mesmo pai são denominados nodos irmãos;
- ▶ Qualquer nodo, exceto a raiz, tem um único nodo pai.

Definições

- ▶ Se um nodo não possui nodos descendentes, ele é chamado de folha ou nodo terminal da árvore;
- ▶ **Grau de um nodo:** é o número de nodos filhos do mesmo. Um nodo folha tem grau zero;
- ▶ **Nível de um nodo:** a raiz tem nível 0. Seus descendentes diretos têm nível 1, e assim por diante;
- ▶ **Grau da árvore:** é igual ao grau do nodo de maior grau da árvore;
- ▶ **Nível da árvore:** é igual ao nível do nodo de maior nível da árvore.

Exercício



Exercício

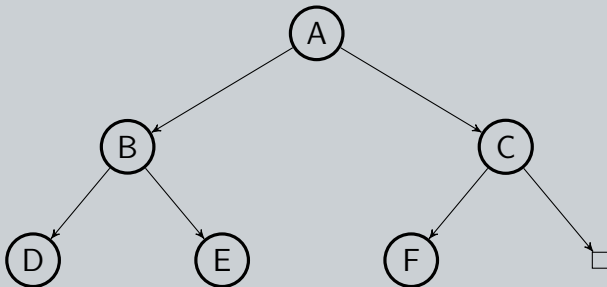
- ▶ Qual é a raiz da árvore?
- ▶ Quais são os nodos terminais?
- ▶ Qual o grau da árvore?
- ▶ Qual o nível da árvore?
- ▶ Quais são os nodos descendentes do nodo D?
- ▶ Quais são os nodos ancestrais do nodo #?
- ▶ Os nodos 4 e 5 são nodos irmãos?
- ▶ Há caminho entre os nodos C e S?
- ▶ Qual o nível do nodo 5?
- ▶ Qual o grau do nodo A?

Árvores Binárias

- ▶ A inclusão de limitações estruturais define tipos específicos de árvores;
- ▶ Até agora, as árvores vistas possuíam nenhuma limitação quanto ao grau máximo de cada nodo;
- ▶ Uma árvore binária é uma árvore cujo grau máximo de cada nodo é 2. Essa limitação define uma nomenclatura específica:
 - ▶ Os filhos de um nodo são classificados de acordo com sua posição relativa à raiz;
 - ▶ Assim, distinguem-se o filho da esquerda e o filho da direita e, conseqüentemente, a subárvore da esquerda e a subárvore da direita.

Árvores Binárias

- Exemplo de árvore binária;



Modelagem: Nodo de uma árvore binária

- ▶ Necessitamos:
 - ▶ Um ponteiro para o filho localizado à esquerda;
 - ▶ Um ponteiro para o filho localizado à direita;
 - ▶ Um ponteiro **genérico** o dado que vamos armazenar.
- ▶ Pseudo-código:

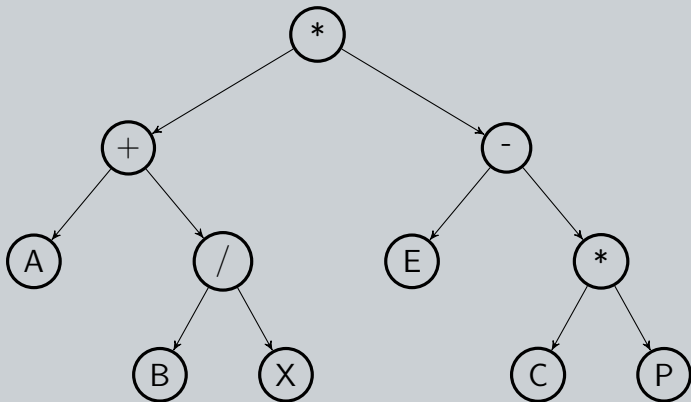
```
estrutura Nodo {  
    Nodo *_filhoEsquerda;  
    Nodo *_filhoDireita;  
    T      *_dado;  
};
```

Percursos em Árvores Binárias

- ▶ O percurso em árvores binárias corresponde ao caminhamento executado em listas:
 - ▶ Partimos de um nodo inicial (raiz) e visitamos todos os demais nodos em uma ordem previamente especificada;
- ▶ Como exemplo, considere uma árvore binária utilizada para representar uma expressão (com as seguintes restrições):
 - ▶ Cada operador representa uma bifurcação;
 - ▶ Seus dois operandos correspondentes são representados por suas subárvores.

Percursos em Árvore Binárias

Expressão: $(A + (B / X)) * (E - (C * P))$



Percursos em Árvores Binárias

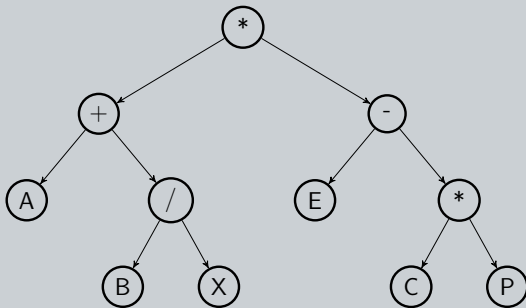
- ▶ Existem três ordens para se percorrer uma árvore binária que são consequência natural da estrutura da árvore, considerando filho à esquerda (e), filho à direita (d) e raiz (r):
 - ▶ Preordem(r,e,d) – *Preorder*;
 - ▶ Emordem(e,r,d) – *Inorder*;
 - ▶ Pósordem(e,d,r) – *Postorder*.

Percursos em Árvores Binárias

- ▶ Essas ordens são definidas recursivamente (definição natural para uma árvore) e em função da raiz(r), da subárvore esquerda(e) e da subárvore direita(d):
 - ▶ Preordem(r,e,d): visite a raiz ANTES das subárvores;
 - ▶ Emordem(e,r,d): visite primeiro a subárvore ESQUERDA, depois a RAIZ e depois a subárvore DIREITA;
 - ▶ Pósordem(e,d,r): visite a raiz DEPOIS das subárvores;
- ▶ As subárvores são SEMPRE visitadas da esquerda para a direita.

Percursos em Árvores Binárias

- ▶ Se percorrermos a árvore anterior usando as ordens definidas, teremos as seguintes seqüências:
 - ▶ Preordem (notação prefixada) : $* + A / B X - E * C P$
 - ▶ Emordem (notação infixada) : $A + B / X * E - C * P$
 - ▶ Pósordem (notação posfixada) : $A B X / + * E C P - *$



Percurso em Preordem

```
Lista* Preordem(Nodo *raiz)
início
  ListaEncadeada* lista = iniciaListaEncadeada();
  se raiz != NULO então
    adicionaNoFim(lista, raiz->_dado);
    Preordem(raiz->_filhoEsquerda);
    Preordem(raiz->_filhoDireita);
  fim se
fim
```

Percurso em Emordem

```
Lista* EmOrdem(Nodo *raiz)
início
  ListaEncadeada* lista = iniciaListaEncadeada();
  se raiz != NULO então
    EmOrdem(raiz->_filhoEsquerda);
    adicionaNoFim(lista, raiz->_dado);
    EmOrdem(raiz->_filhoDireita);
  fim se
fim
```

Percurso em Posordem

```
Lista PosOrdem(tNodo *raiz)
início
  ListaEncadeada* lista = iniciaListaEncadeada();
  se raiz != NULO então
    PosOrdem(raiz->_filhoEsquerda);
    PosOrdem(raiz->_filhoDireita);
    adicionaNoFim(raiz->_dado);
  fim se
fim
```

Árvores Binárias de Busca

- ▶ Árvores (binárias) são muito utilizadas para se representar um grande conjunto de dados onde se deseja encontrar um elemento de acordo com a sua chave.
- ▶ Definição - Árvore Binária de Busca (Niklaus Wirth):
 - ▶ “Uma árvore que se encontra organizada de tal forma que, para cada nodo t_i , todas as chaves (_dado) da subárvore à esquerda de t_i são menores que t_i e à direita são maiores (ou iguais) que t_i ”;
- ▶ Termo em Inglês: Search Tree.

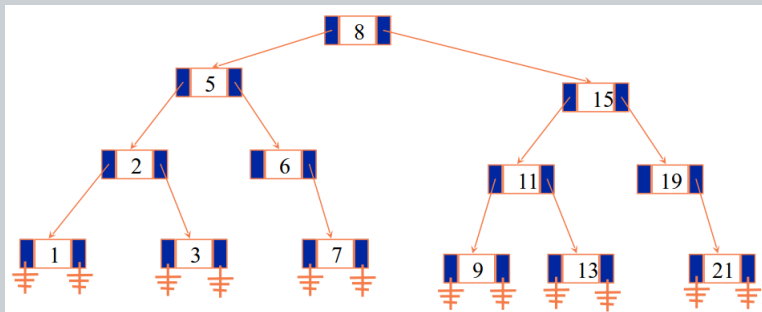
Características de Árvores Binárias de Busca

- ▶ Em uma árvore binária de busca é possível encontrar-se qualquer chave existente descendo-se pela árvore:
 - ▶ Sempre à esquerda toda vez que a chave procurada for menor do que a chave do nodo visitado;
 - ▶ Sempre à direita toda vez que for maior ou igual;
- ▶ A escolha da direção de busca só depende da chave que se procura e da chave que o nodo atual possui.

Características de Árvores Binárias de Busca

- ▶ A busca de um elemento em uma árvore balanceada com n elementos toma tempo médio $< \log(n)$, sendo a busca então $O(\log n)$;
- ▶ Graças à estrutura de árvore a busca poderá ser feita com apenas $\log(n)$ comparações de elementos.

Exemplo de árvore binária de busca

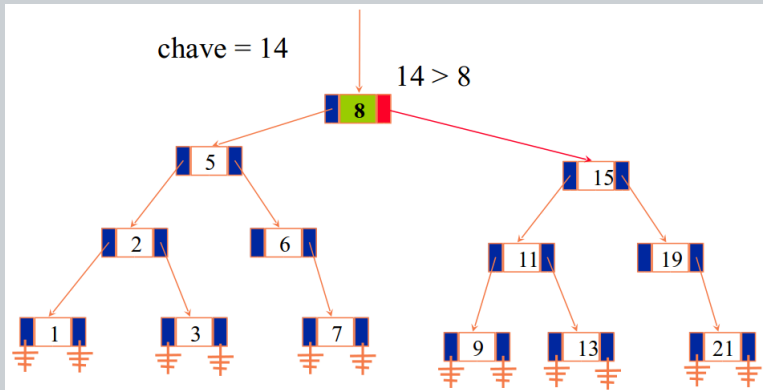


Algoritmo de Busca

```
Nodo* busca (T dado, Nodo* raiz)
início
    enquanto(raiz != NULO E raiz->_dado != dado) faça
        // Esquerda ou direita.
        se (raiz->_dado < dado) então
            raiz <- raiz->filhoDireita
        senão
            raiz <- raiz->filhoEsquerda;
        fim se
    fim enquanto
    retorne raiz;
fim
```

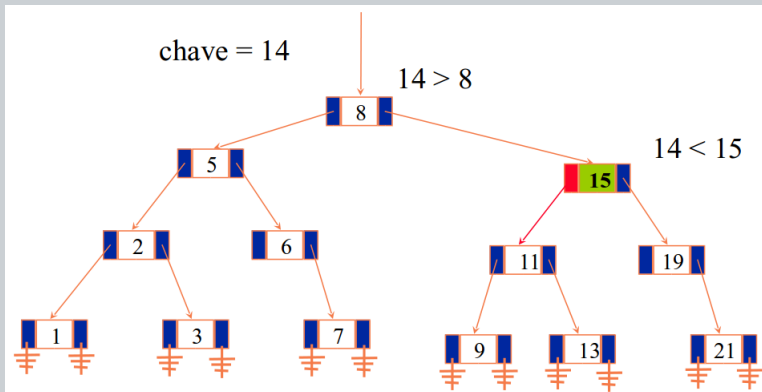

Exemplo

Inserção de um elemento com chave = 14



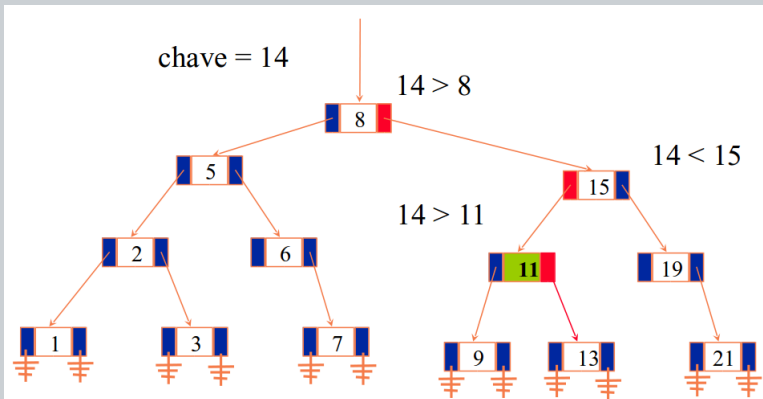
Exemplo

Inserção de um elemento com chave = 14



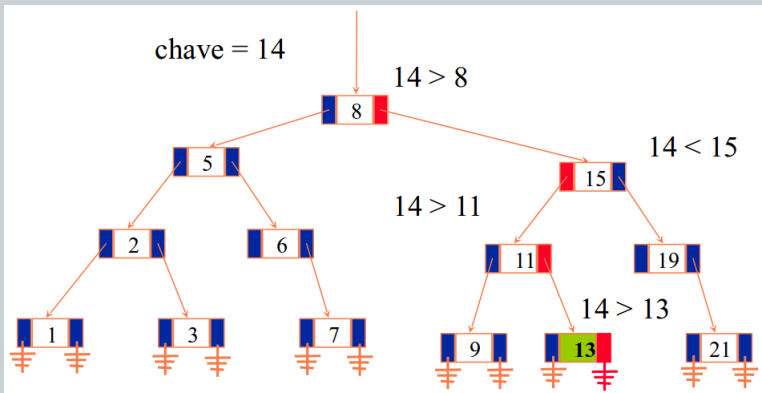
Exemplo

Inserção de um elemento com chave = 14



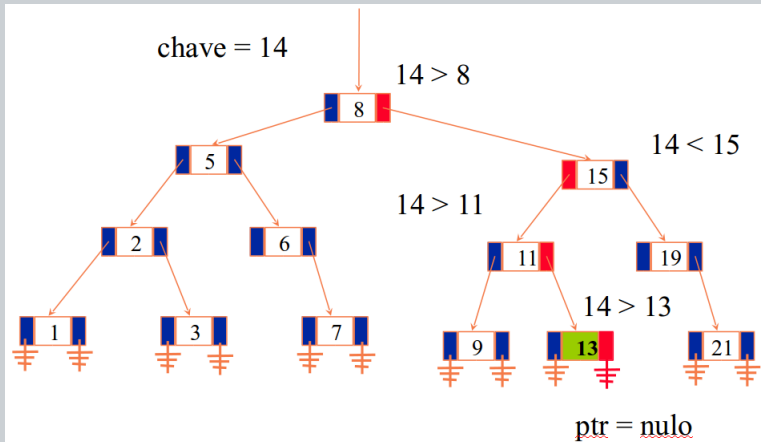
Exemplo

Inserção de um elemento com chave = 14



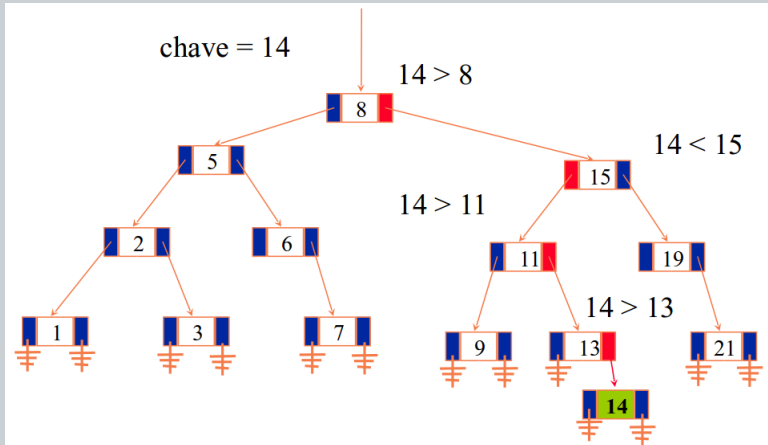
Exemplo

Inserção de um elemento com chave = 14



Exemplo

Inserção de um elemento com chave = 14



Algoritmo de Inserção

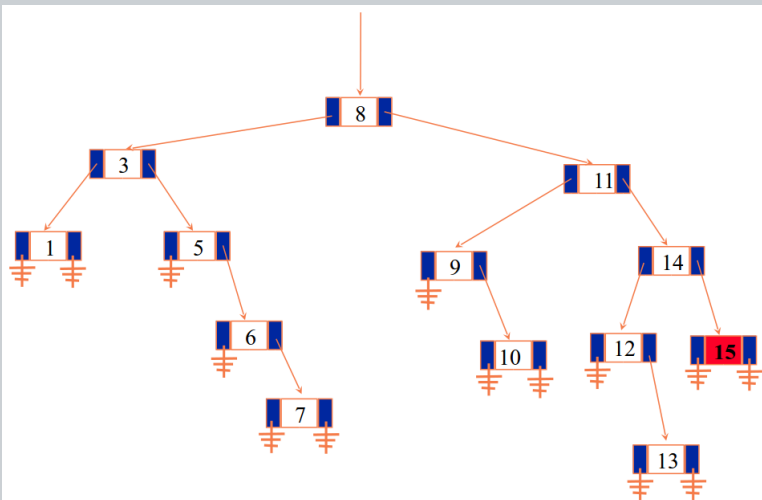
```
void inserir(Nodo* raiz, T* dado)
início
  se (dado < raiz->_dado) então
    // Inserção à esquerda.
    se (raiz->_filhoEsquerda = NULO) então
      Nodo* oNovo <- aloque(Nodo); oNovo->_dado <- dado;
      oNovo->filhoEsquerda <- NULO; oNovo->filhoDireita <- NULO;
      raiz->filhoÀEsquerda <- oNovo;
    senão
      inserir(raiz->_filhoEsquerda, dado);
  fim se
  senão
    // Inserção à direita.
    se (raiz->filhoDireita = NULO) então
      Nodo* oNovo <- aloque(Nodo); oNovo->_dado <- dado;
      oNovo->filhoEsquerda <- NULO; oNovo->filhoDireita <- NULO;
      raiz->filhoÀDireita <- oNovo;
    senão
      inserir(raiz->filhoDireita, dado);
  fim se
fim se
fim
```

Algoritmo de Deleção

- ▶ A deleção é mais complexa do que a inserção;
- ▶ A razão básica é que a característica organizacional da árvore não deve ser quebrada:
 - ▶ A subárvore da direita de um nodo não deve possuir chaves menores do que o pai do nodo eliminado;
 - ▶ A subárvore da esquerda de um nodo não deve possuir chaves maiores do que o pai do nodo eliminado.
- ▶ Para garantir isso, o algoritmo de deleção deve remanejar os nodos.

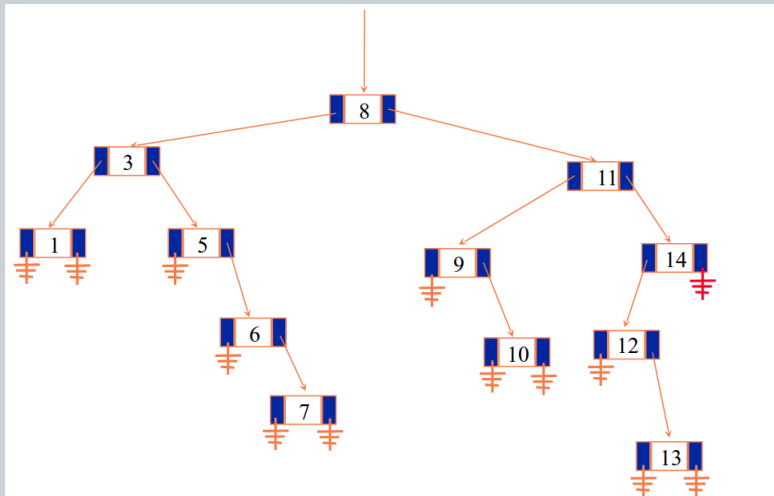
Exemplo

Deleção do nodo com chave = 15



Exemplo

Deleção do nodo com chave = 15

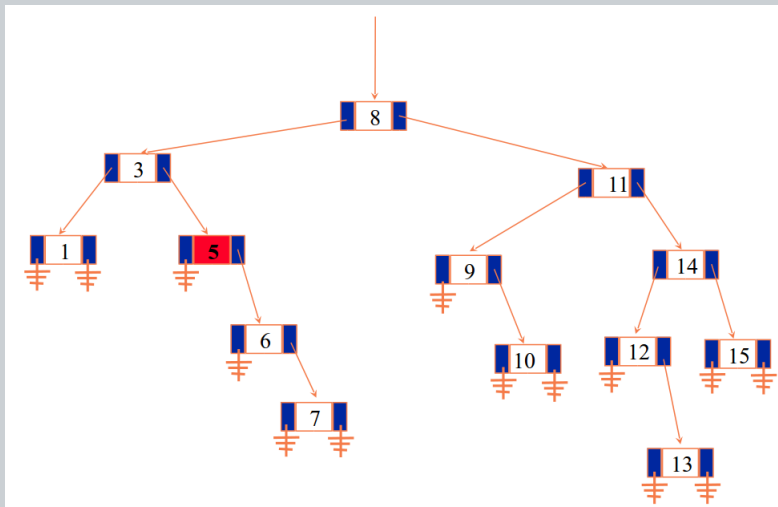


Deleção em uma Arvore de Busca Binária

- ▶ Se o nodo possuir somente uma subárvore filha:
 - ▶ Podemos simplesmente mover esta subárvore toda para cima;
 - ▶ O único sucessor do nodo a ser excluído será um dos sucessores diretos do pai do nodo a ser eliminado;
 - ▶ Se o nodo a ser excluído é filho esquerdo de seu pai, o seu filho será o novo filho esquerdo deste e vice-versa.

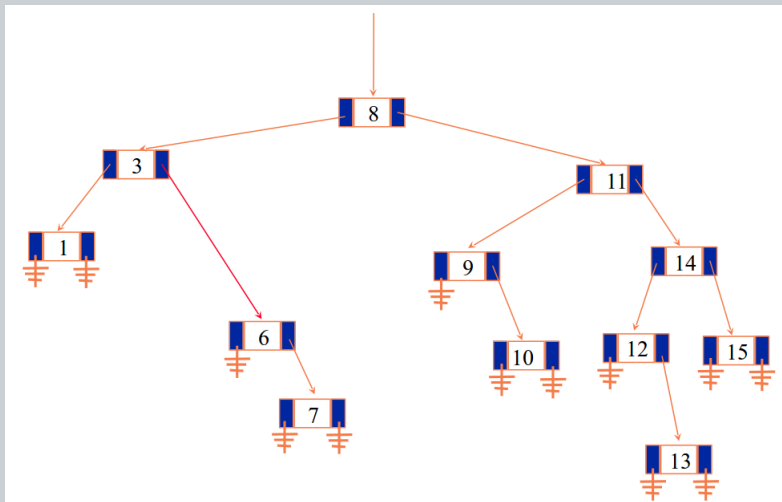
Exemplo

Deleção do nodo com chave = 5



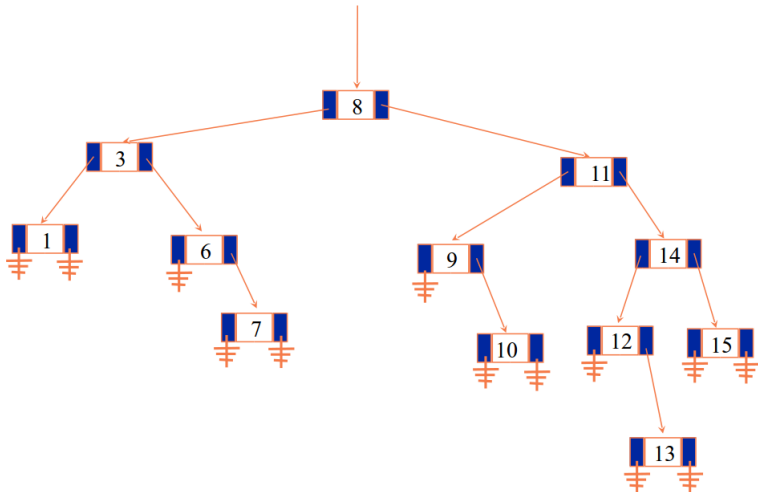
Exemplo

Deleção do nodo com chave = 5



Exemplo

Deleção do nodo com chave = 5

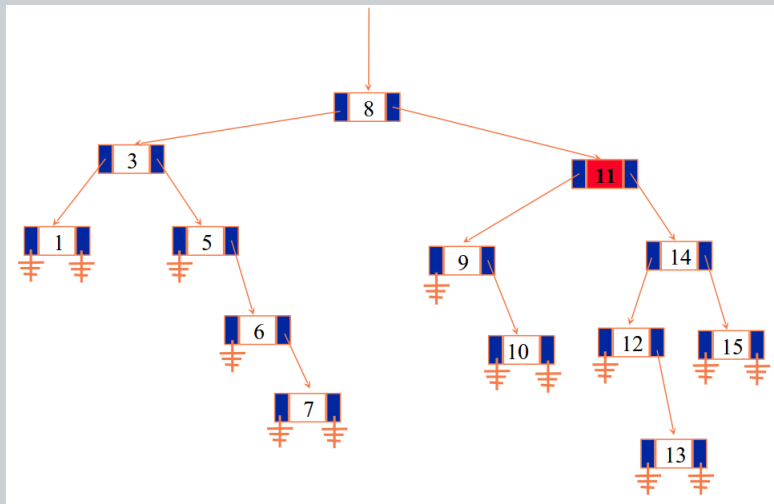


Deleção em uma Arvore de Busca Binária

- ▶ Se o nodo possuir duas subárvores filhas:
 - ▶ Se o filho à direita não possui subárvore esquerda, é ele quem ocupa o seu lugar;
 - ▶ Se possuir uma subárvore esquerda, a raiz desta será movida para cima e assim por diante;
 - ▶ A estratégia geral (Mark Allen Weiss) é sempre substituir a chave retirada pela menor chave da subárvore direita.

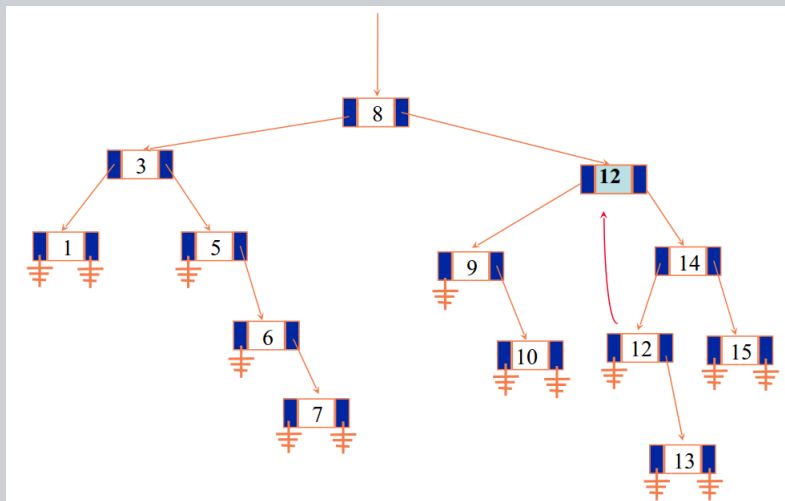
Exemplo

Deleção do nodo com chave = 11



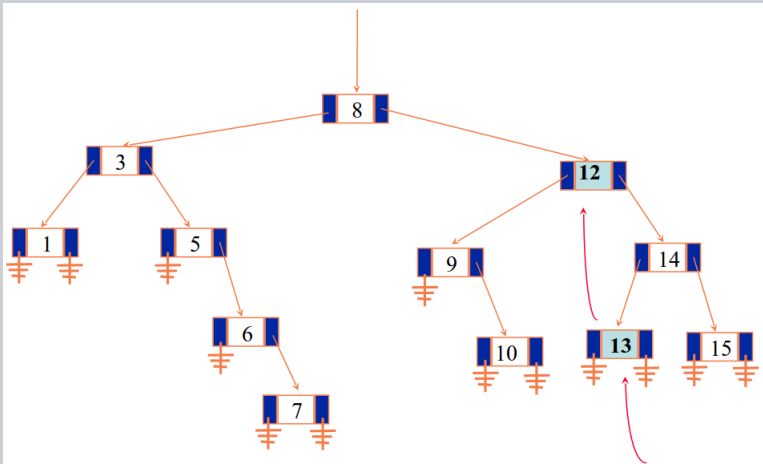
Exemplo

Deleção do nodo com chave = 5



Exemplo

Deleção do nodo com chave = 5



Algoritmo de Deleção

```
tNodo* delete(info: tInfo, arv: *tNodo)
    tmp, filho: *tNodo;
    início
        se (arv = NULO) então retorne arv
        senão
            se (info < arv->info) // Vá à esquerda.
                arv->filhoÀEsquerda <- delete(info, arv->filhoÀEsquerda);
                retorne arv;
            senão
                se (info > arv->info) // Vá à direita.
                    arv->filhoÀDireita <- delete(info, arv->filhoÀDireita);
                    retorne arv;
                senão // Encontrei elemento que quero deletar.
                    se (arv->filhoÀDireita~=NULO E arv->filhoÀEsquerda~=NULO)//
                        2 filhos.
                        tmp <- mínimo(arv->filhoÀDireita); arv->info <- tmp->info;
                        arv->filhoÀDireita <-delete(arv->info,arv->filhoÀDireita);
                        retorne arv;
                    //(CONTINUA NO PROX SLIDE)
    ^^I^^I
```

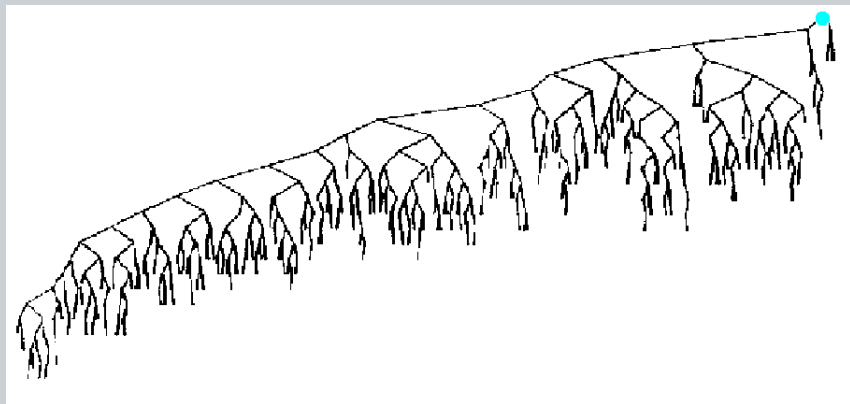
Algoritmo de Deleção

```
senão // 1 filho.  
    tmp <- arv;  
    se (arv->filhoÀDireita ~= NULO) então // Filho à direita.  
        filho <- arv->filhoÀDireita; retorne filho;  
senão  
    se (arv->filhoÀEsquerda ~= NULO) então // Filho à esquerda  
        .  
        filho <- arv->filhoÀEsquerda; retorne filho;  
senão // Folha.  
    libere arv;  
    retorne NULO;  
fim se  
fim se  
fim se  
fim se  
fim se  
fim se  
fim  
^^I^^I
```

Problemas com Árvores de Busca Binária

- ▶ Deterioração:
 - ▶ Quando inserimos utilizando a inserção simples, dependendo da distribuição de dados, pode haver deterioração;
 - ▶ Árvores deterioradas perdem a característica de eficiência de busca.

Problemas com Árvores de Busca Binária



Trabalho

- ▶ Implemente uma classe NoBinario para representar a sua árvore;
- ▶ Implemente a árvore usando Templates;
- ▶ Use as melhores práticas de orientação a objetos;
- ▶ Documente todas as classes, métodos e atributos;
- ▶ Aplique os testes unitários disponíveis no moodle da disciplina para validar sua estrutura de dados;
- ▶ Entregue até a data definida no moodle.

Perguntas????



UNIVERSIDADE FEDERAL
DE SANTA CATARINA



Este trabalho está licenciado sob uma Licença Creative Commons Atribuição 4.0 Internacional. Para ver uma cópia desta licença, visite

<http://creativecommons.org/licenses/by/4.0/>.



UNIVERSIDADE FEDERAL
DE SANTA CATARINA