CES-11

Algoritmos e Estruturas de Dados

Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

CES-11

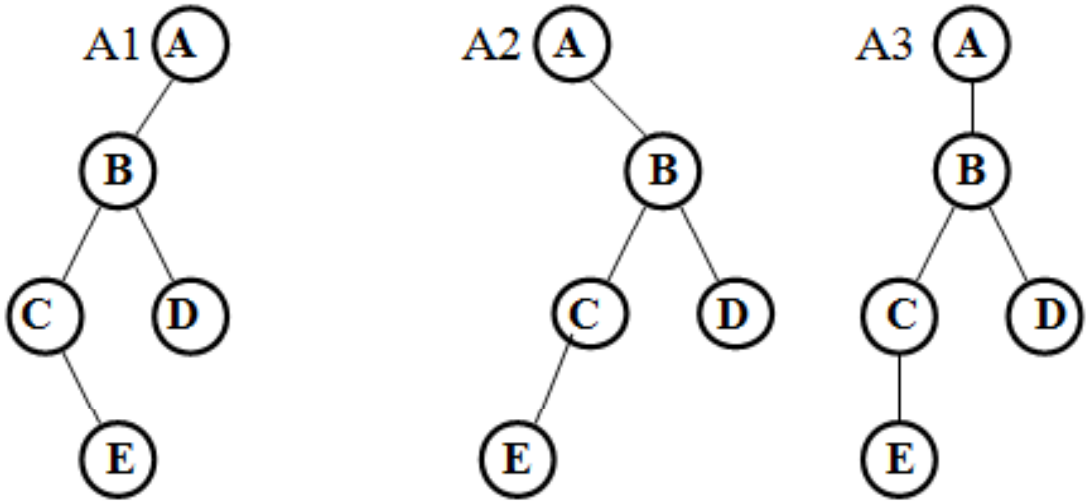
- Árvores binárias
- Árvores binárias de busca
  - Conceito
  - Operações

CES-11

- Árvores binárias
- Árvores binárias de busca
  - Conceito
  - Operações

Árvores binárias

- Uma árvore binária é:
  - uma árvore vazia;
  - ou uma árvore onde qualquer nó possui:
    - nenhum filho;
    - ou um filho esquerdo ou um filho direito;
    - ou os dois filhos citados.



A1 é binária? Sim

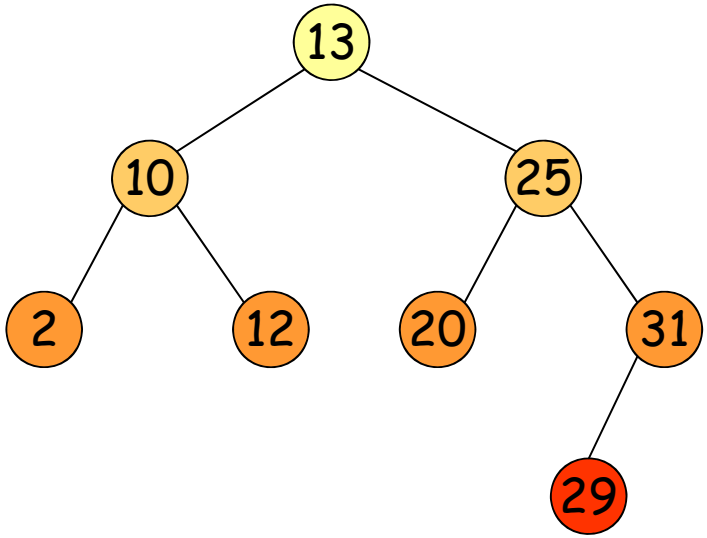
A2 é binária? Sim

A3 é binária? Não

No nó A, não há distinção entre filho esquerdo e direito...

Árvores binárias

- Exemplo de ordenação de nós numa árvore binária:
  - Por nível (largura): 13 10 25 2 12 20 31 29
  - Pré-ordem: 13 10 2 12 25 20 31 29
  - Pós-ordem: 2 12 10 20 29 31 25 13
  - Ordem-central: 2 10 12 13 20 25 29 31

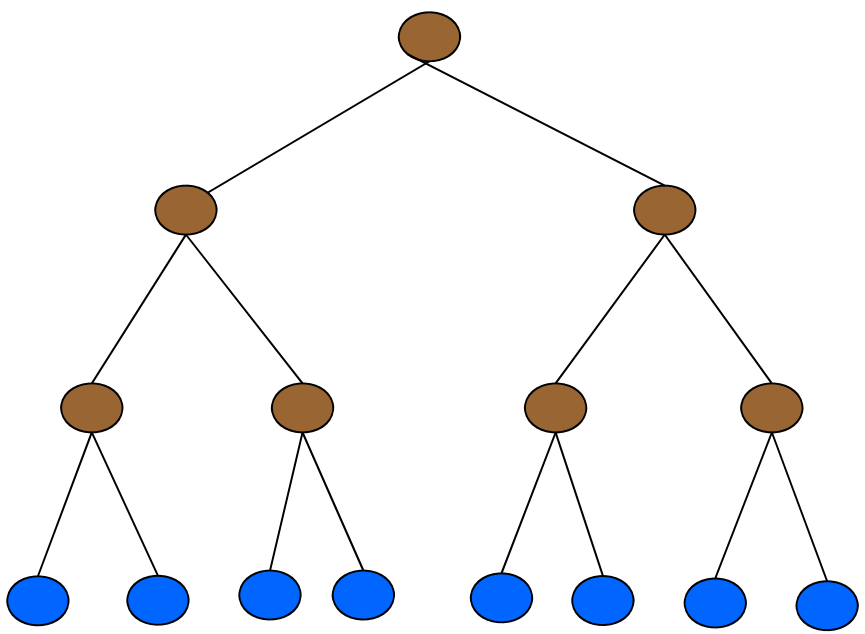


Árvores binárias

- Uma árvore binária é completa quando os nós internos têm exatamente dois filhos e as folhas têm a mesma distância da raiz.
- Uma árvore binária completa de altura h tem exatamente $2^h - 1$ nós internos e 2^h folhas.
- Numa árvore binária completa com n folhas, a distância da raiz até qualquer folha é $lg\ n$.
- Prove!!

Árvores binárias

- Considere a árvore binária completa abaixo:



h = 3 (altura)
n = 8 (folhas)

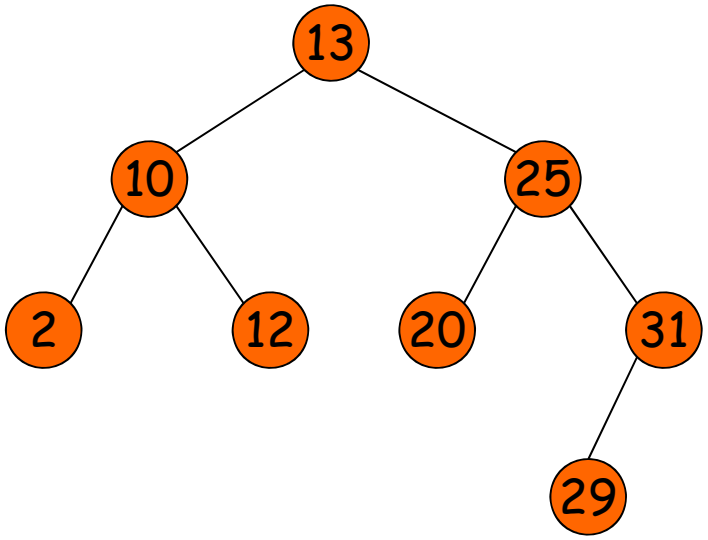
$n = 2^h$ ⟺ $h = \lg n$

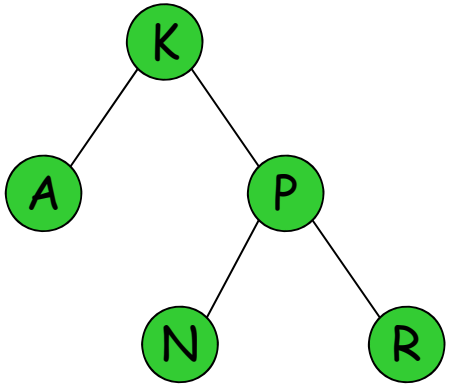
CES-11

- Árvores binárias
- Árvores binárias de busca
  - Conceito
  - Operações

Árvores binárias de busca

- Definição: Uma árvore binária é de busca se *em cada um de seus nós* :
  - as chaves armazenadas na sua sub-árvore esquerda são menores que a chave do próprio nó;
  - as chaves armazenadas na sua sub-árvore direita são maiores que a chave do próprio nó.
- Exemplos:



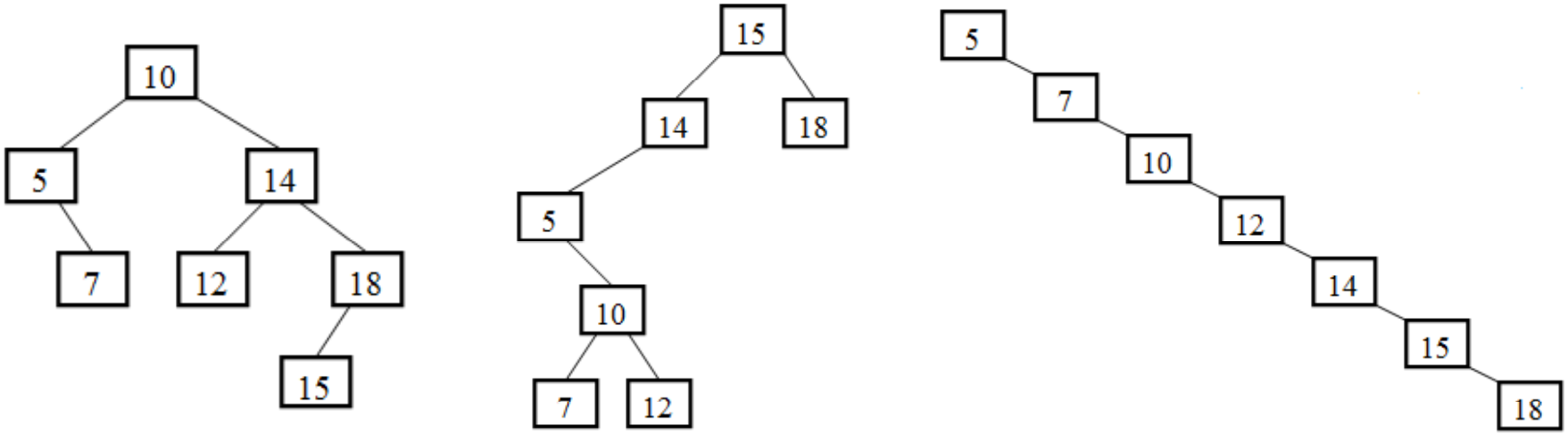


Árvores binárias de busca

- Uma utilização: implementar dicionários
  - Consiste em um conjunto de chaves ou elementos **distintos** (sem repetições)
  - De modo geral, cada chave tem uma informação associada
  - Operações básicas:
    - Busca de uma chave (teste de pertinência)
    - Inserção de uma chave
    - Eliminação de uma chave
    - Acesso sequencial e ordenado a todas as chaves
  - Aplicações comuns: cadastro de clientes, bancos de dados, etc.

Outro exemplo

- Seja o conjunto de chaves $C = \{5, 7, 10, 12, 14, 15, 18\}$.
- Há várias árvores binárias de busca possíveis:



- Como veremos adiante, esses formatos terão impacto no desempenho das operações.

CES-11

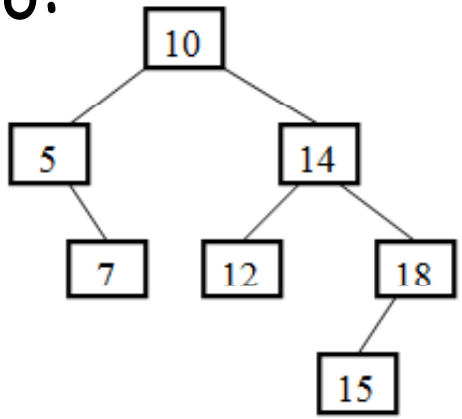
- Árvores binárias
- Árvores binárias de busca
  - Conceito
  - Operações

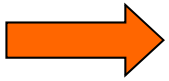
Declarações

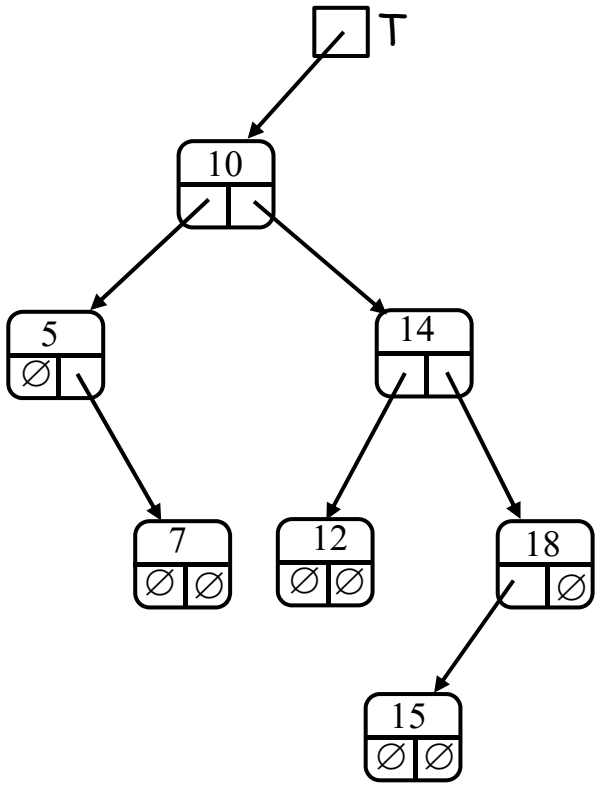
```
struct Celula  {
        int chave;
        // omitimos os eventuais dados associados
        Celula *fesq, *fdir;
};
typedef Celula *Dicionario;

Dicionario T;
```

- Exemplo:





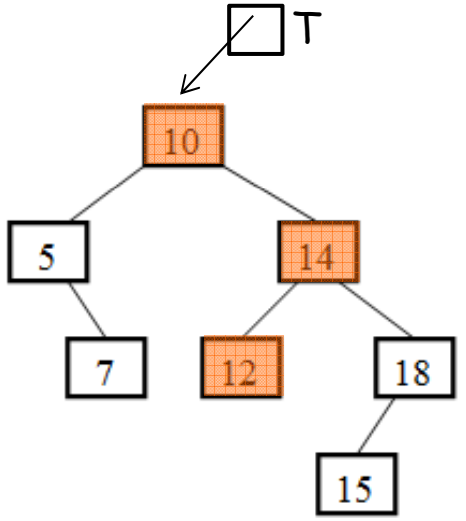


Busca de chave

- Algoritmo de busca é simples e direto.
- Repetição que começa na raiz:
  - Compare com a chave armazenado no nó.
  - Se for igual, a busca chegou ao fim.
  - Se for menor, vá para a sub-árvore esquerda.
  - Se for maior, vá para a sub-árvore direita.
  - Se não houver como continuar, a chave não está na árvore.

Implementação da busca

```
bool membro (int x, Dicionario dic) {
      if (dic == NULL)
         return false;
      if (x == dic->chave)
         return true;
      if (x < dic->chave)
         return membro(x, dic->fesq);
      return membro(x, dic->fdir);
}
```

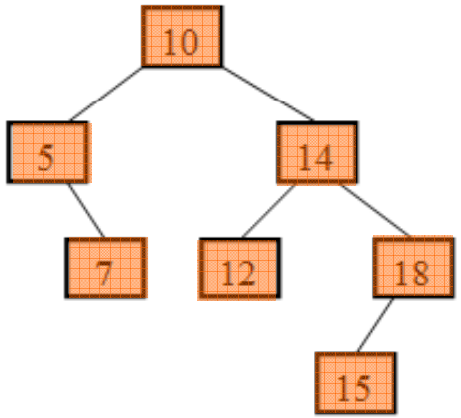


- Simulação para `membro(13,T)`
- Código poderia ser iterativo
- Pior caso: tempo proporcional à altura da árvore

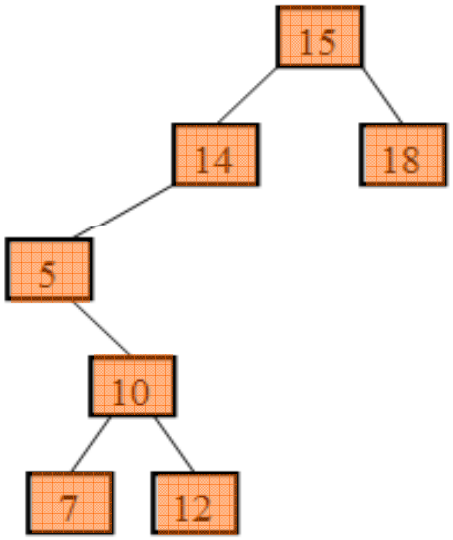
Inserção de chave

- Operação **`inserir(x,&dic)`**
  - Insere a chave **`x`** no dicionário **`dic.`**
  - Ideia semelhante à busca.
- A **sequência** de inserções determinará o **formato** da árvore.
  - Exemplos:

10, 14, 5, 12, 7, 18, 15



15, 14, 5, 18, 10, 12, 7



Implementação da inserção

```
void inserir (int x, Dicionario *dic)  {
   if (*dic == NULL) {
      *dic = (Celula *) malloc (sizeof(Celula));
      (*dic)->chave = x;
      (*dic)->fesq = NULL;
      (*dic)->fdir = NULL;
   }
   else if (x < (*dic)->chave)
      inserir (x, &(*dic)->fesq);
   else if (x > (*dic)->chave)
      inserir (x, &(*dic)->fdir);
}
```

Quando se chega a um nó nulo, um novo nó é criado

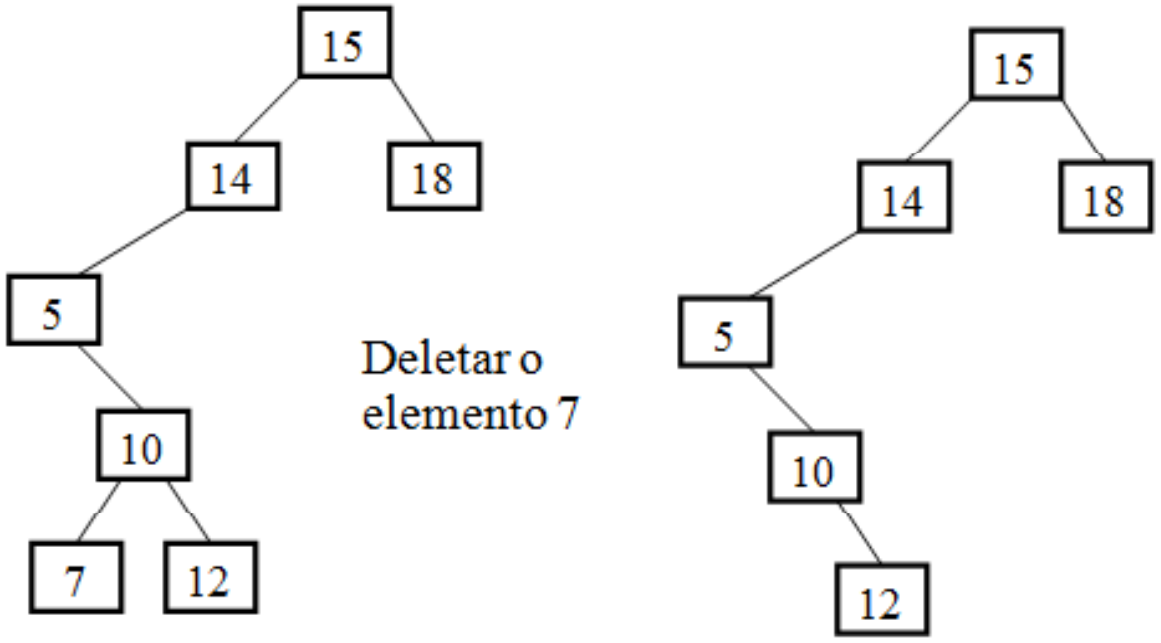
Chamadas recursivas

E quando **x** for igual a **(*dic)->chave**?

Não faz nada, pois no dicionário não há chaves repetidas

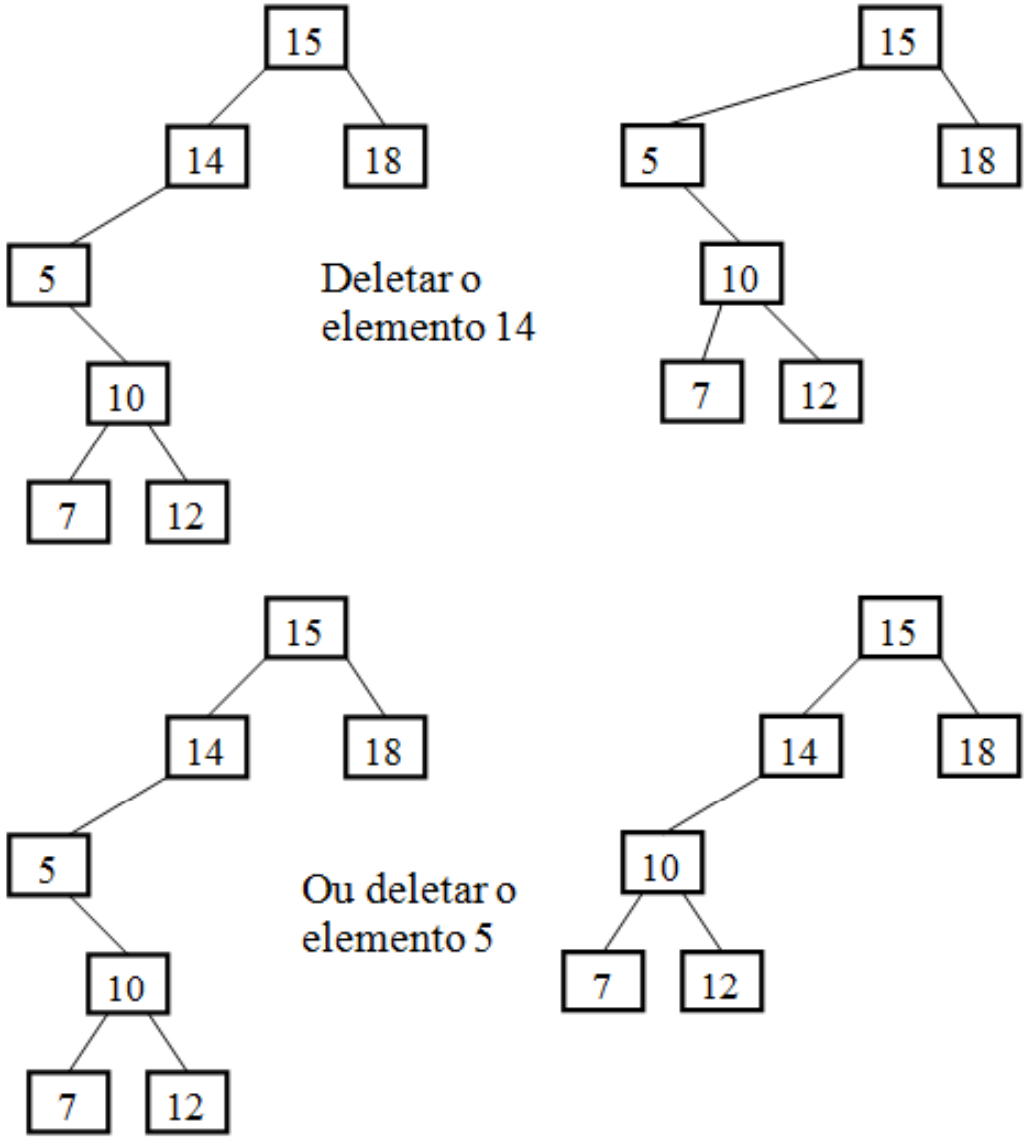
Eliminação de chave

- **<u>Caso 1</u>: a chave a ser eliminada está numa folha.**
- Solução: basta retirá-la da estrutura.
- Exemplo:



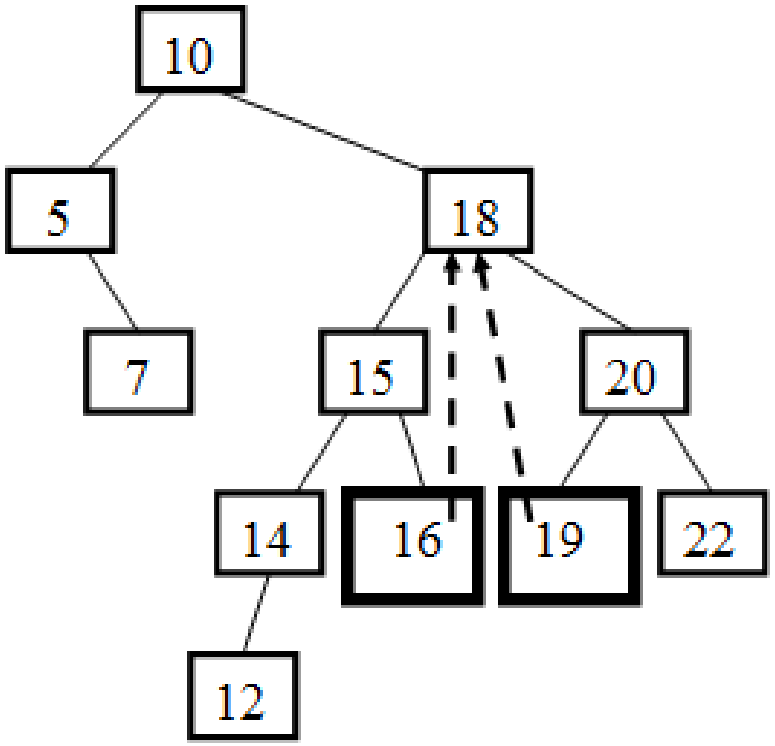
Eliminação de chave

- **<u>Caso 2</u>: a chave a ser eliminada está em um nó com um único filho.**
- Solução: basta que esse filho único ocupe a posição do pai.
- Exemplos:



Eliminação de chave

- **<u>Caso 3</u>: a chave a ser eliminada está em um nó com dois filhos.**
- Solução: encontrar uma chave que possa ocupar o nó da chave a ser eliminada.
- Veja o exemplo ao lado: eliminação da chave 18.
- Duas opções:
  - Maior chave da sub-árvore esquerda
  - Menor chave da sub-árvore direita desse nó

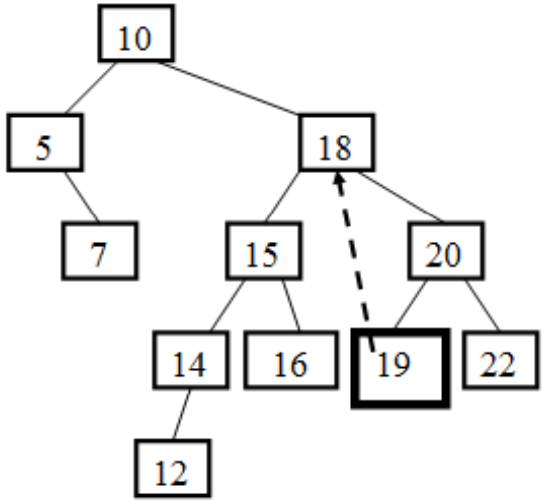


Opção escolhida por convenção

Função auxiliar

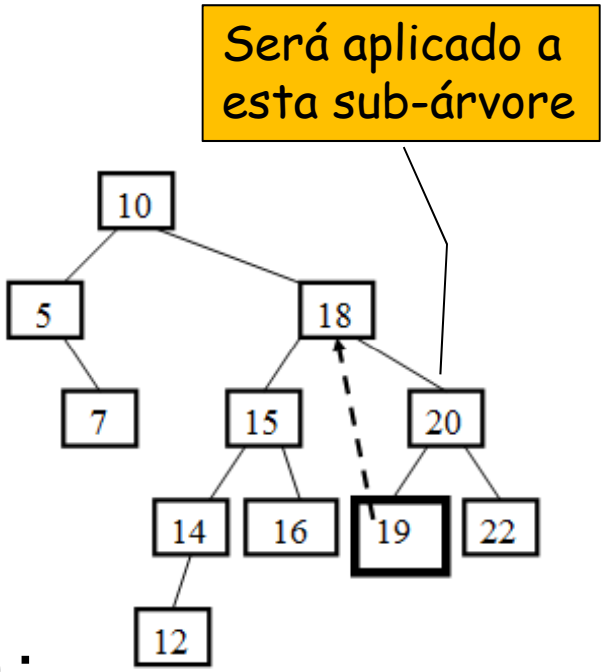
- Vamos definir a função **`int eliminarMin (Dicionario *dic)`**
  - Elimina o nó de **`dic`** com menor chave, retornando seu valor.

- No exemplo ao lado:
  - Deseja-se eliminar a chave **18**.
  - Aplica-se essa função auxiliar à sub-árvore direita do nó que contém **18**.
  - Será eliminado o nó da chave **19**, cujo valor, ao ser retornado, será armazenado no nó que continha **18**.



Implementação da função auxiliar

```
int eliminarMin (Dicionario *dic) {
   Celula *p;
   int ret;
   if ((*dic)->fesq == NULL) {
      ret = (*dic)->chave;
      p = *dic;
      *dic = (*dic)->fdir;
      free(p);
   }
   else
      ret = eliminarMin (&(*dic)->fesq);
   return ret;
}
```

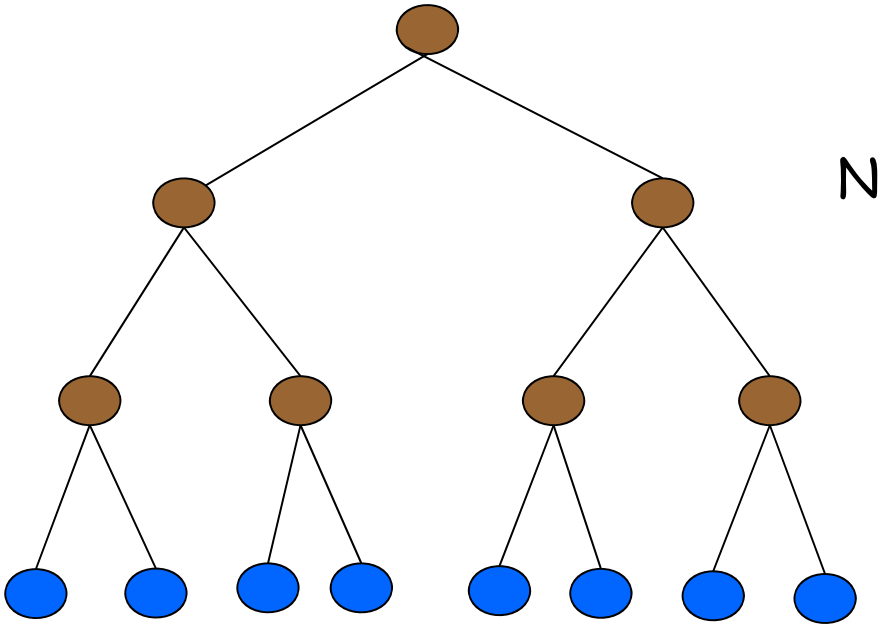


Implementação da eliminação

```
void eliminar (int x, Dicionario *dic) {
   Celula *p;
   if (*dic != NULL) {
      if (x < (*dic)->chave)                  // Busca à esquerda
         eliminar (x, &(*dic)->fesq);
      else if (x > (*dic)->chave)             // Busca à direita
         eliminar(x, &(*dic)->fdir);
      else if ((*dic)->fesq == NULL && (*dic)->fdir == NULL) {  // Caso 1: folha
          free (*dic);
          *dic = NULL;
      }
      else if ((*dic)->fesq == NULL){         // Caso 2: com um único filho
           p = *dic;
          *dic = (*dic)->fdir;
           free (p);
      }
      else if ((*dic)->fdir == NULL){
          p = (*dic);
          *dic = (*dic)->fesq;
          free (p);
      }
      else                                    // Caso 3: com dois filhos
          (*dic)->chave = eliminarMin (&(*dic)->fdir);
   }
}
```

Altura de uma árvore binária

- É fácil constatar que o tempo de pior caso dos algoritmos apresentados é proporcional à altura da árvore binária de busca.
- Considerando h como a altura de uma árvore binária com n nós, pode-se demonstrar que $n \leq 2^{h+1} - 1$, ou seja, $h = \Omega(\log n)$.
- Vejamos abaixo uma árvore binária completa de altura h = 3:



Número de folhas: $2^h = 8$

Número de nós internos: $2^h - 1 = 7$

$n = 2^{h+1} - 1 = 15$

Não existe uma árvore binária de altura 3 com mais do que 15 nós...

Balanceamento

- Os algoritmos apresentados mantiveram as propriedades de uma árvore binária de busca.
- No entanto, a aplicação aleatória desses algoritmos pode gerar árvores binárias de busca com formatos muito variados (por exemplo, de altura proporcional ao número de nós), comprometendo a eficiência das futuras operações...
- É desejável que a altura da árvore permaneça o mínimo possível (como vimos, proporcional ao logaritmo do número de nós), para que essas operações tenham sempre um bom desempenho.
- Esta propriedade, chamada *balanceamento*, pode ser garantida por outros algoritmos mais elaborados.
- Os casos mais conhecidos de árvores binárias de busca balanceadas são as árvores AVL e as rubro-negras.