Estruturas de Dados II

Árvore Binária de Busca

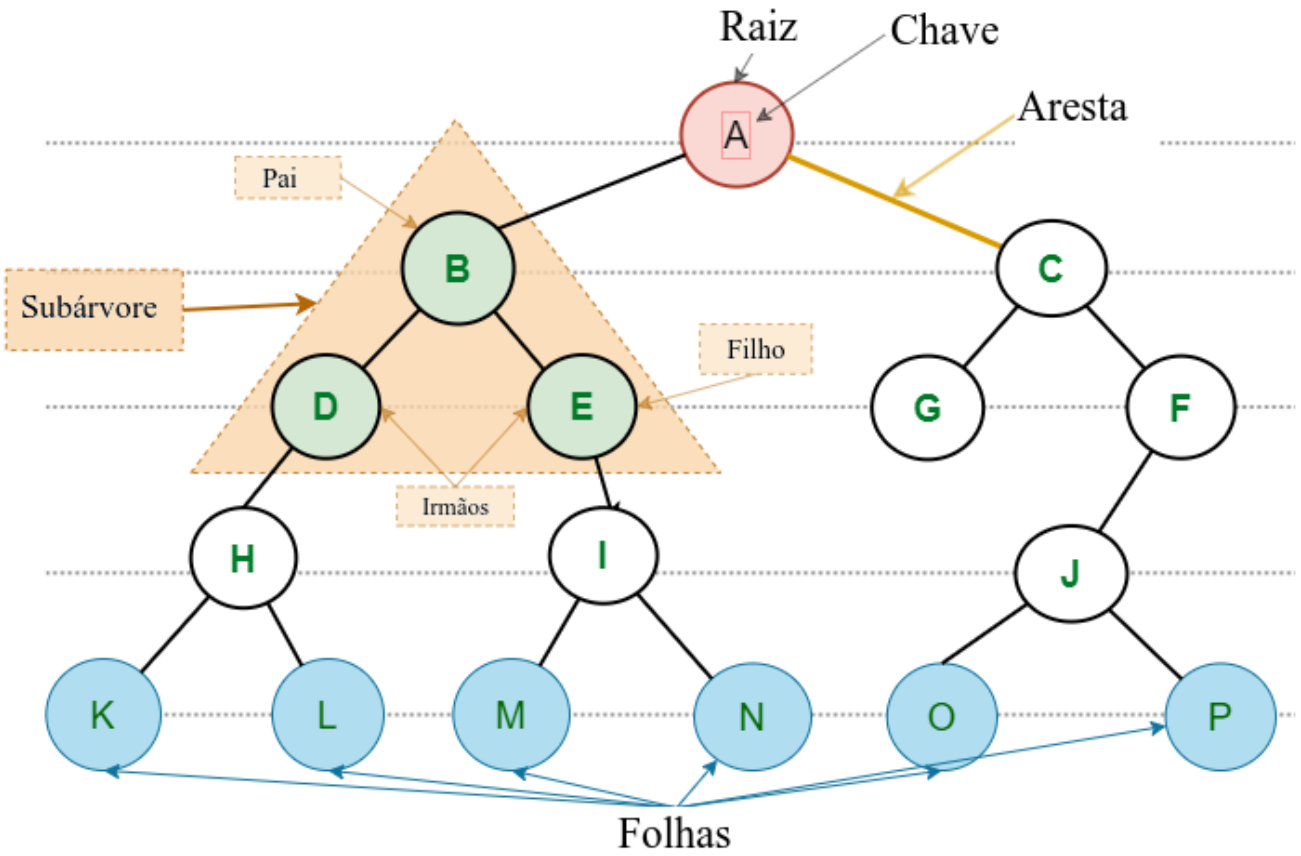
Prof. Bruno Azevedo

Instituto Federal de São Paulo

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

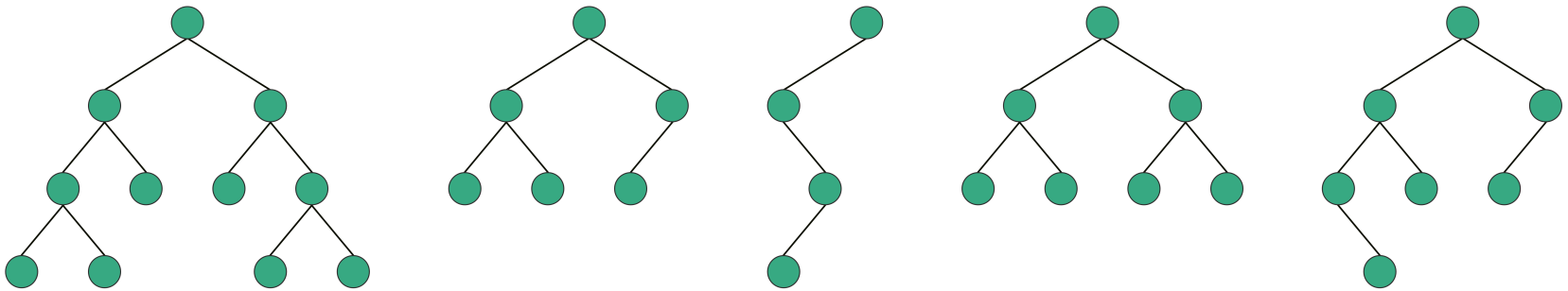
Árvore

- Na computação, uma árvore é um tipo de dado abstrato que representa uma estrutura hierárquica contendo um conjunto de nós conectados.
- Cada nó na árvore pode estar conectado a múltiplos filhos, mas deve estar conectado exatamente a um pai. Exceto pelo **nó raiz**, que não tem pai.



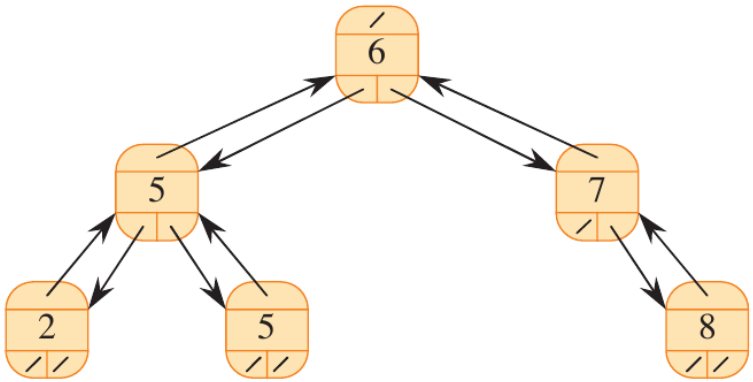
Árvore Binária

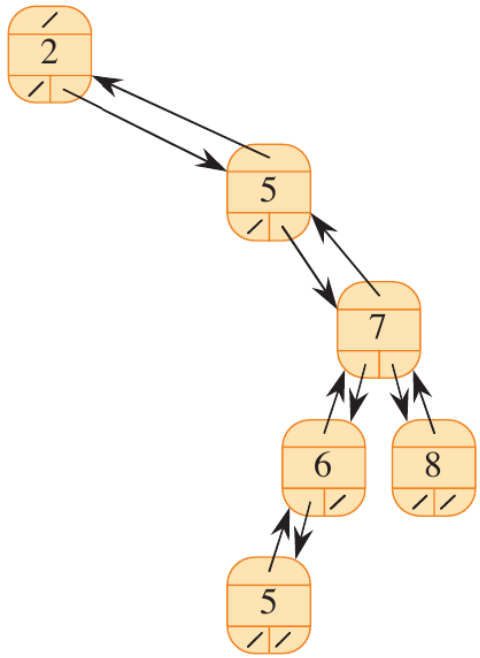
- Uma Árvore Binária é uma estrutura de dados organizada como árvore onde cada nó possui **no máximo** dois filhos.



Árvore Binária de Busca

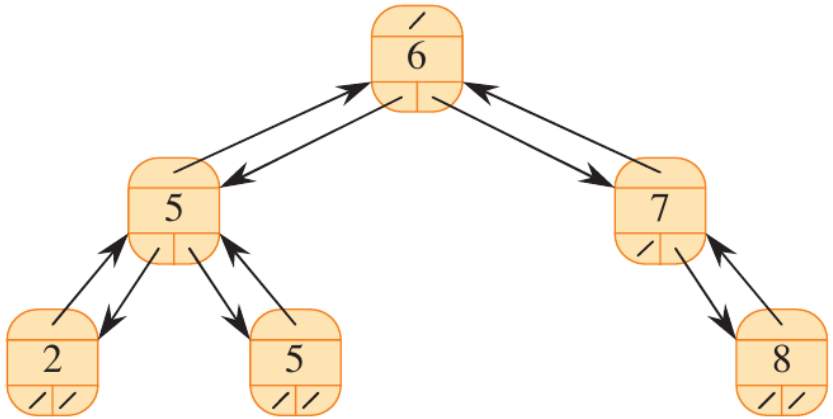
- Uma **Árvore Binária de Busca** é uma árvore binária enraizada que organiza os elementos de modo a facilitar as operações de busca, inserção e remoção.
- Pode ser representada com uma modificação simples na estrutura de lista duplamente ligada, permitindo que cada elemento se conecte a dois elementos subsequentes, em vez de apenas um, como em uma lista ligada típica.
- Cada nó identifica um filho esquerdo, um filho direito, o pai, e uma chave.





Árvore Binária de Busca

- A organização da Árvore Binária de Busca obedece a seguinte regra: dado um nó x da árvore, se y for um nó na subárvore esquerda de x, então y->chave $<$ x->chave. Se y for um nó na subárvore direita de x, então y->chave $>$ x->chave.
- Na implementação, vocês podem definir em qual lado colocarão as chaves de mesmo valor ou se proibirão tais elementos.

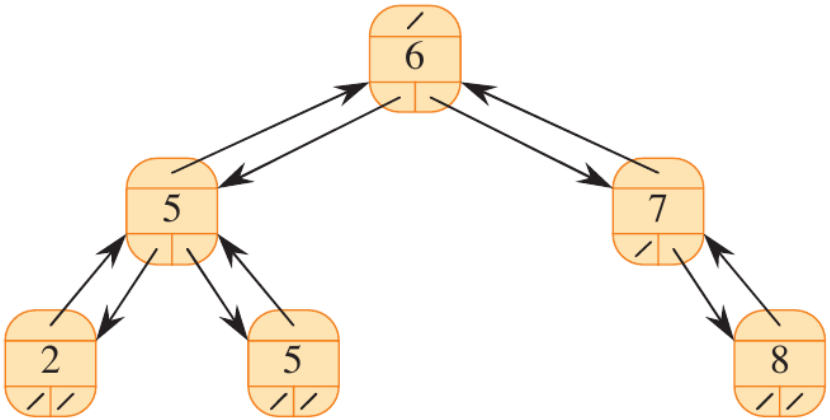


Percurso em Ordem

- Para exibirmos os valores de todas as chaves de uma árvore, de modo ordenado, podemos utilizar o seguinte algoritmo recursivo.

```
PERCURSO_EM_ORDEM (x)
if x ≠ NULL
  PercursoEmOrdem(x->esq)
  print x->chave
  PercursoEmOrdem(x->dir)
```

- Onde x é a raiz.



- Intuitivamente podem perceber que esse algoritmo executa em O(n) (mais especificamente $\Theta(n)$), já que o algoritmo é chamado duas vezes recursivamente para cada nó.

Busca

- Para buscarmos um nó com uma determinada chave podemos utilizar o seguinte algoritmo iterativo:

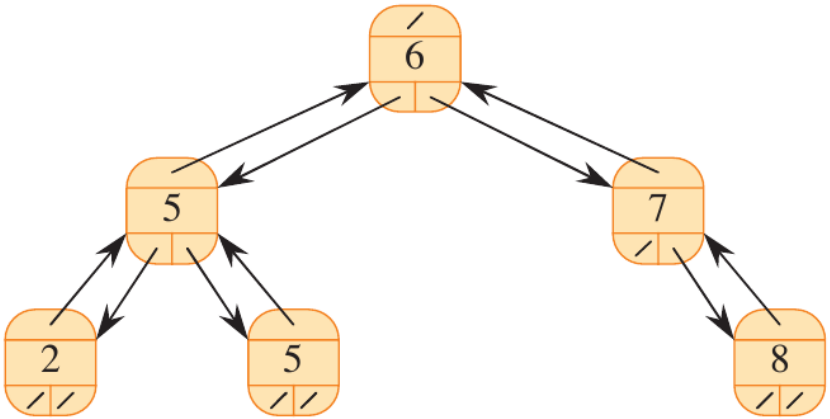
```
BUSCA_NA_ARVORE (x, k)
while x ≠ NULL and k ≠ x->chave
  if k < x->chave
    x = x->esq
  else
    x = x->dir
return x
```

- Onde `x` é a raiz da árvore e `k` é a chave buscada.
- Notem que estou colocando os elementos iguais na subárvore direita.

Busca

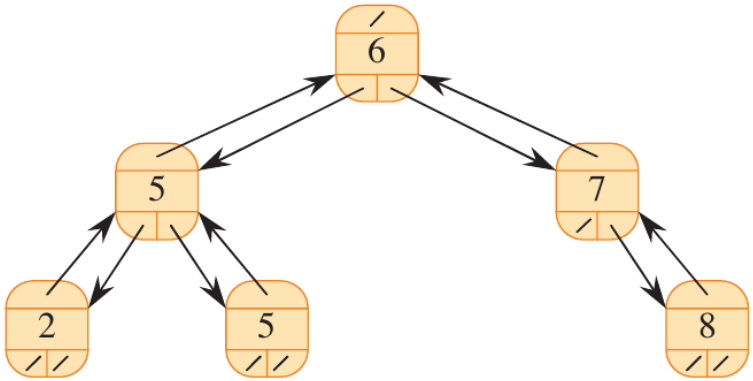
- Vamos acompanhar a execução do algoritmo na árvore da figura abaixo.

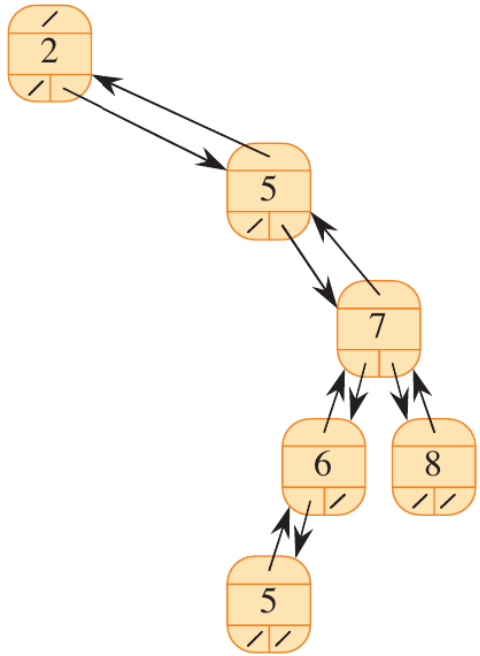
```
BUSCA_NA_ARVORE (x, k)
while x ≠ NULL and k ≠ x->chave
  if k < x->chave
    x = x->esq
  else
    x = x->dir
return x
```



Busca na Árvore

- É evidente que a busca em árvore possui complexidade de tempo O(h), onde `h` é a altura da árvore.
- Afinal, o máximo de comparações que efetuaremos será a altura da árvore mais um.
- Se temos uma árvore balanceada, onde a diferença de altura entre as subárvores esquerda e direita de qualquer nó é limitada a um, o tempo de execução será de O(log(n)).
- Se temos uma árvore degenerada, onde todos os seus nós internos têm uma única subárvore associada, o custo será de O(n).



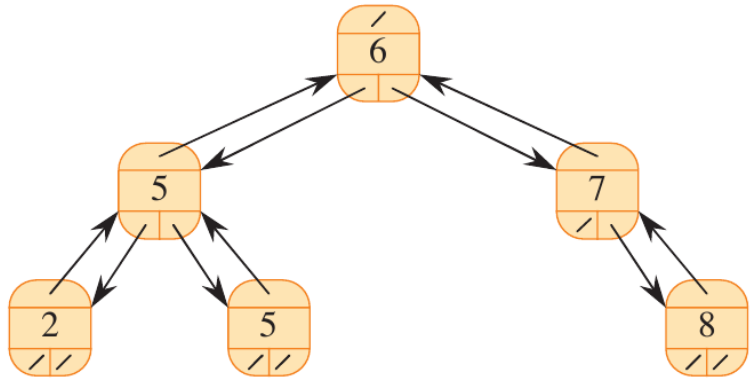


Mínimo e Máximo

- É simples encontrar os valores mínimo e máximo de uma árvore binária de busca.
- Dada a organização da árvore, o valor mínimo será a folha mais à esquerda e o valor máximo será a folha mais à direita.

```
ÁRVORE_MÍNIMO (x)
while x->esq ≠ NULL
  x = x->esq
return x

ÁRVORE_MAXIMO (x)
while x->dir ≠ NULL
  x = x->dir
return x
```



- No pior caso teremos que percorrer a altura da árvore, portanto, as funções ÁRVORE_MÍNIMO e ÁRVORE_MAXIMO executam em tempo O(h).

Inserção

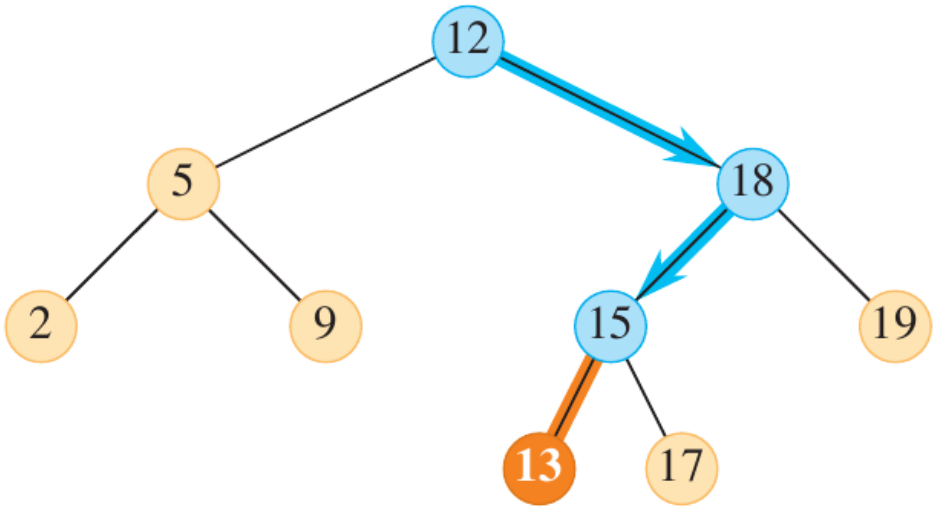
- A inserção de um novo elemento deve obedecer a propriedade da árvore binária. Portanto, devemos encontrar o lugar correto para que o novo item seja inserido.
- Considerando a estrutura da árvore, a inserção de um novo elemento sempre será como filho de uma das folhas.

```
INSERÇÃO_NA_ARVORE (A, i)
  x = A->raiz
  if x == NULL // Árvore vazia
    A->raiz = i
  else
    no_pai = NULL
    while x ≠ NULL
      no_pai = x
      if i->chave < x->chave
        x = x->esq
      else
        x = x->dir
    if i->chave < no_pai->chave
      no_pai->esq = i
    else
      no_pai->dir = i
    i->pai = no_pai
```

- Onde A representa a árvore e i é o nó sendo inserido. Considere que o nó inserido já está criado corretamente, com os ponteiros contendo valores NULL.

Inserção

```
INSERÇÃO_NA_ARVORE (A, i)
  x = A->raiz
  if x == NULL // Árvore vazia
    A->raiz = i
  else
    no_pai = NULL
    while x ≠ NULL
      no_pai = x
      if i->chave < x->chave
        x = x->esq
      else
        x = x->dir
    if i->chave < no_pai->chave
      no_pai->esq = i
    else
      no_pai->dir = i
    i->pai = no_pai
```



Inserção

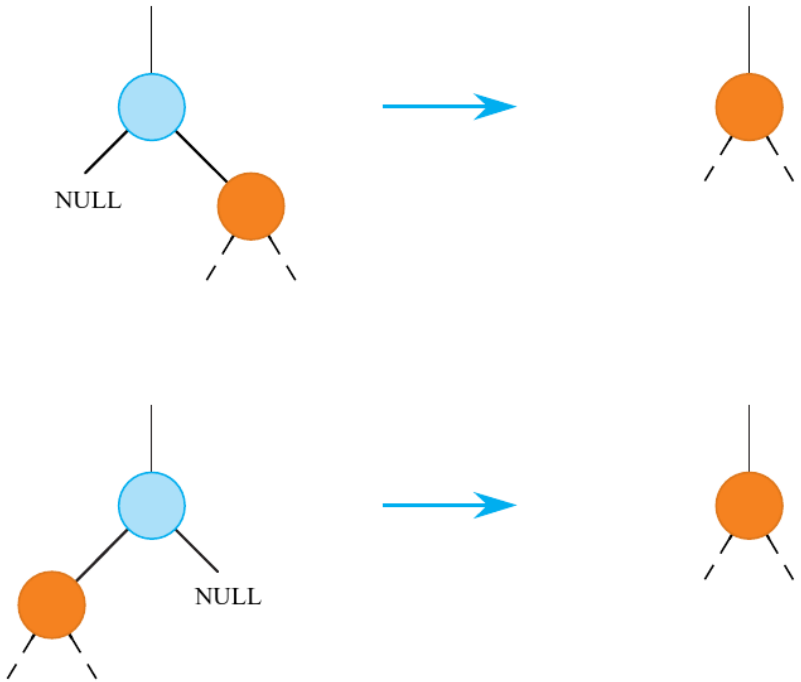
- Como no caso da busca, o tempo de execução da função de inserção é O(h).
- Afinal, no pior caso teremos que percorrer, efetuando comparações, por toda a altura da árvore até inserir o novo nó.

Deleção

- A operação de deleção de um nó na árvore é um pouco mais complicada do que as operações anteriores.
- Precisamos garantir que a propriedade de Árvore Binária de Busca seja mantida após a remoção do nó, o que exigirá alguma manipulação dos elementos da árvore.
- Vamos considerar alguns casos básicos e tratar cada um deles separadamente.

Deleção

- Caso 1: Se o nó não possuir filhos, basta deletar o nó.
- Caso 2: Se o nó possuir apenas um filho, este tomará o lugar do nó deletado.

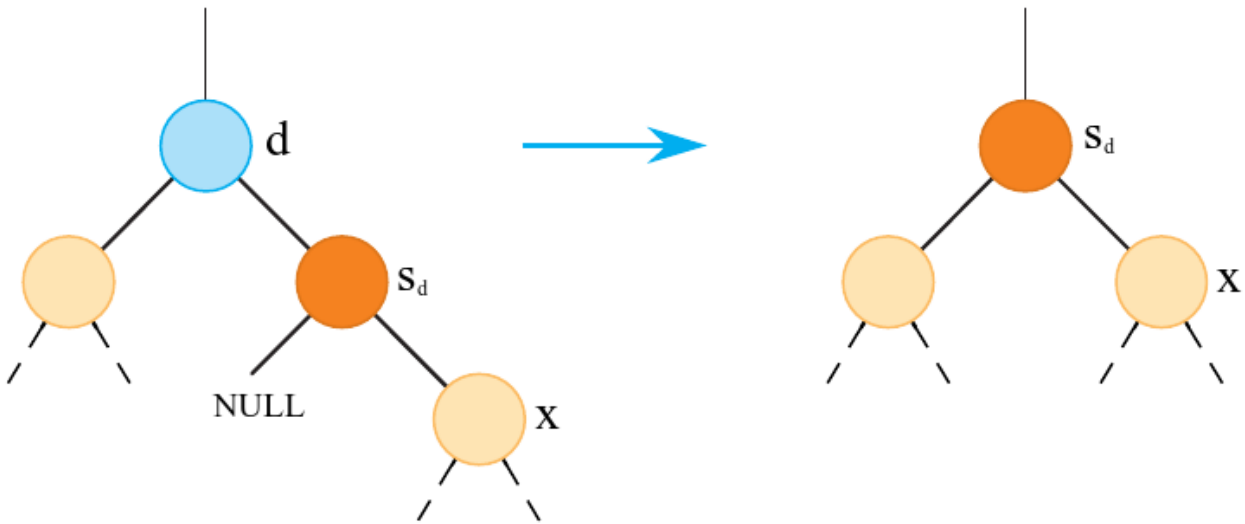


Deleção

- Caso 3: Se o nó possuir dois filhos, encontraremos o sucessor do nó a ser deletado, que chamaremos de $\mathtt{d}$.
- Encontraremos o sucessor de $\mathtt{d}$, que chamaremos de $\mathtt{S_d}$, que está na subárvore direita de $\mathtt{d}$. Este não possuirá filho à esquerda (por quê? Resolva o exercício 5).
- Remova o nó $\mathtt{S_d}$ de sua posição atual e substitua $\mathtt{d}$ por $\mathtt{S_d}$ na árvore.
- **Como fazer isso dependerá se $\mathtt{S_d}$ for o filho direito de $\mathtt{d}$ ou não.**

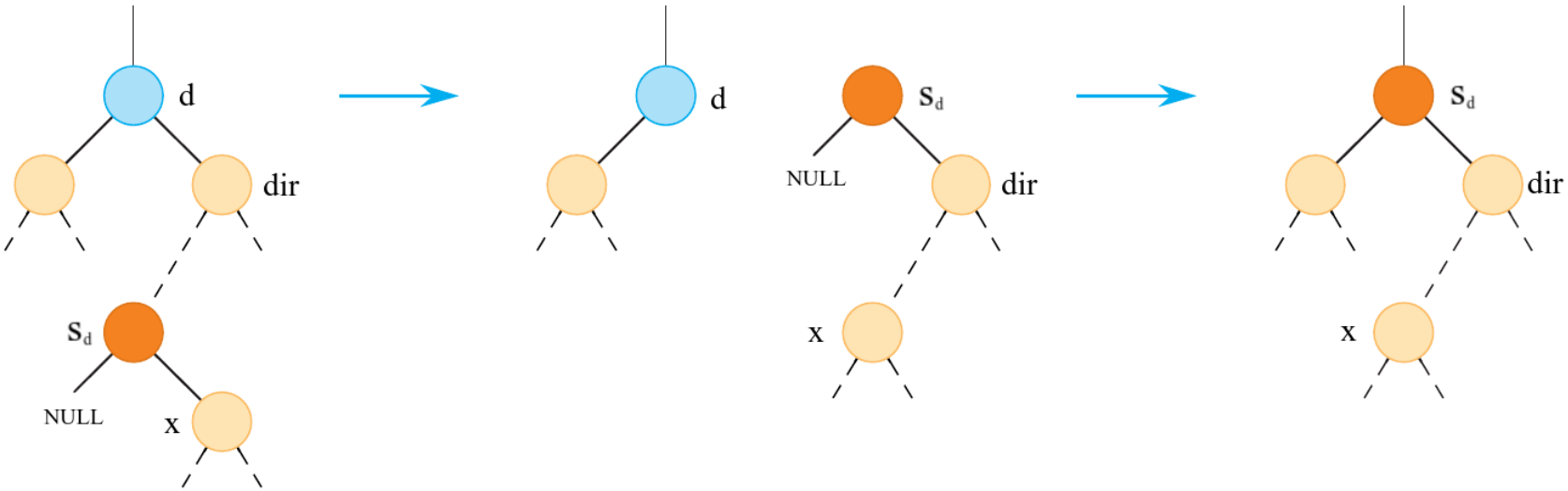
Deleção

- Caso 3.1: Se S_d for o filho direito de d, então, como na ilustrado figura abaixo, substitua d por S_d, deixando o filho direito de S_d inalterado.



Deleção

- Caso 3.2: Caso contrário, S_d está na subárvore direita de d, mas não é o filho direito de d. Nesse caso, como ilustrado na figura abaixo, primeiro substitua S_d por seu próprio filho direito, e então substitua d por S_d.



Deleção

- Para implementarmos essas operações, precisaremos de uma função que mova subárvores.
- A função SUBSTITUI_SUBÁRVORE é uma função muito simples que substitui a subárvore enraizada no nó u pela subárvore enraizada no nó v.

```
SUBSTITUI_SUBARVORE(A, u, v)
  if u->pai == NULL      // É a raiz
      A->raiz = v
  else
    if u == u->pai->esq // u é o filho esquerdo
      u->pai->esq = v
    else
      u->pai->dir = v
  if v ≠ NULL
      v->pai = u->pai
```

Deleção

- Segue a função completa para efetuar a deleção de um nó em uma Árvore Binária de Busca.

```
DELEÇÃO_NA_ARVORE (A, d)
if d->esq == NULL
    SUBSTITUI_SUBÁRVORE(A, d, d->dir)    // Substituir por seu filho à direita
else
  if d->dir == NULL
    SUBSTITUI_SUBÁRVORE(A, d, d->esq)    // Substituir por seu filho à esquerda
  else
    s = ÁRVORE_MÍNIMO(d->dir)            // s é o sucessor
    if s ≠ d->dir                        // s não é o filho direito DIRETO de d
      SUBSTITUI_SUBÁRVORE(A, s, s->dir) // Substituir s por seu filho à direita
      s->dir = d->dir                    // O filho à direita de d se torna
      s->dir->pai = s                    // o filho à direita de s
    SUBSTITUI_SUBÁRVORE(A, d, s)         // Substituir por seu sucessor s
    s->esq = d->esq                      // e dê o filho à esquerda de d para s
    s->esq->pai = s
```

Deleção

- Cada linha de código da função DELEÇÃO_NA_ARVORE executa em tempo constante, excetuando pela chamada da função ÁRVORE_MÍNIMO, que excecuta em tempo O(h).
- Portanto, todas as funções básicas que apresentamos para Árvores Binárias de Busca executam em tempo O(h).

Exercícios

1. Implemente a Árvore Binária de Busca e suas operações básicas em C/C++.
2. Implemente uma função para encontrar o nó sucessor dada uma chave `k`. O sucessor, dada uma chave `k` é o nó com a menor chave maior que `k`. Também implemente uma função para encontrar o nó predecessor, este sendo o nó com a maior chave menor que `k`.
3. Implemente a função de Busca recursivamente.
4. Implemente as funções de encontrar mínimo e máximo recursivamente.
5. Implemente a função de Inserção recursivamente.
6. Está correto afirmar que se um nó em uma Árvore Binária de Busca tem dois filhos, então seu sucessor não possui filho à esquerda e seu predecessor não possui filho à direita. Por quê?
7. Podemos criar um algoritmo de ordenação utilizando a Árvore Binária de Busca. Basta inserir cada elemento na árvore usando a função de Inserção e então executar a função de Percurso em Ordem para imprimir os valores de maneira ordenada. Qual o pior caso deste algoritmo? Qual o melhor caso? Demonstre os resultados.