AULA 9

Arvores binárias

Fonte: https://www.tumblr.com/

Referências: Árvores binárias de busca (PF); Binary Search Trees (S&W); slides (S&W)

Mais tabela de símbolos

Uma **tabela de símbolos** (= *symbol table* = *dictionary*) é um conjunto de objetos (itens), cada um dotado de uma chave (= `key`) e de um valor (= `val`).

As chaves podem ser números inteiros ou *strings* ou outro tipo de dados.

Uma tabela de símbolos está sujeito a dois tipos de operações, entre possíveis outras:

- inserção (= `put()`): consiste em introduzir um objeto na tabela;
- busca (= `get()`): consiste em encontrar um elemento que tenha uma dada chave.

Problema

Problema: Organizar uma tabela de símbolos de maneira que as operações de inserção e busca sejam *razoavelmente eficientes*.

Em geral, uma organização que permite inserções rápidas impede buscas rápidas e vice-versa.

Já vimos como organizar tabelas de símbols através de vetores e listas encadeadas e skip lists.

Hoje: mais uma maneira de organizar uma tabela de símbolos.

Árvore binárias

Uma **árvore binária** (= *binary tree*) é um conjunto de nós/células que satisfaz certas condições.

Cada nó tera três campos:

```
public class BT {
   Node r;
   private class Node {
       private Item item; // conteudo
       private Node left, right;
       public Node(Item item) {
           this.item = item;
       }
   }
```

Pais e filhos

Os campos `left` e `right` dão estrutura à árvore.

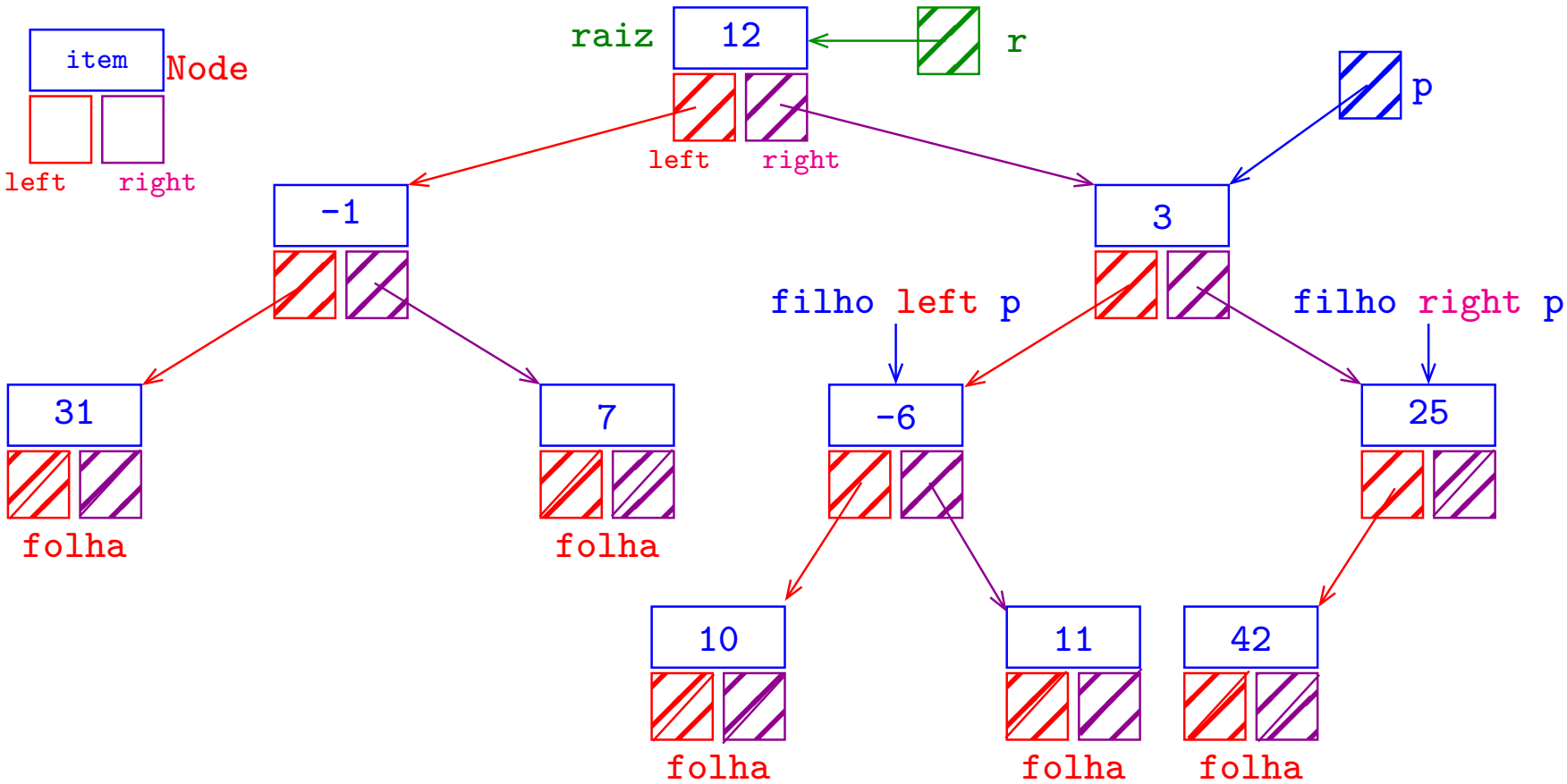
Se `x.left == y`, `y` é o **filho esquerdo** de `x`.

Se `x.right == y`, `y` é o **filho direito** de `x`.

Assim, `x` é o **pai** de `y` se `x.left == y` ou `x.right == y`.

Um **folha** é um nó sem filhos.
Ou seja, se `x.left == null` e `x.right == null` então `x` é um folha

Ilustração de uma árvore binária



Árvores e subárvores

Suponha que `r` e `p` são nós.
`p` é **descendente** de `r` se `p` pode ser alcançada pela iteração dos comandos

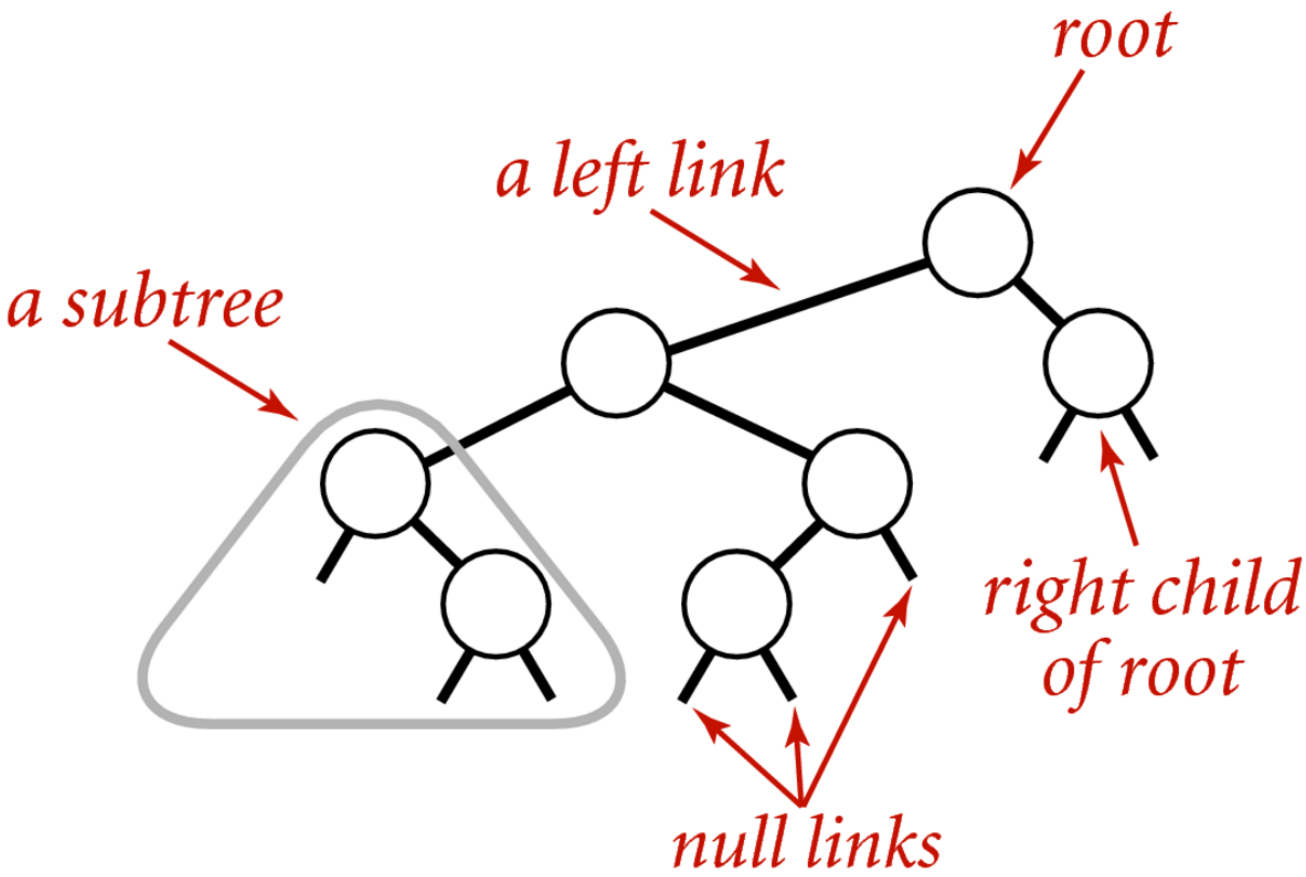
```
p = p.left;        p = p.right;
```

em qualquer ordem.

Um nó `r` juntamente com todos os seus descendentes é uma **árvore binária** e `r` é dito a **raiz** (=*root*) da árvore.

Para qualquer nó `p`, `p.left` é a raiz da subárvore esquerda de `p` e `p.right` é a raiz da subárvore direita de `p`.

Anatomia de uma árvore binária



Anatomy of a binary tree

Fonte: `algs4`

Endereço de uma árvore

O endereço de uma árvore binária é o endereço de sua raiz.

```
Node r;
```

Um objeto `r` é uma árvore binária se

- `r == null` ou
- `r->left` e `r->right` são árvores binárias.

Maneiras de varrer uma árvore

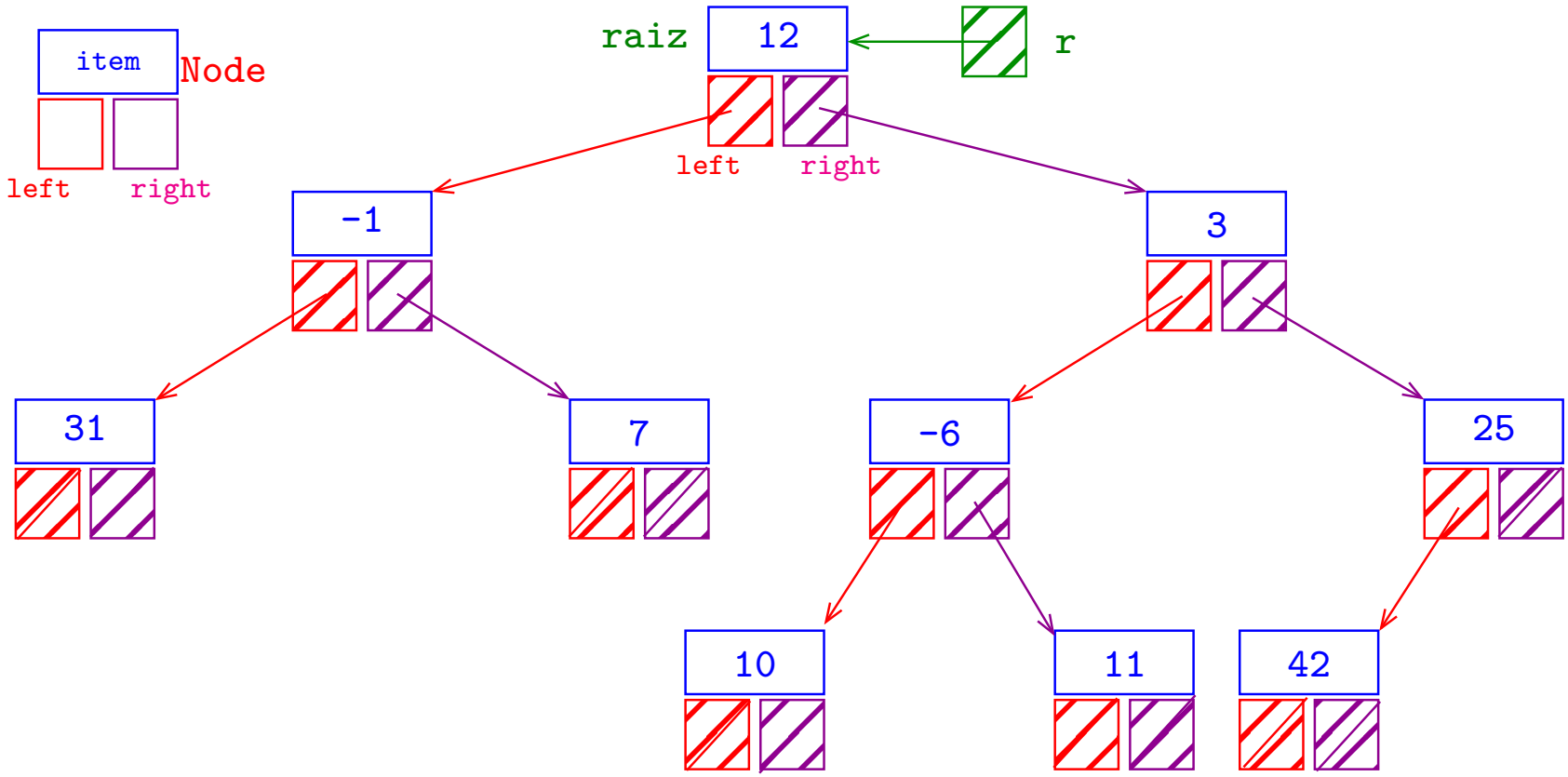
Existem várias maneiras de percorrermos uma árvore binária. Talvez as mais tradicionais sejam:

- *inorder traversal*: esquerda-raiz-direita (e-r-d);
- *preorder traversal*: raiz-esquerda-direita (r-e-d);
- *posorder traversal*: esquerda-direita-raiz (e-d-r);

`preOrdem()`, `inOrdem()` e `posOrdem()` retornam todos os itens da árvore como `Iterable`. Para iterar sobre todos os itens de uma `BT` de nome `st` basta fazermos

```
for (Item item:  st.preOrdem()) {
    StdOut.println(item);
}
```

Ilustração de percursos em árvores binárias

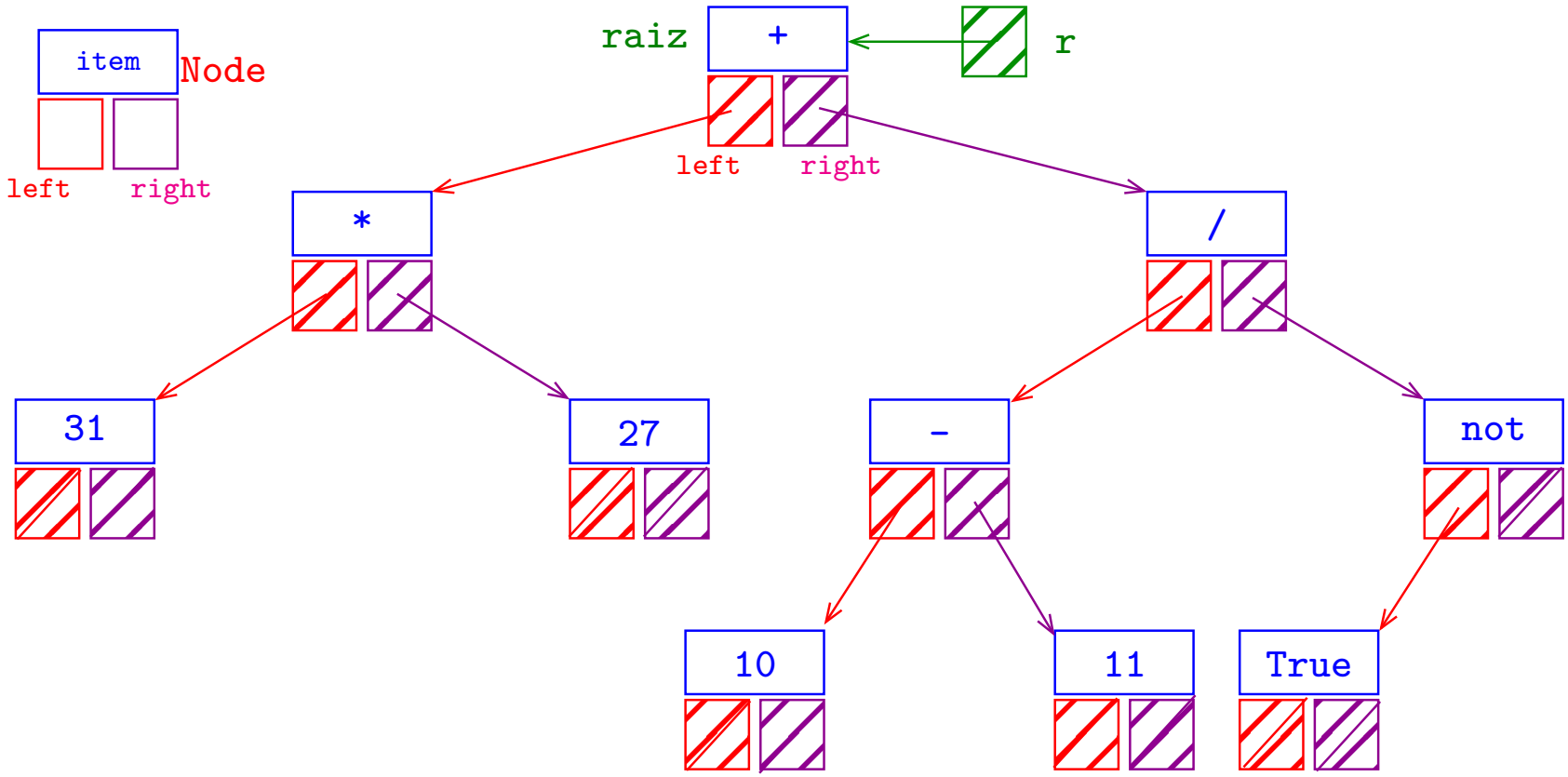


in-ordem (e-r-d): 31 -1 7 12 10 -6 11 3 42 25

pré-ordem (r-e-d): 12 -1 31 7 3 -6 10 11 25 42

pós-ordem (e-d-r): 31 7 -1 10 11 -6 42 25 3 12

Ilustração de percursos em árvores binárias



in-ordem (e-r-d): `31 * 27 + 10 - 11 / True not`

pré-ordem (r-e-d): `+ * 31 27 / - 10 11 not True`

pós-ordem (e-d-r): `31 27 * 10 11 - True not / +`

esquerda-raiz-direita

Visitamos

1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. depois a subárvore direita da raiz, em ordem e-r-d;

```
public Iterable<Item> inOrdem() {
   Queue<Item> queue = new Queue<Item>();
   inOrdem(r, queue)
   return queue;
}
```

esquerda-raiz-direita

Visitamos

1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. depois a subárvore direita da raiz, em ordem e-r-d;

```
private void inOrdem(Node r,
                     Queue<Item> queue) {
   if (r != null) {
       inOrdem(r.left, queue);
       queue.enqueue(r.item);
       inOrdem(r.right,queue);
   }
}
```

esquerda-raiz-direita versão iterativa

```
private void inOrdem(Node r,
                     Queue<Item> queue) {
   Stack<Node> s = new Stack<Node>();
   while (r != null || !s.isEmpty()) {
       if (r != null) {
           s.push(r); r = r.left;
       }
       else {
           r = s.pop();
           queue.enqueue(r.item);
           r = r.right;
       }
   }
}
```

raiz-esquerda-direita

Visitamos

1. a raiz;
2. depois a subárvore esquerda da raiz, em ordem r-e-d;
3. depois a subárvore direita da raiz, em ordem r-e-d;

```
public Iterable<Item> preOrdem() {
   Queue<Item> queue = new Queue<Item>();
   preOrdem(r, queue)
   return queue;
}
```

raiz-esquerda-direita

Visitamos

1. a raiz;
2. depois a subárvore esquerda da raiz, em ordem r-e-d;
3. depois a subárvore direita da raiz, em ordem r-e-d;

```
private void preOrdem(Node r,
                      Queue<Item> queue) {
   if (r != null) {
       queue.enqueue(r.item);
       preOrdem(r.left, queue);
       preOrdem(r.right,queue);
   }
}
```

esquerda-direita-raiz

Visitamos

1. a subárvore esquerda da raiz, em ordem e-d-r;
2. depois a subárvore direita da raiz, em ordem e-d-r;
3. depois a raiz;

```
public Iterable<Item> preOrdem() {
   Queue<Item> queue = new Queue<Item>();
   posOrdem(r, queue)
   return queue;
}
```

esquerda-direita-raiz

Visitamos

1. a subárvore esquerda da raiz, em ordem e-d-r;
2. depois a subárvore direita da raiz, em ordem e-d-r;
3. depois a raiz;

```
private void posOrdem(Node r,
                      Queue<Item> queue) {
   if (r != null) {
       posOrdem(r.left, queue);
       posOrdem(r.right,queue);
       queue.enqueue(r.item);
   }
}
```

Primeiro nó esquerda-raiz-direita

Recebe a raiz r de uma árvore binária não vazia e retorna o primeiro nó na ordem e-r-d

```
private Node primeiro(Node r)
{
   while (r.left != null)
       r = r.left;
   return r;
}
```

Altura

A **profundidade** (=*depth*) de um nó de uma BT é o número de links no caminho que vai da raiz até o nó.

A **altura** (=*height*) de uma BT é o máximo das profundidades dos nós, ou seja, a profundidade do nó mais profundo.

```
private int altura(Node r) {
   if (r == null) return -1;
   int hLeft = altura(r.left);
   int hRight = altura(r.right);
   return Math.max(hLeft, hRight) + 1
}
```

Árvores balanceadas

A altura de uma árvore com n nós é um número entre $\lg(n)$ e n.

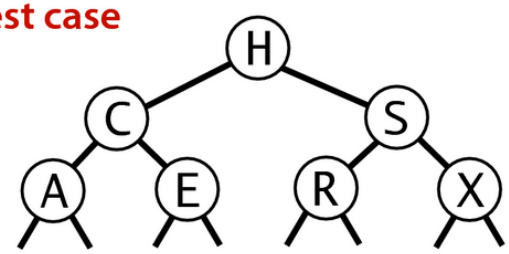
Uma árvore binária é **balanceada** (ou **equilibrada**) se, em cada um de seus nós, as subárvores esquerda e direita tiverem *aproximadamente* a mesma altura.

Árvores balanceadas têm altura *próxima* de $\lg(n)$.

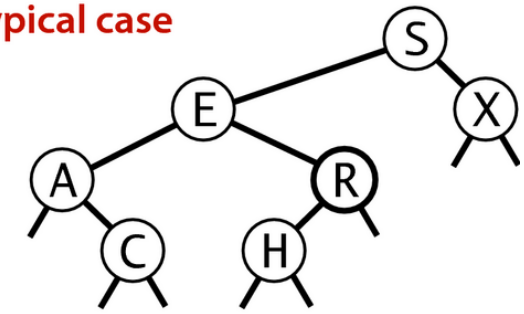
O consumo de tempo dos algoritmos que manipulam árvores binárias dependem frequentemente da altura da árvore.

Exemplos

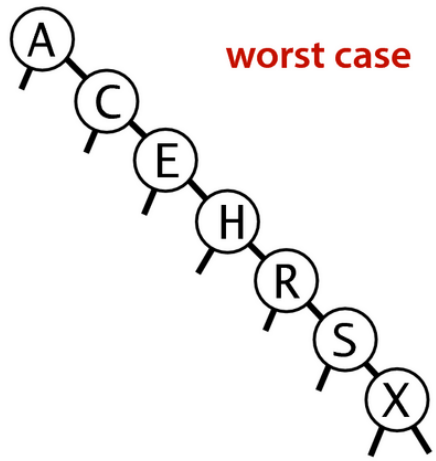
best case



typical case



worst case

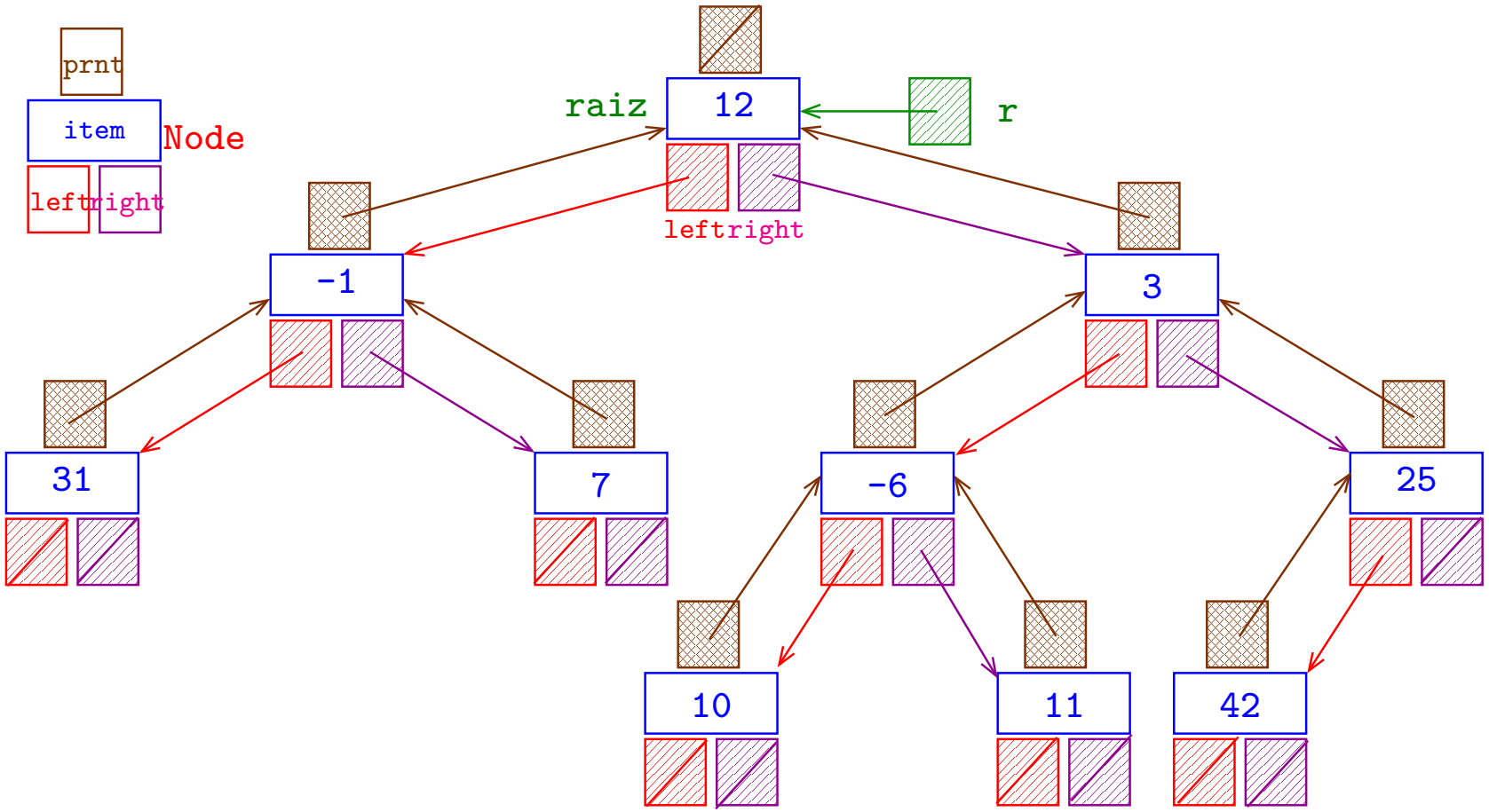


Nós com campo pai

Em algumas aplicações é conveniente ter acesso imediato ao pai de qualquer nó.

```
private class Node {
   private Item item; // conteudo
   private Node left, prnt, right;
   public Node(Item item) {
       this.item = item;
   }
}
```

Ilustração de nós com campo pai



Sucessor e predecessor

Recebe um nó p de uma árvore binária não vazia e retorna o seu **sucessor** na ordem e-r-d.

```
private Node sucessor(Node p) {
   if (p.right != NULL) {
       Node q = p.right;
       while (q.left != null) q = q.left;
       return q;
   }
   while (p.prnt!=null && p.prnt.right==p)
       p = p.prnt;
   return p.prnt;
}
```

Exercício: função que retorna o **predecessor**.

Comprimento interno

O **comprimento interno** (=*internal path length*) de uma `BT` é a soma das profundidades dos seus nós, ou seja, a soma dos comprimentos de todos os caminhos que levam da raiz até um nó.

Esse conceito é usado para estimar o desempenho esperado de STs implementadas com BSTs