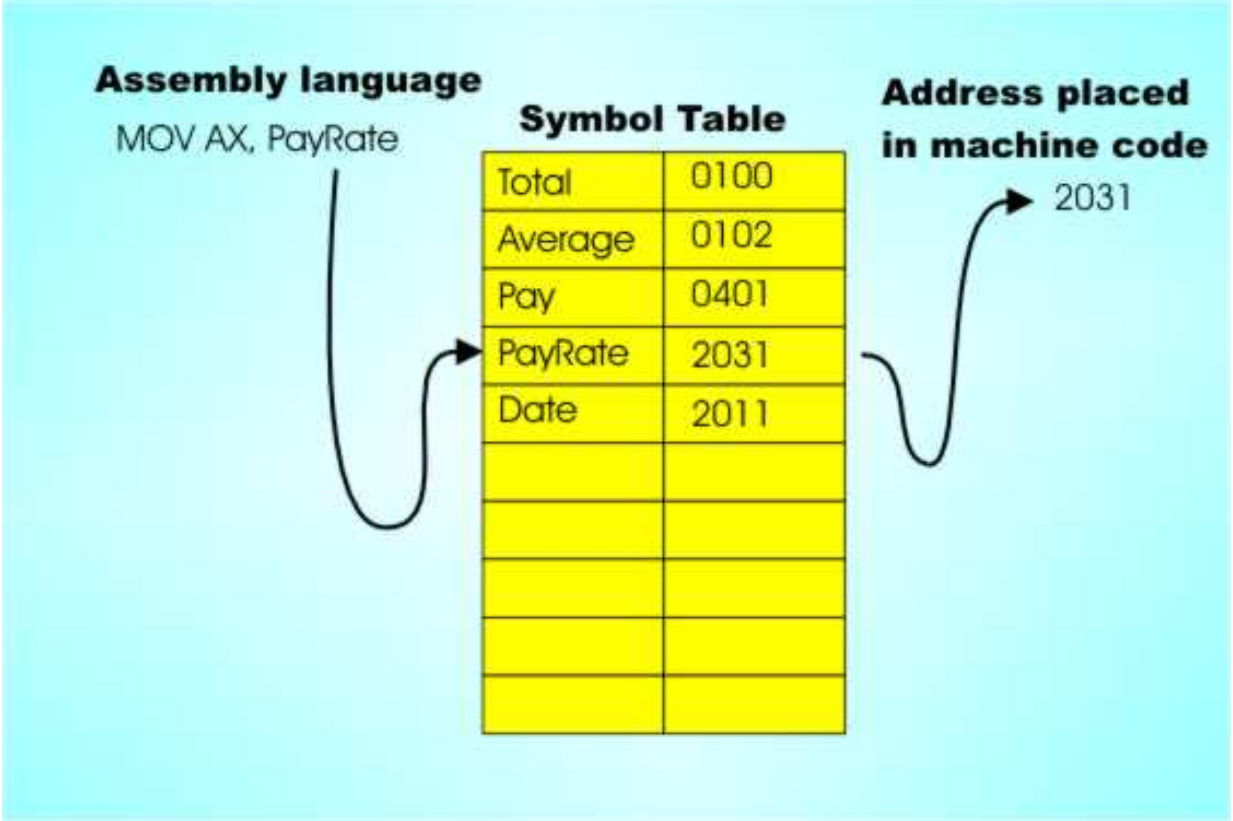
MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

Nos episódios anteriores . . .

Tabelas de Símbolos



Fonte: http://www.i-programmer.info/

Tabelas de símbolos (PF)
Elementary Symbol Tables (S&W)

Tabelas de símbolos

Uma **tabela de símbolos** (`ST` = *symbol table*) é um ADT que consiste em um conjunto de itens, sendo cada item um par `chave`-`valor` ou `key`-`value`, munido de duas operações fundamentais:

- `put()`, que insere um novo item na `ST`, e
- `get()`, que busca o valor associado a uma dada chave.

Tabelas de símbolos

Convenções sobre `STs`:

- não há chaves repetidas
  (as chaves são duas a duas distintas),
- `NULL` nunca é usado como `key`,
- `NULL` nunca é usado como `value`
  associado a uma `key`.

`STs` são também chamadas de
dictionarys, *maps* e *associative arrays*.

Interface ST.h

Arquivo ST.h

`void`	`stInit()`	cria uma ST
`void`	`stPut(Key key, Value val)`	insere (`key`,`val`) na ST
`Value`	`stGet(Key key)`	busca o valor associado a `key`
`void`	`stDelete(Key key)`	remove (`key`,`val`) da ST
`int`	`stRank(Key key)`	no. de keys menor que `key`
`bool`	`stEmpty()`	ST está vazia?
`bool`	`stContains(Key key)`	a `key` está na ST?

Experimentos

Consumo de tempo para se criar um `ST` em que as chaves são as palavras em `les_miserables.txt` e os valores o número de ocorrências.

estrutura	`ST`	tempo
vetor	não-ordenada	59.5
vetor MTF	não-ordenada	7.6
vetor	ordenada	1.5
lista ligada	não-ordenada	147.1
lista ligada MTF	não-ordenada	15.3
lista ligada	ordenada	115.2
skiplist ♡	ordenada	1.1

Tempos em segundos obtidos com `StopWatch`.

AULA 11

Árvores 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 símbolos através de vetores, 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 terá três campos:

```
typedef struct node *Node;

struct node {
    Item item;
    Node left, right;
}
Node newNode(Item item, Node left, Node right) {
    Node p = mallocSafe(sizeof(*p));
    p->item = item;
    p->left = left;    p->right = right;
    return p;
}
```

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`.

Folhas

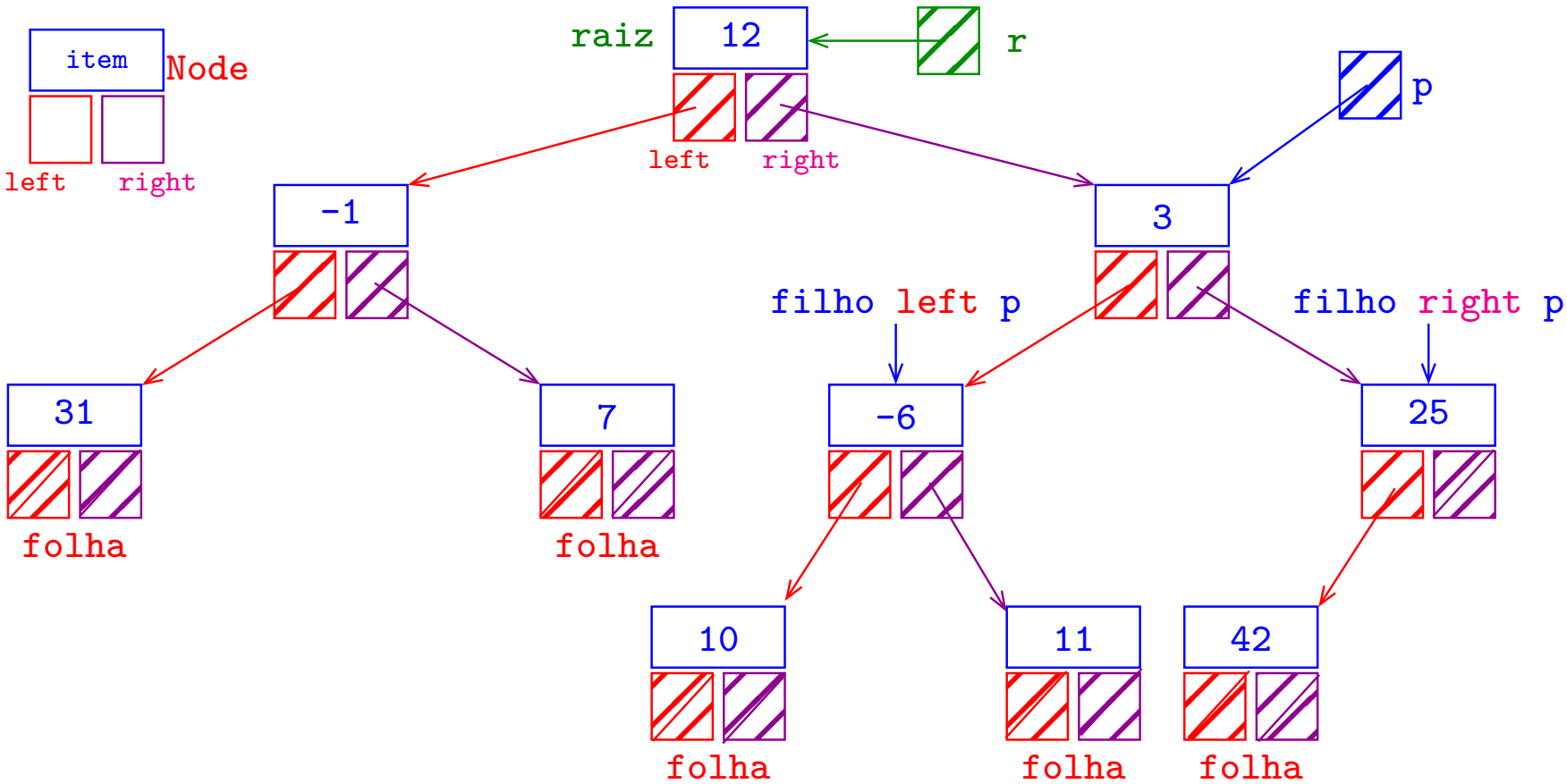
Uma **folha** é um nó sem filhos.

Ou seja,

se `x->left == NULL` e `x->right == NULL`
então `x` é uma folha.

```
bool isLeaf(Node x) {
   return x->left == NULL
          && x->right == NULL;
}
```

Ilustração de uma árvore binária



Árvores e subárvores

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

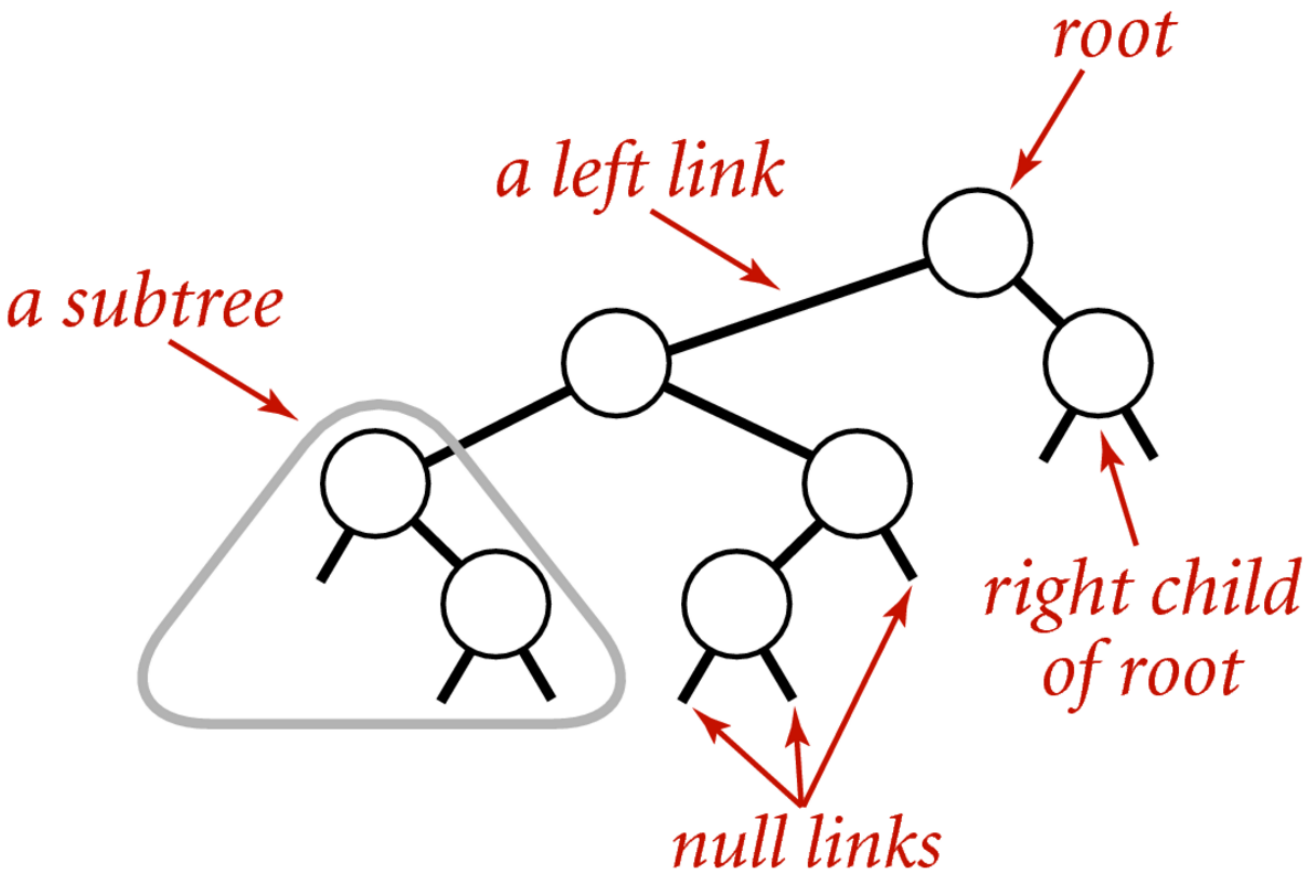
```
p = p->left;        p = p->right;
```

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.

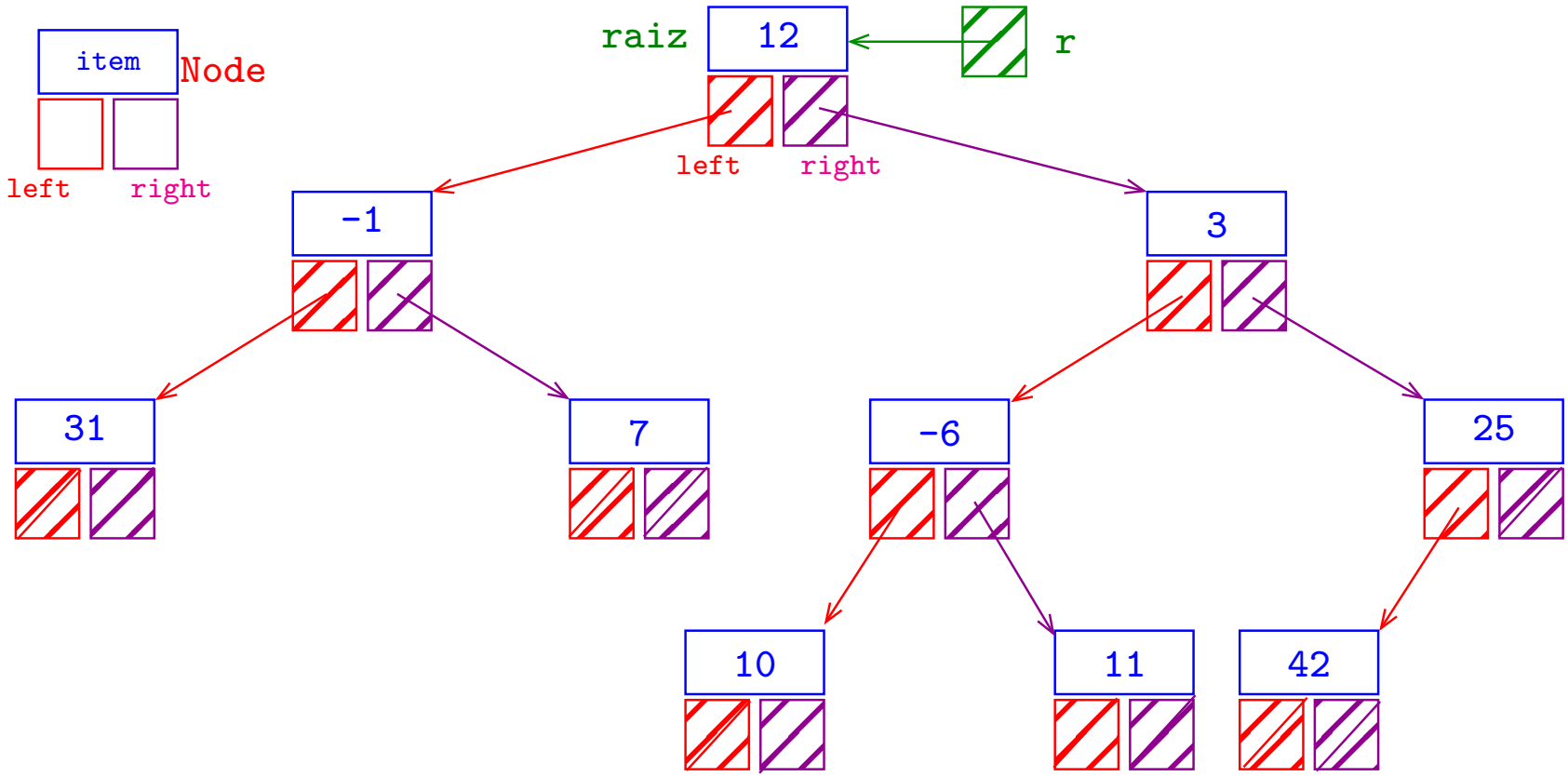
Maneiras de varrer uma árvore

Existem várias maneiras de
percorrermos uma árvore binária.

Talvez as mais tradicionais sejam:

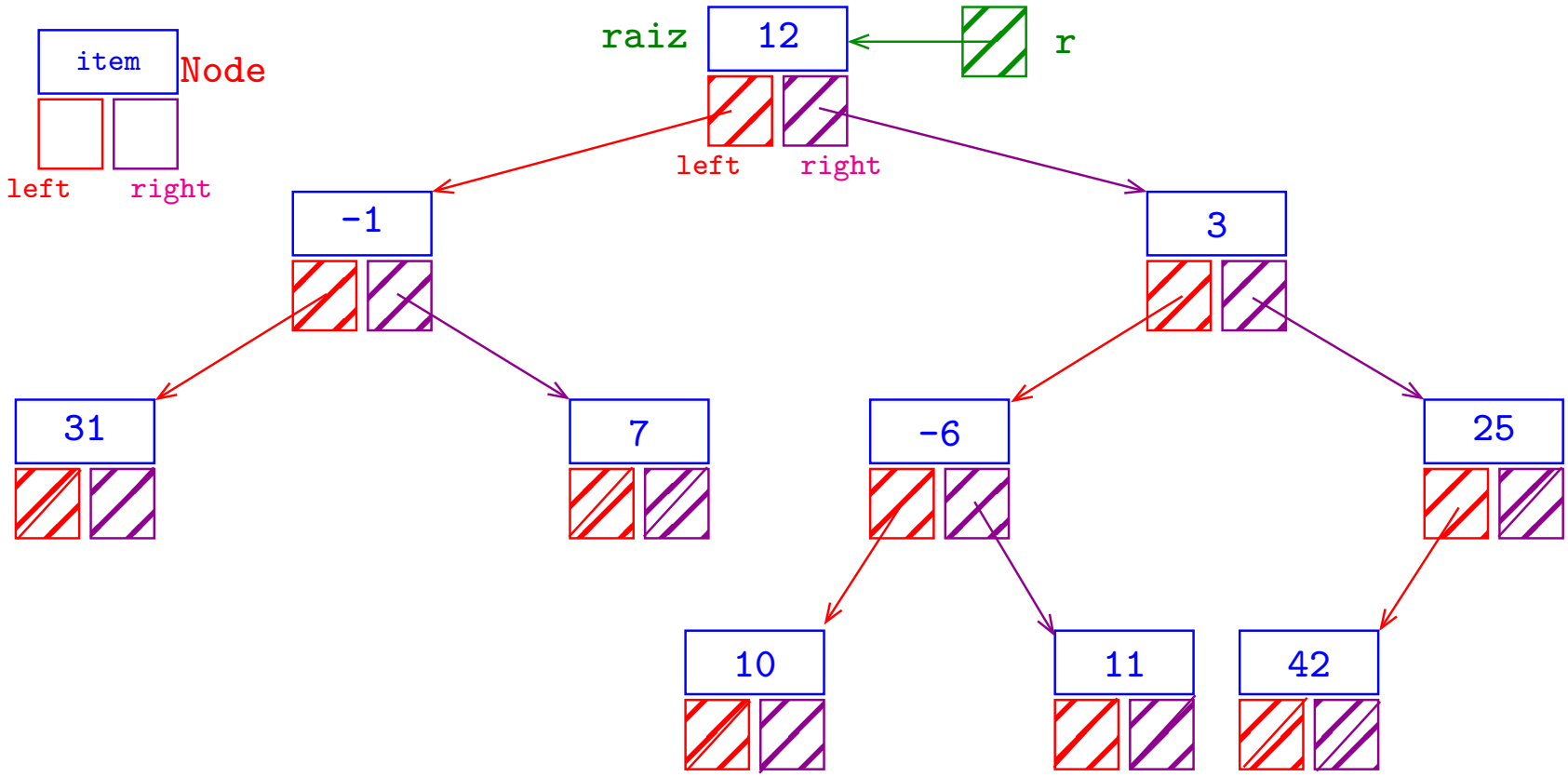
- *inorder traversal*: esquerda-raiz-direita (e-r-d);
- *preorder traversal*: raiz-esquerda-direita (r-e-d);
- *posorder traversal*: esquerda-direita-raiz (e-d-r);

Ilustração de percursos em árvores binárias



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

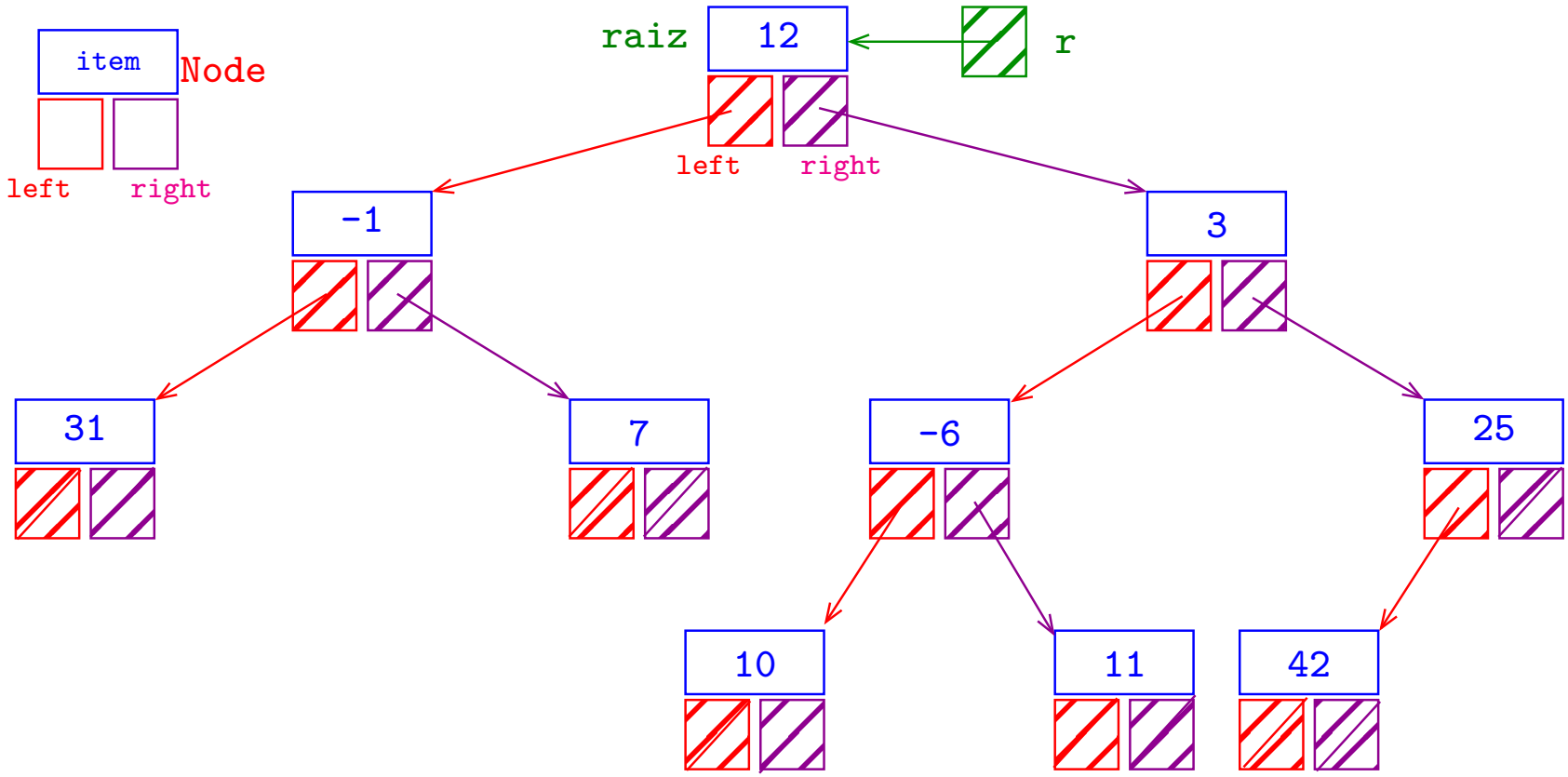
Ilustração de percursos em árvores binárias



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

Ilustração de percursos em árvores binárias

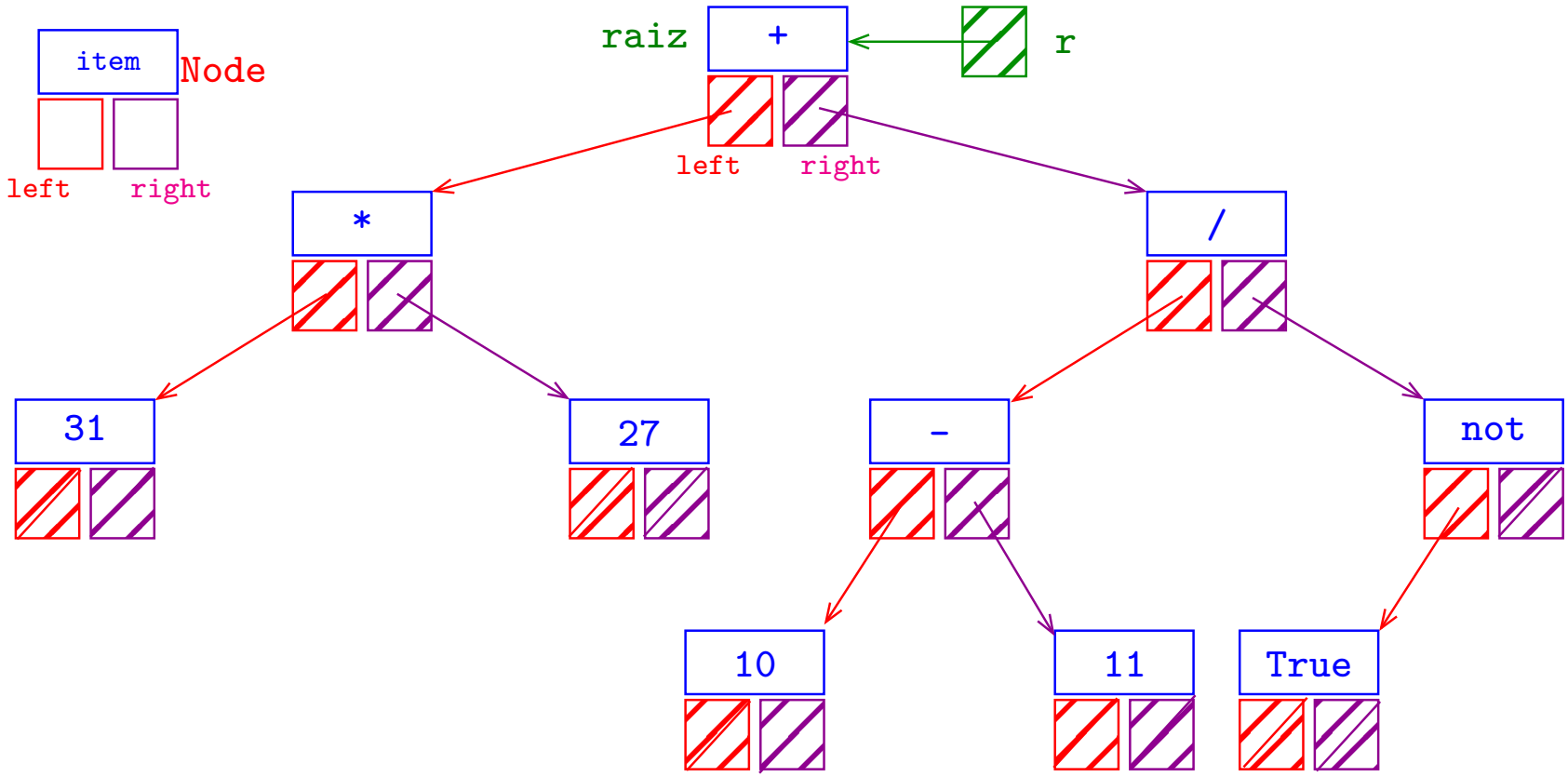


in-ordem (e-r-d): 31 -1 7 12 10 -6 11 3 42 25

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;

```
Queue inOrdem() {
    Queue queue = queueInit();
    inOrdem(r, queue);
    return queue;
}
```

Devolve uma fila com os `Item`s da árvore inOrdem.

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;

```
static void inOrdem(Node r, Queue queue) {
   if (r != NULL) {
      inOrdem(r->left, queue);
      enqueue(queue, r->item);
      inOrdem(r->right,queue);
   }
}
```

esquerda-raiz-direita versão iterativa

```
static void inOrdem(Node r, Queue queue){
   Stack s = stackInit();
   while (r != NULL || !isEmpty(s)) {
      if (r != NULL) {
         stackPush(s, r);
         r = r->left;
      }
      else {
         r = stackPop(s);
         enqueue(queue, 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;

```
Queue preOrdem() {
   Queue queue = queueInit();
   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;

```
static void preOrdem(Node r, Queue queue) {
   if (r != NULL) {
      enqueue(queue, 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;

```
Queue posOrdem() {
    Queue queue = queueInit();
    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;

```
static void posOrdem(Node r, Queue queue) {
   if (r != NULL) {
      posOrdem(r->left, queue);
      posOrdem(r->right,queue);
      enqueue(queue, 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.

```
static 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 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.

```
static int altura(Node r) {
   int hLeft, hRight;
   if (r == NULL) return -1;
   hLeft  = altura(r->left);
   hRight = altura(r->right);
   if (hLeft > hRight) return hLeft + 1;
   else return hRight + 1;
}
```

Árvores balanceadas

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

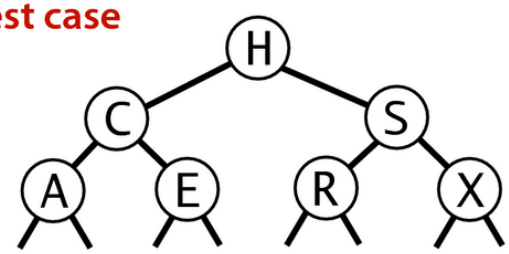
Uma árvore binária é **balanceada** (ou **equilibrada**) se, em cada um de seus nós, as subárvores esquerda e direita tiverem *aproximadamente* a mesma altura.

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

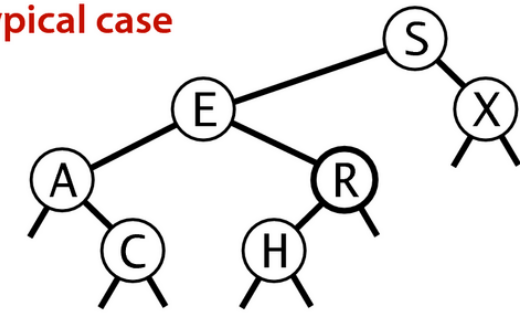
O consumo de tempo dos algoritmos que manipulam árvores binárias dependem frequentemente da altura da árvore.

Exemplos

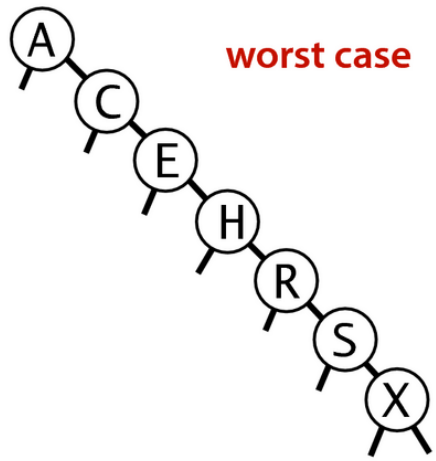
best case



typical case



worst case

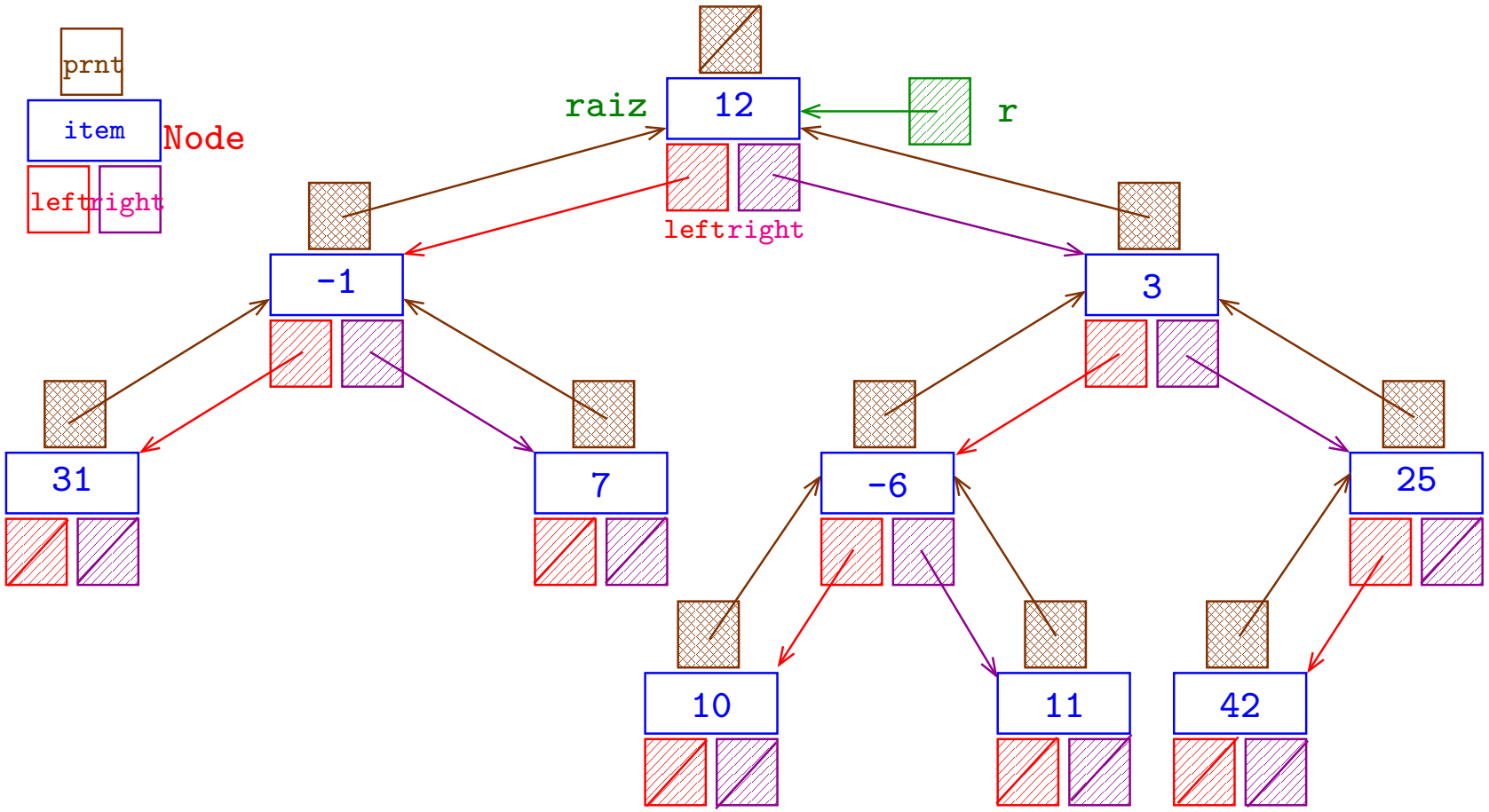


Nós com campo pai

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

```
typedef struct node *Node;

struct node {
   Item item;
   Node prnt, left, right;
}
```

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.

```
static Node sucessor(Node p) {
    Node q;
    if (p->right != NULL) {
        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

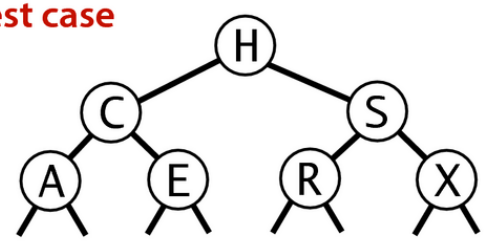
Fique atento

O código a seguir percorre a árvore em pré-ordem e imprime uma sequência de bits que a codifica.

```
static void writeBT(Node x) {
   if (x == NULL) {
      printf("1");
      return;
   }
   printf("0");
   writeBT(x->left);
   writeBT(x->right);
}
```

Fique atento!

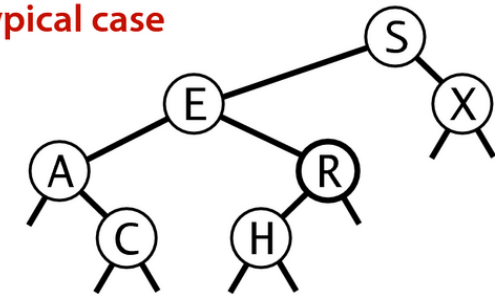
best case



0 0 0 1 1 0 1 1 0 0 1 1 0 1 1

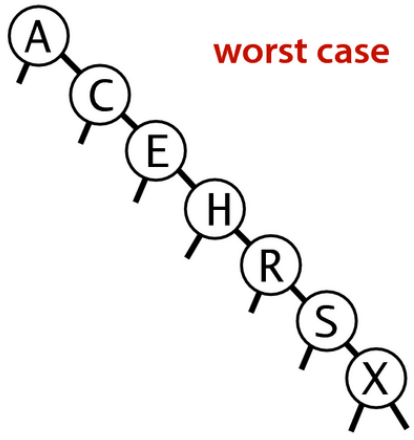
H C A - - E - - S R - - X - -

typical case



0 0 0 1 0 1 1 0 0 1 1 1 0 1 1

S E A - C - - R H - - - X - -



worst case

0 1 0 1 0 1 0 1 0 1 0 1 0 1 1

A - C - E - H - R - S - X - -

BST possibilities

Fonte: algs4

Fique atento

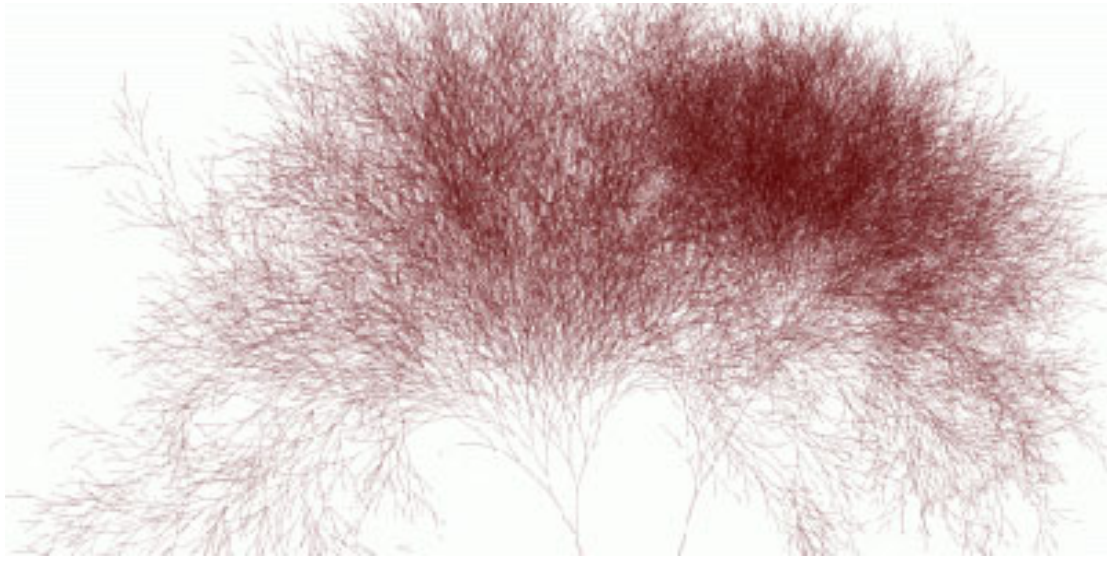
Usaremos essa codificação mais a frente em umas coisas bem bacanas.

```
static Node readBT() {
   Node left, right;
   int b;
   scanf("%d", &b);
   if (b) return NULL;
   left = readBT();
   right = readBT();
   return newNode(NULL, left, right);
}
```

0 0 0 1 0 1 1 0 0 1 1 1 0 1 1

S E A - C - - R H - - - X - -

Árvores binárias de busca



Fonte: http://infosthetics.com/archives/

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

Árvore binárias de busca

Considere uma árvore binária cujos nós têm um campo chave (key como int ou String, por exemplo).

```
typedef struct node *Node;

struct node {
    Key key;
    Value val;
    Node *left, *right;
}

Node newNode(Key key, Value val) {
    Node p = mallocSafe(sizeof(*p));
    p->key = key;       p->val = val;
    p->left = NULL;     p->right = NULL;
    return p;
}
```

Árvore binárias de busca

Uma árvore binária deste tipo é de **busca** (em relação ao campo `key`) se para cada nó `x`: `x->key` é

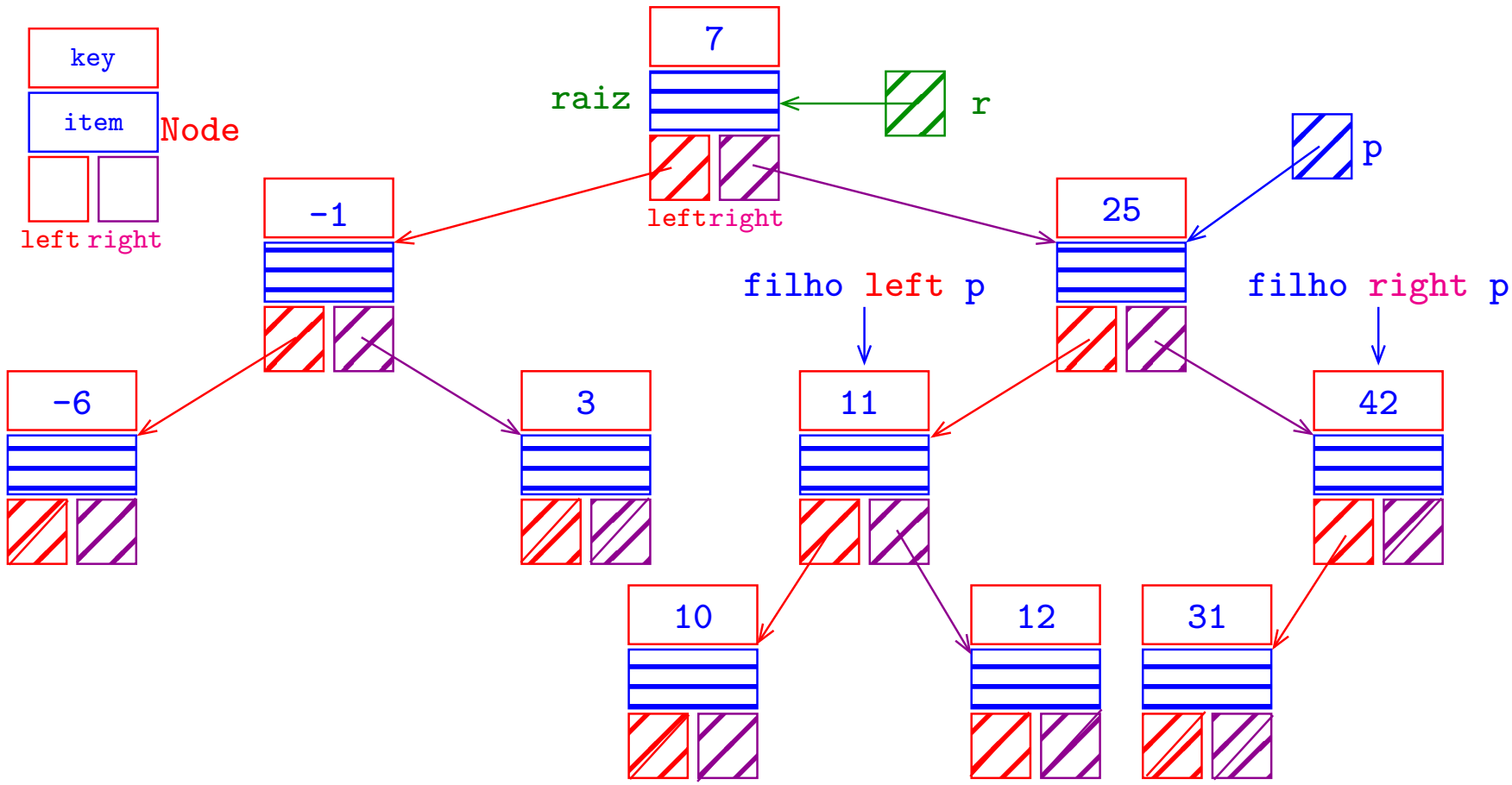
1. maior ou igual à chave de qualquer nó na subárvore esquerda de `x` e
2. menor à chave de qualquer nó na subárvore direita de `x`.

Assim, se `p` é um nó qualquer então vale que

```
compare(q->key, p->key) <= 0 e
compare(p->key, t->key) < 0
```

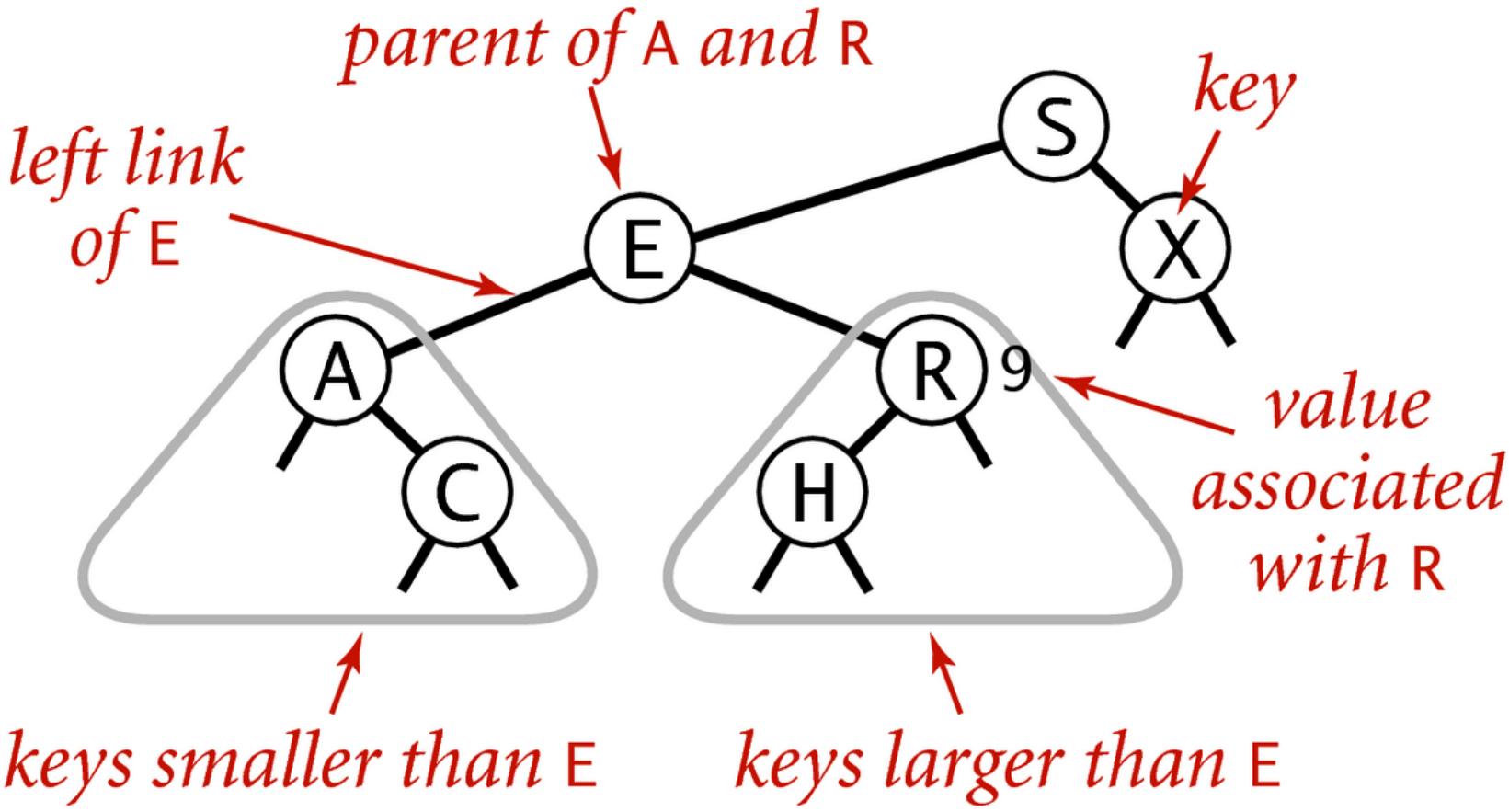
para todo nó `q` na subárvore esquerda de `p` e todo nó `t` na subárvore direita de `p`.

Ilustração de uma árvore binária de busca



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

Anatomia de uma árvore binária de busca



Anatomy of a binary search tree

Fonte: algs4

BST: get(key)

Recebe uma chave key e
retorna o valor val associado à key;
se key não está na BST, retorna NULL.

```
Value get(Key key) {
   Node x = getTree(r, key);
   if (x == null) return NULL;
   return x->val;
}
```

BST: get(key)

Recebe uma chave key e
retorna o valor val associado à key;
se key não está na BST, retorna NULL.

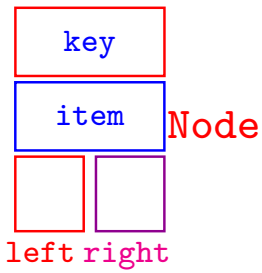
```
static Node getTree(Node x, Key key) {
   /* Considera subárvore que tem raiz x */
   if (x == NULL) return NULL;
   int cmp = compare(key, x->key);
   if (cmp < 0)
      return getTree(x->left, key);
   if (cmp > 0)
      return getTree(x->right, key);
   return x;
}
```

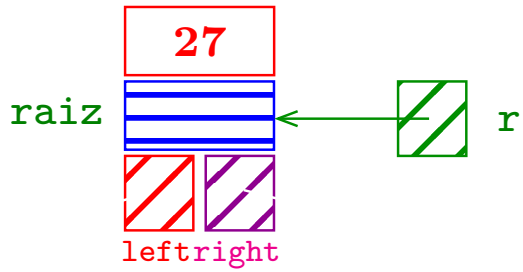
BST: get(key) versão iterativa

Recebe uma chave key e
retorna o valor val associado à key;
se key não está na BST, retorna NULL.

```
Value getTree(Node x, Key key) {
   int cmp;
   while (x != NULL) {
      cmp = compare(key, x->key);
      if (cmp == 0) return x;
      if (cmp < 0) x = x->left;
      else x = x->right;
   }
   return NULL;
}
```

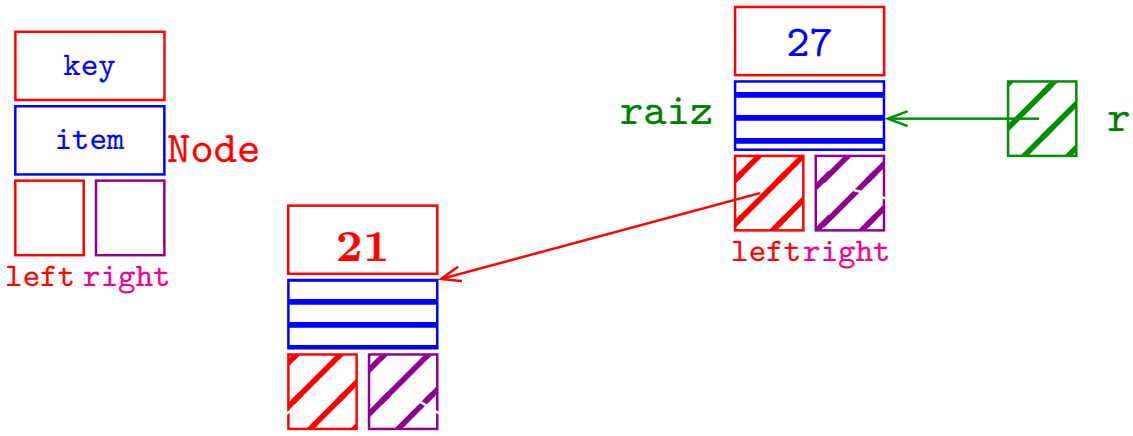
Ilustração de put()





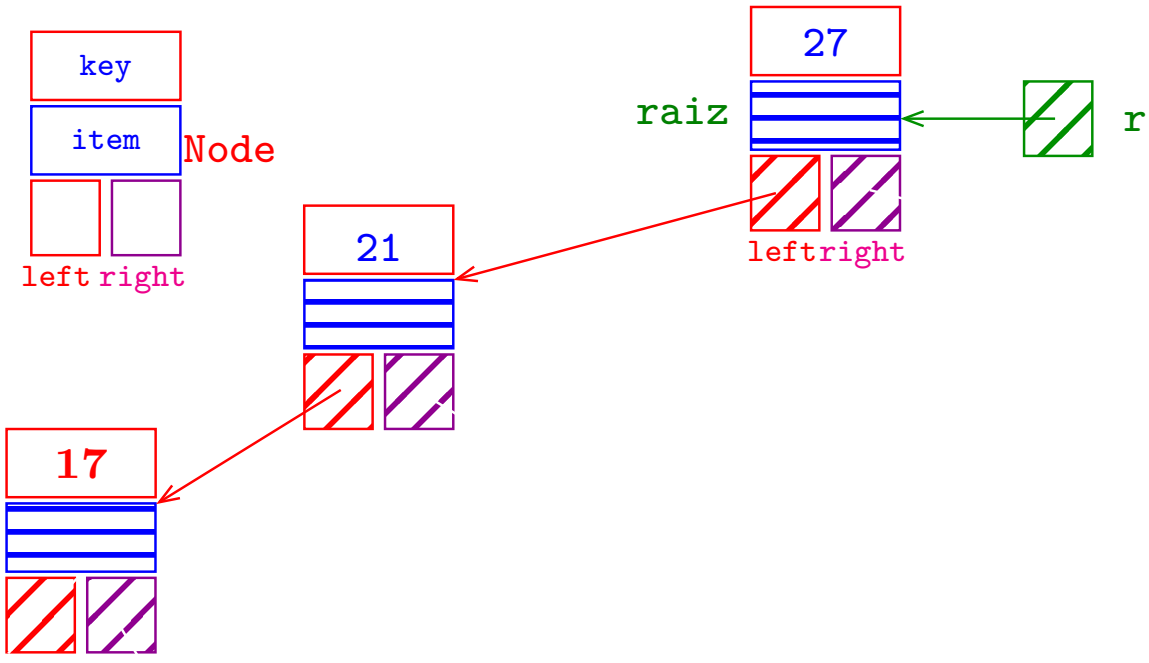
ordem put(): **27**

Ilustração de put()



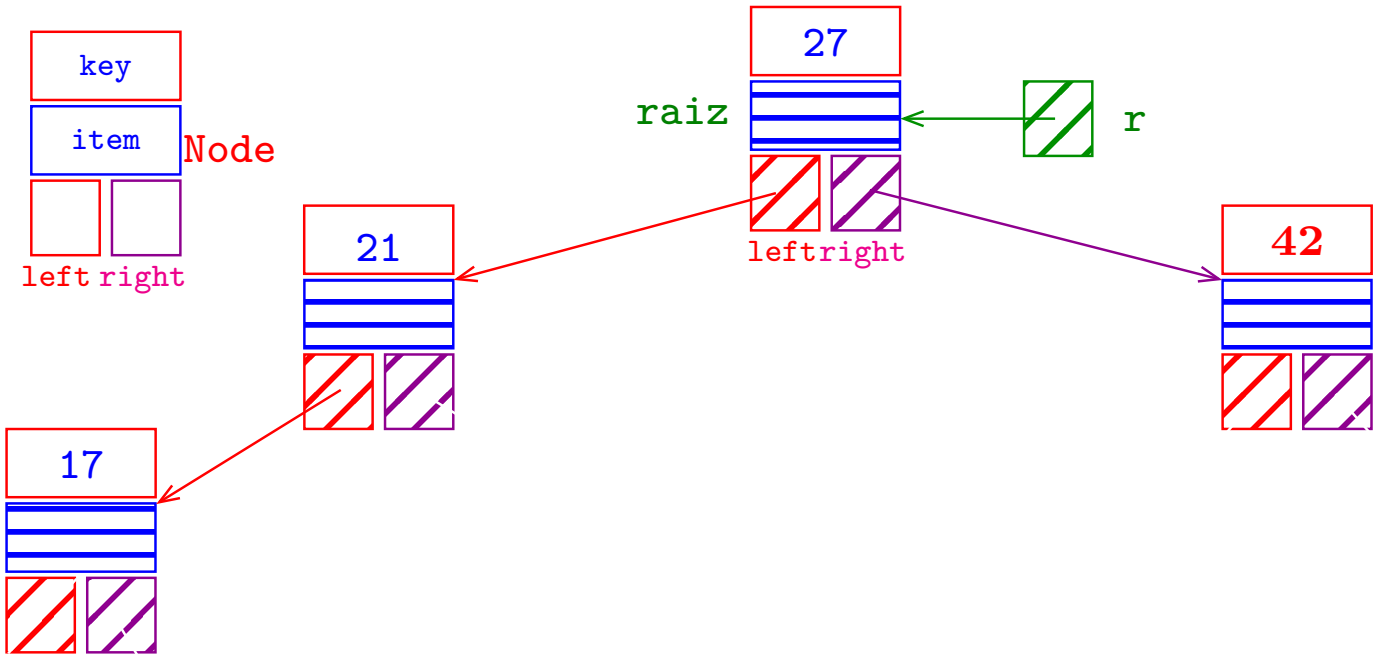
ordem put(): 27 **21**

Ilustração de put()



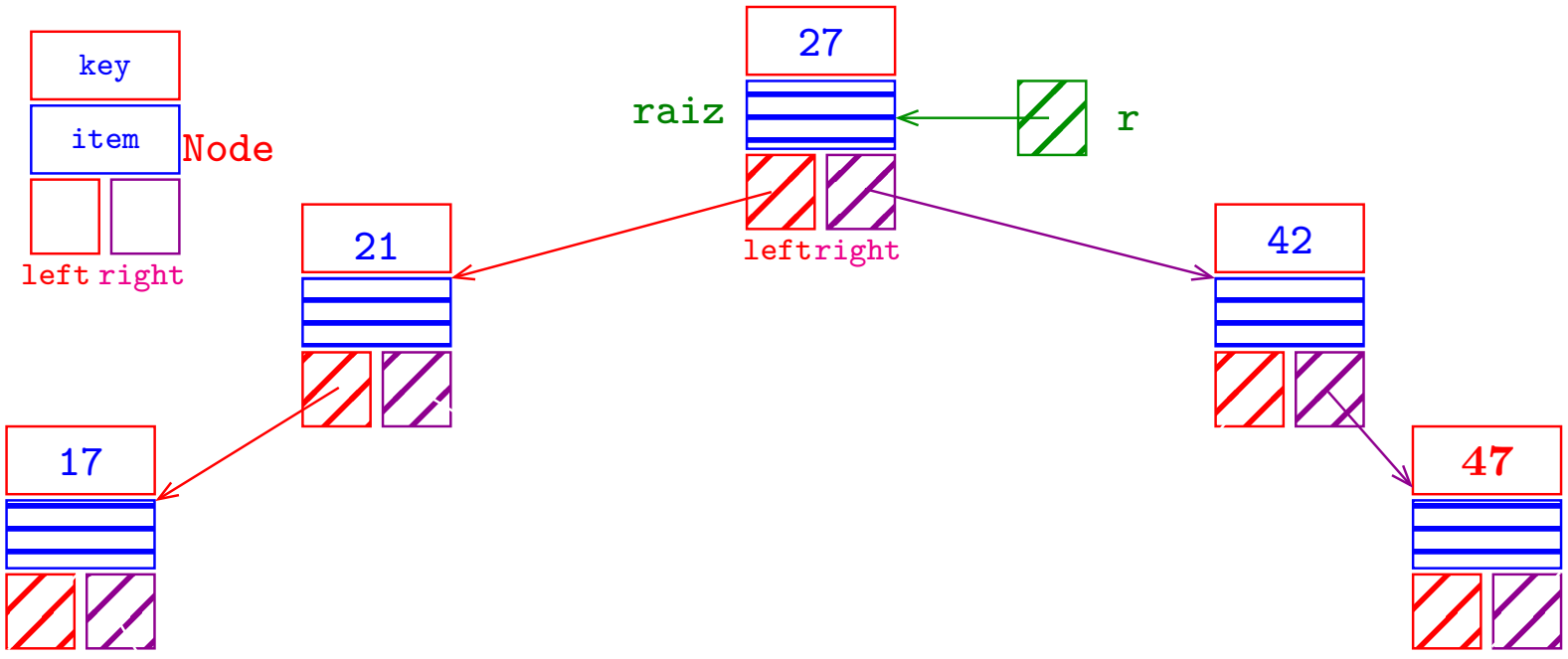
ordem put(): 27 21 **17**

Ilustração de put()



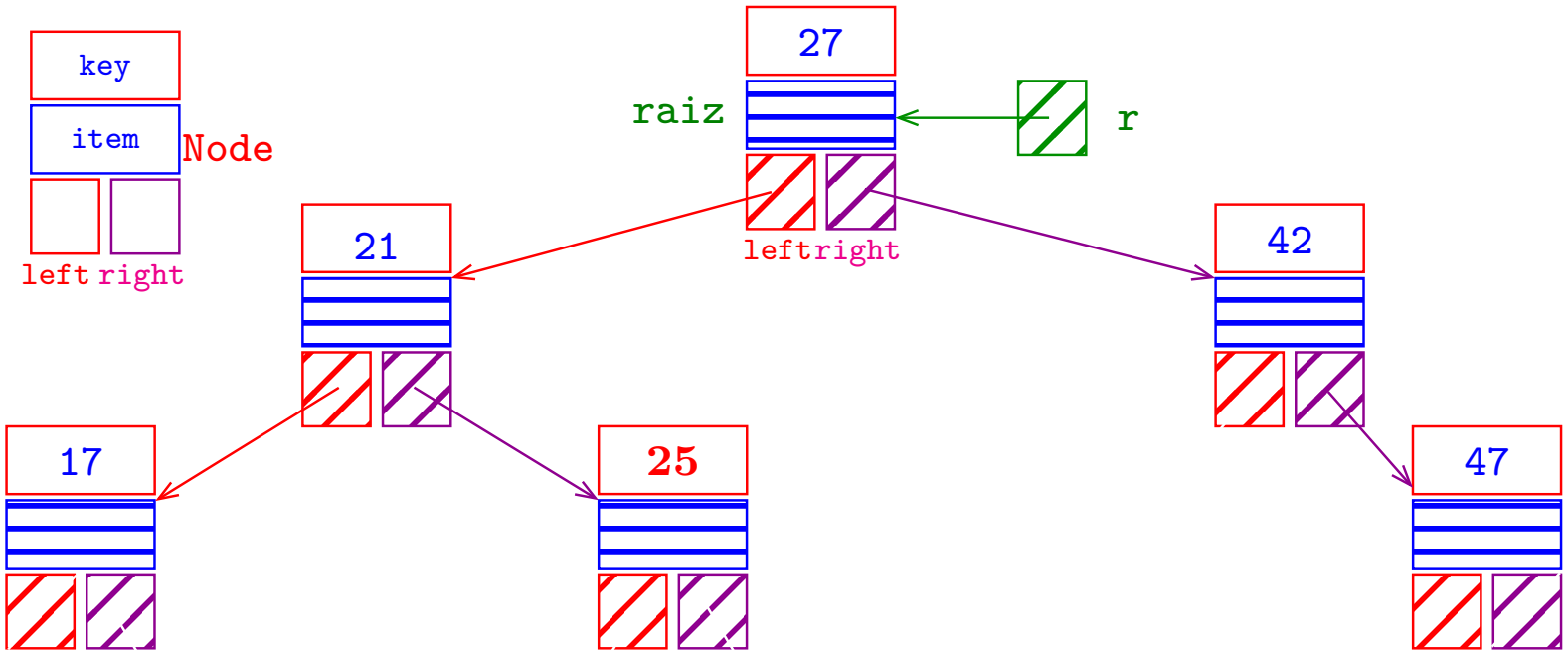
ordem put(): 27 21 17 **42**

Ilustração de put()



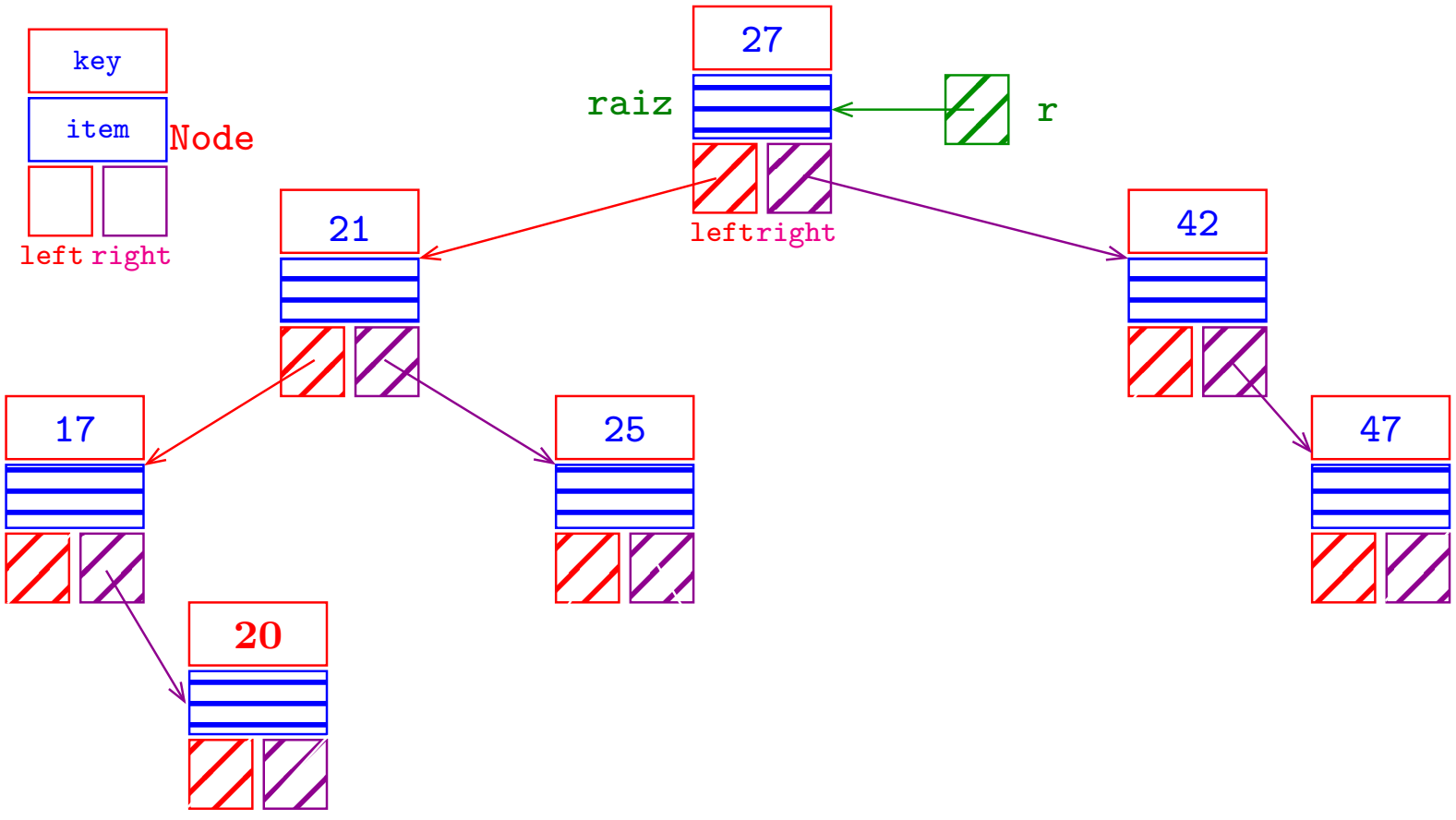
ordem put(): 27 21 17 42 **47**

Ilustração de put()



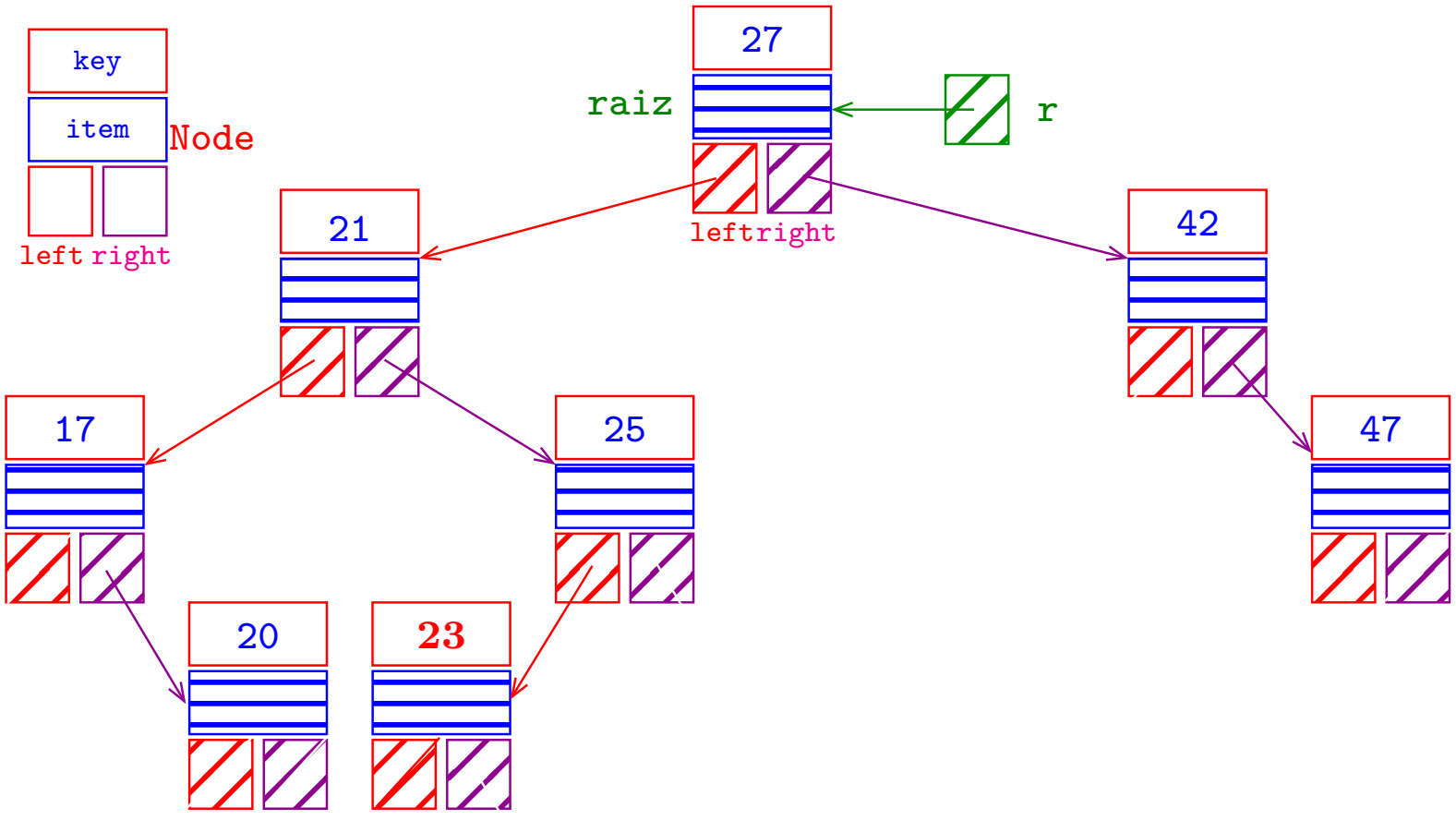
ordem `put()`: 27 21 17 42 47 **25**

Ilustração de put()



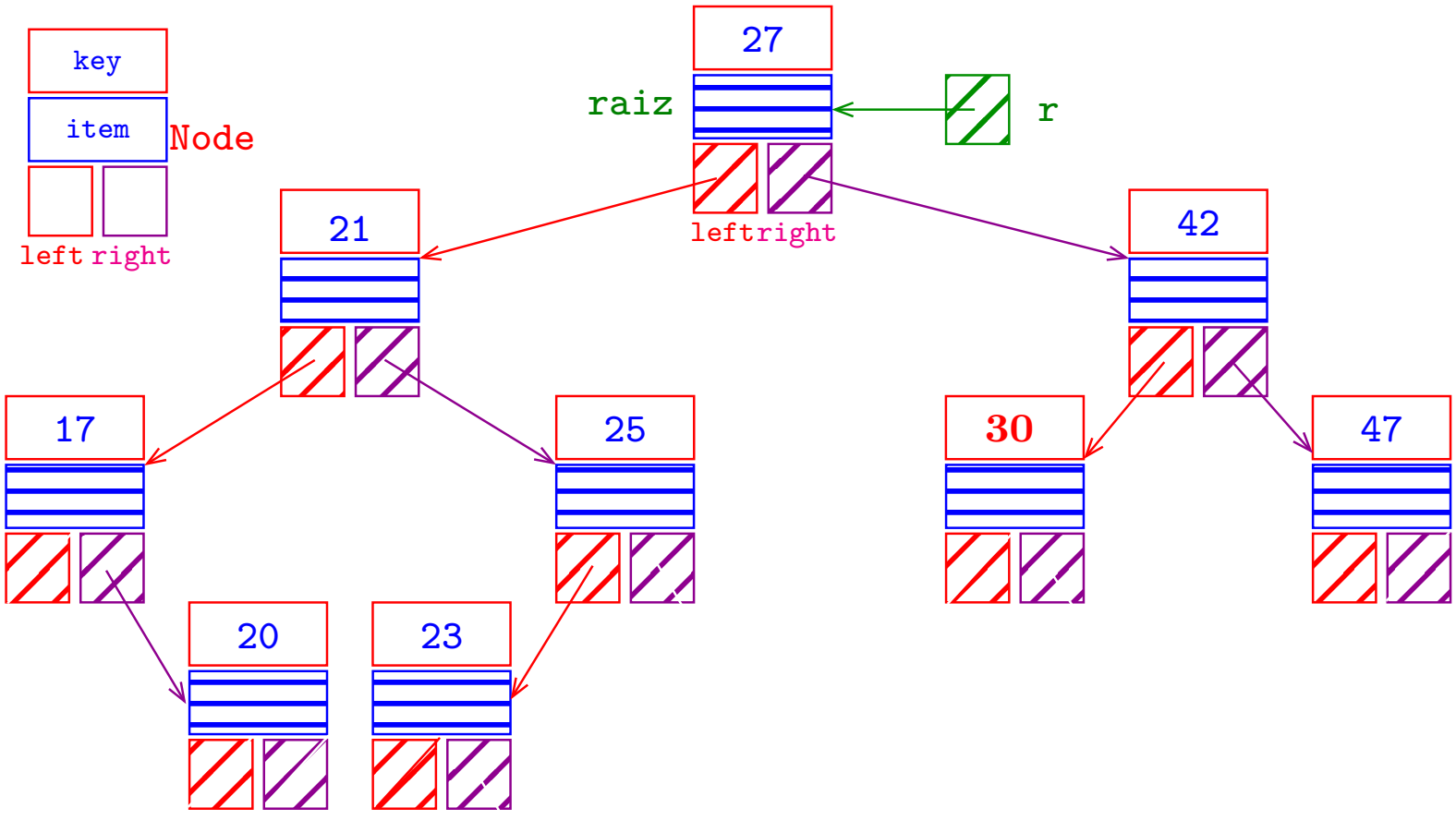
ordem `put()`: 27 21 17 42 47 25 **20**

Ilustração de put()



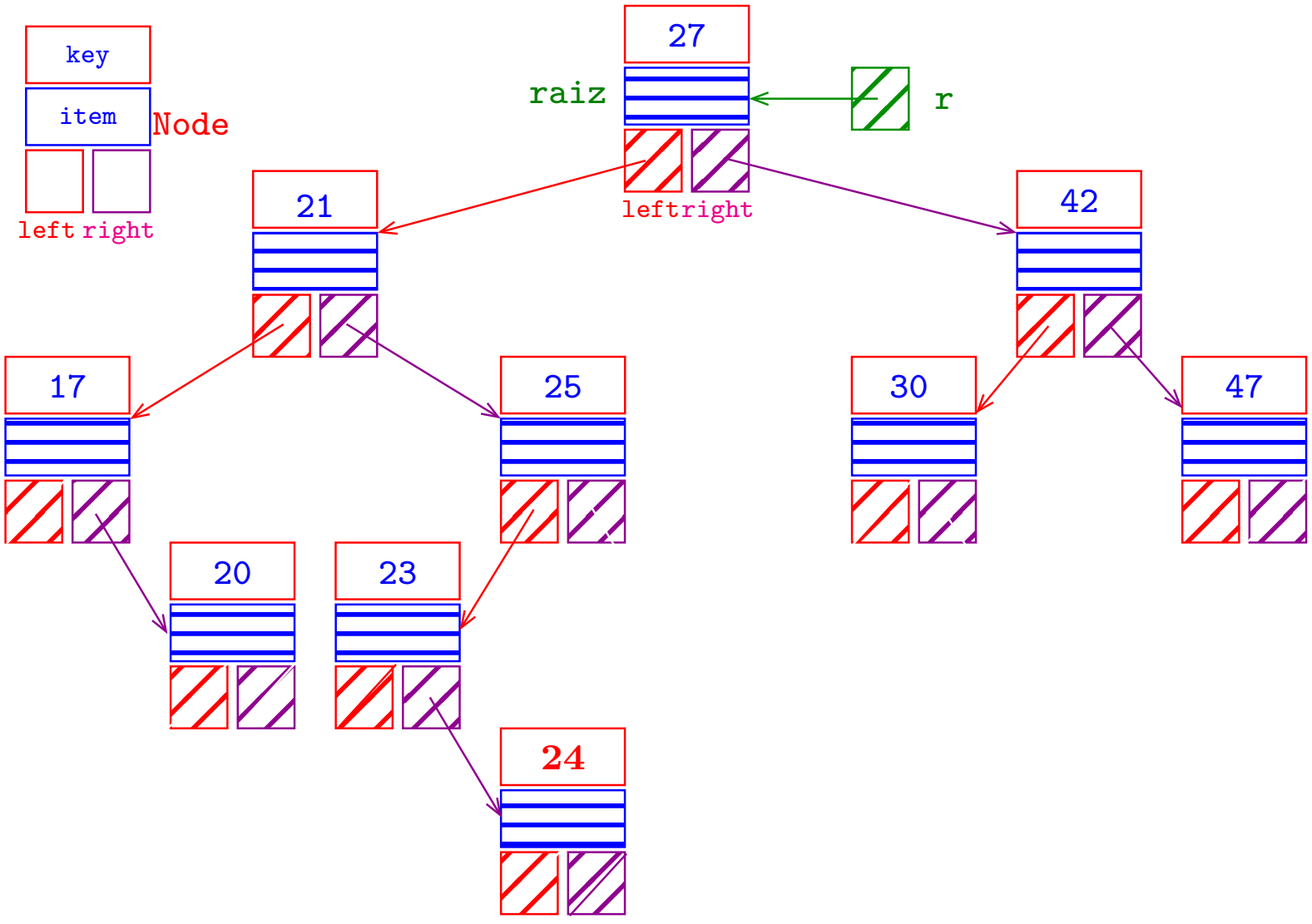
ordem put(): 27 21 17 42 47 25 20 **23**

Ilustração de put()



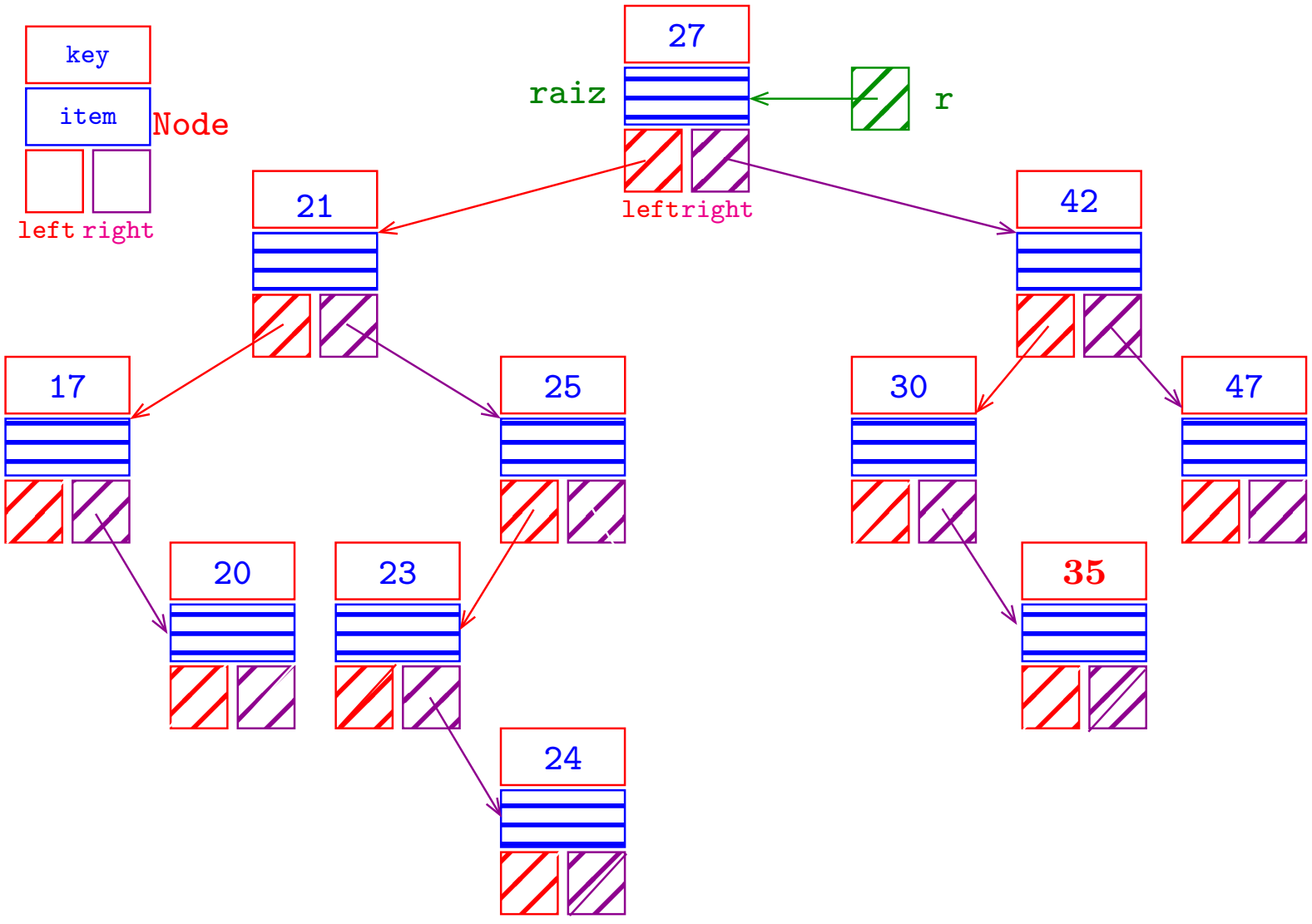
ordem put(): 27 21 17 42 47 25 20 23 **30**

Ilustração de put()



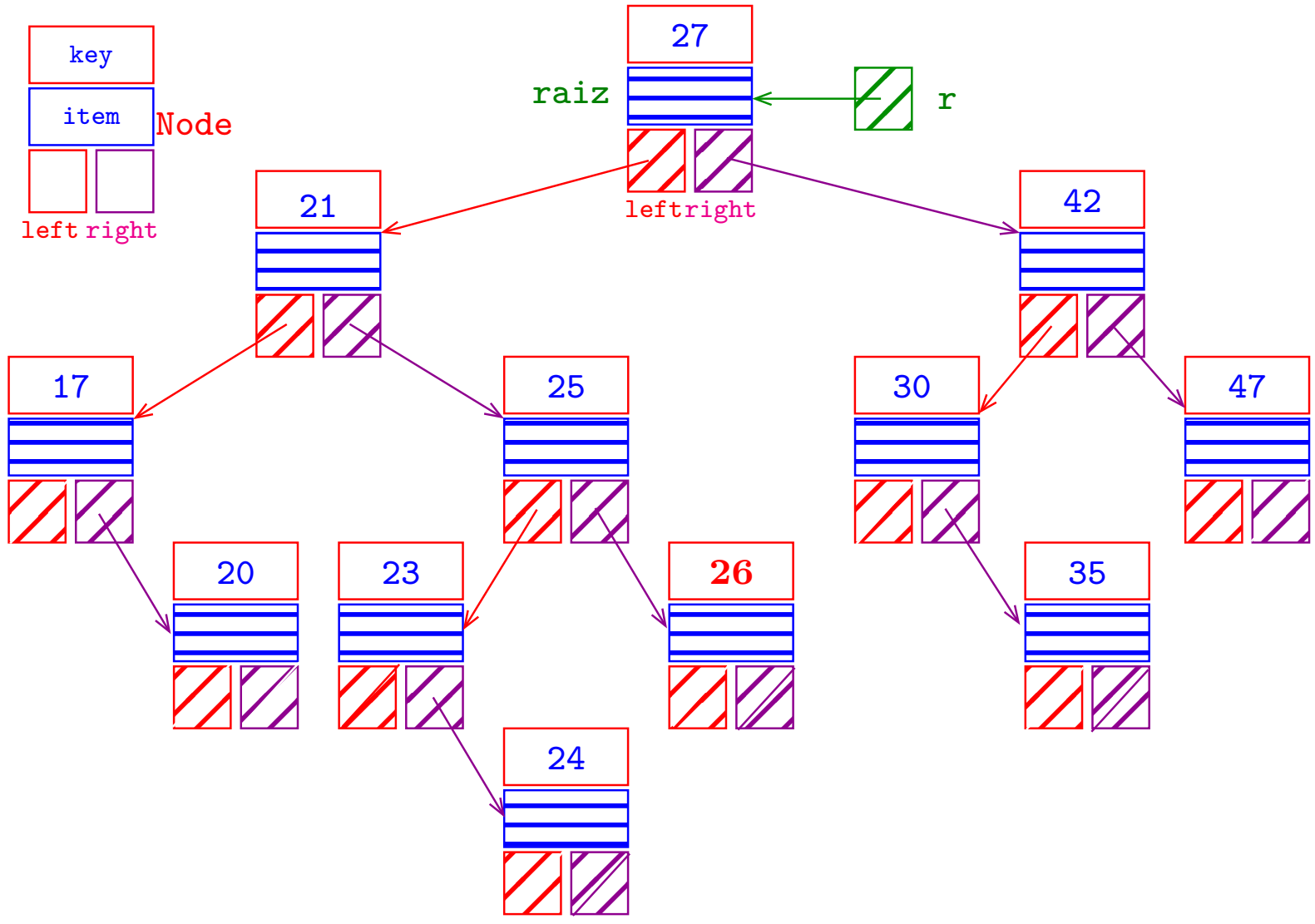
ordem put(): 27 21 17 42 47 25 20 23 30 **24**

Ilustração de put()



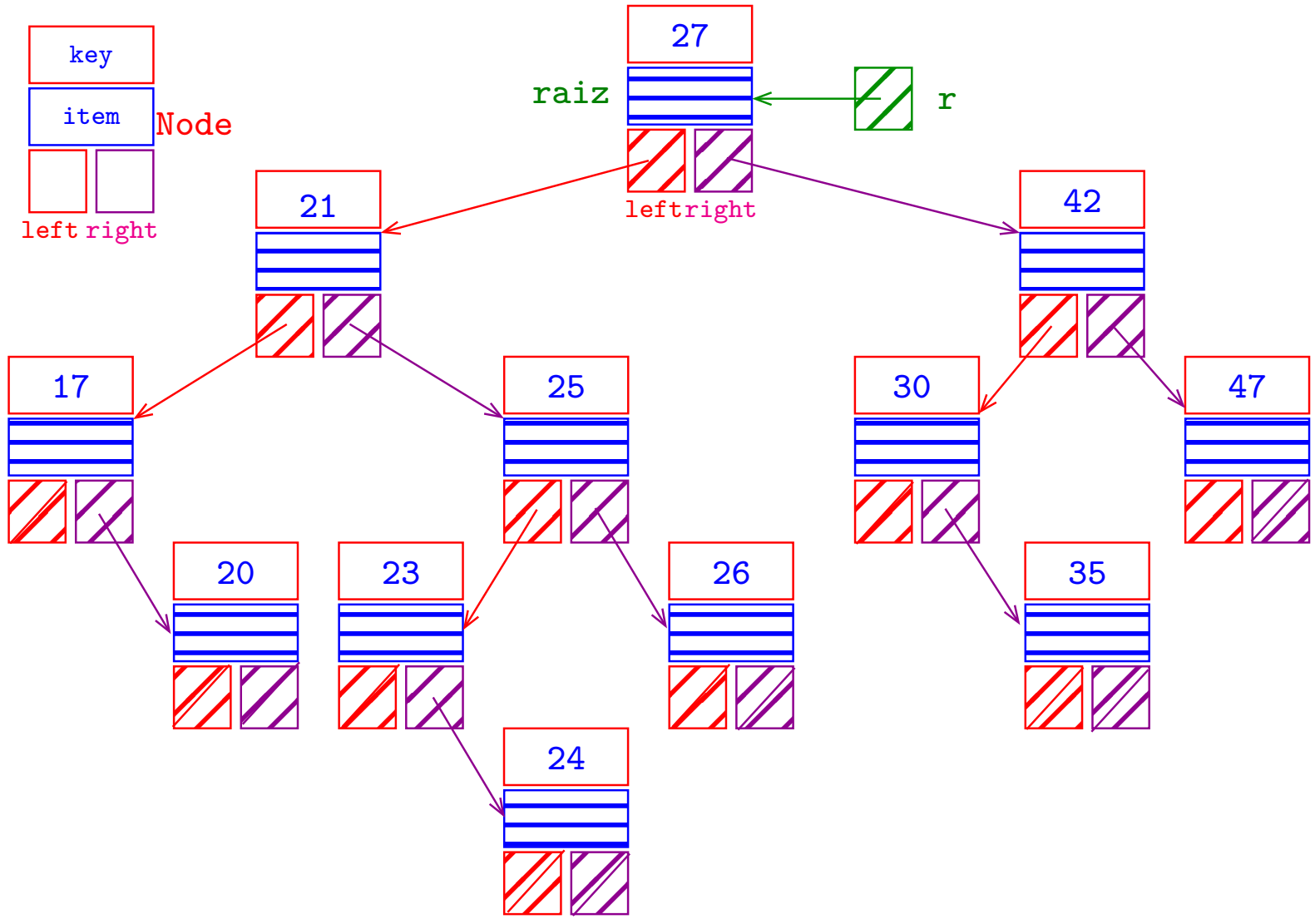
ordem put(): 27 21 17 42 47 25 20 23 30 24 **35**

Ilustração de put()



ordem put(): 27 21 17 42 47 25 20 23 30 24 35 **26**

Ilustração de put()



ordem put(): 27 21 17 42 47 25 20 23 30 24 35 26

BST: put(key, val)

Recebe key e val e insere um novo nó no lugar correto da árvore de modo que a árvore continue sendo de busca. Se key está na BST, o seu valor é atualizado.

```
void put(Key key, Value val) {
   r = putTree(r, key, val);
}
```

BST: put(key, val)

```
static
Node putTree(Node x, Key key, Value val) {
   if (x == NULL)
      return newNode(key, val);
```

BST: put(key, val)

```
static
Node putTree(Node x, Key key, Value val) {
   if (x == NULL)
      return newNode(key, val);
   int cmp = compare(key, x->key);
   if (cmp < 0)
      x->left = putTree(x->left, key, val);
   else if (cmp > 0)
      x->right = putTree(x->right, key, val);
   else x->val = val;
   return x;
}
```

Ilustração de `min()`

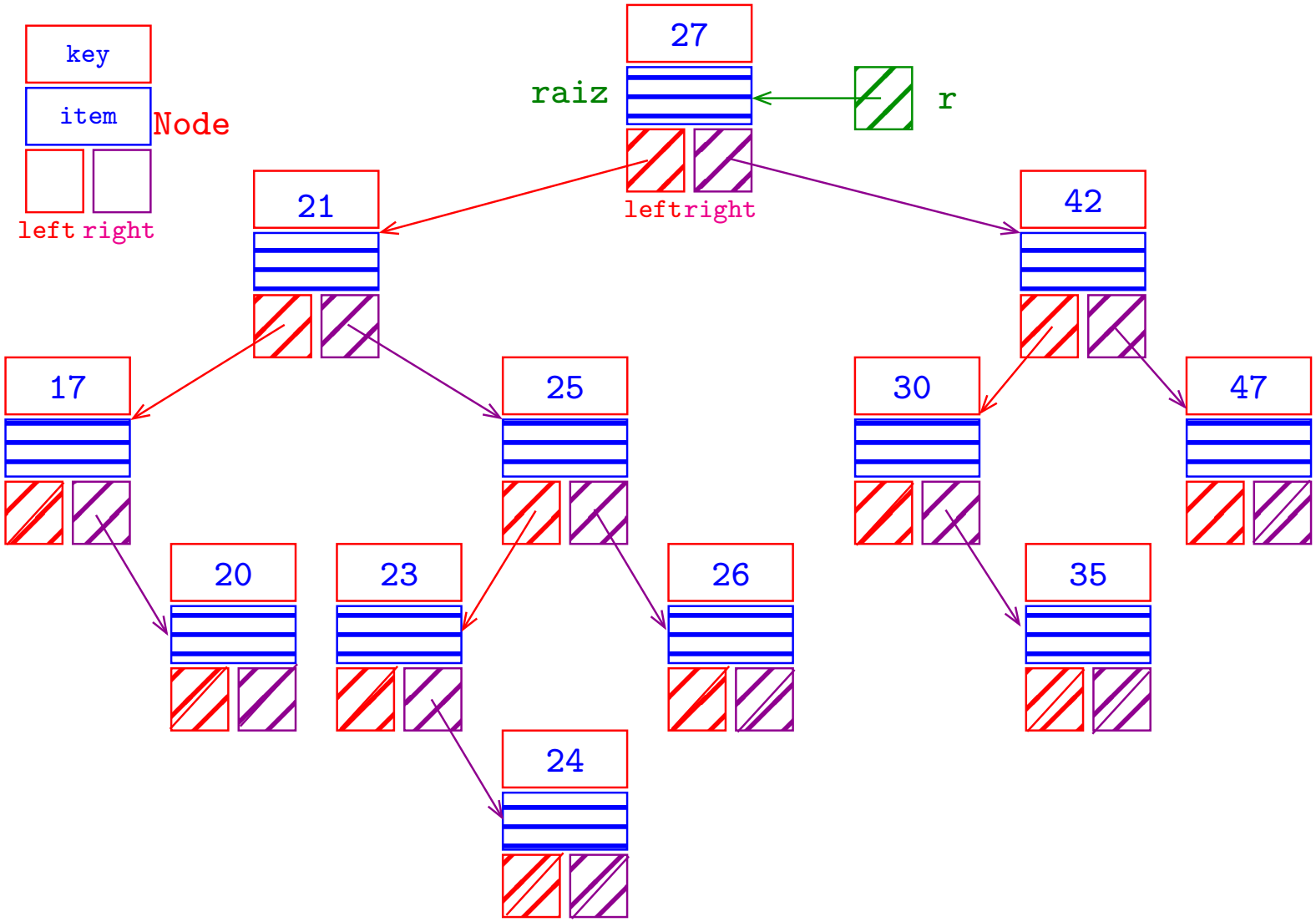
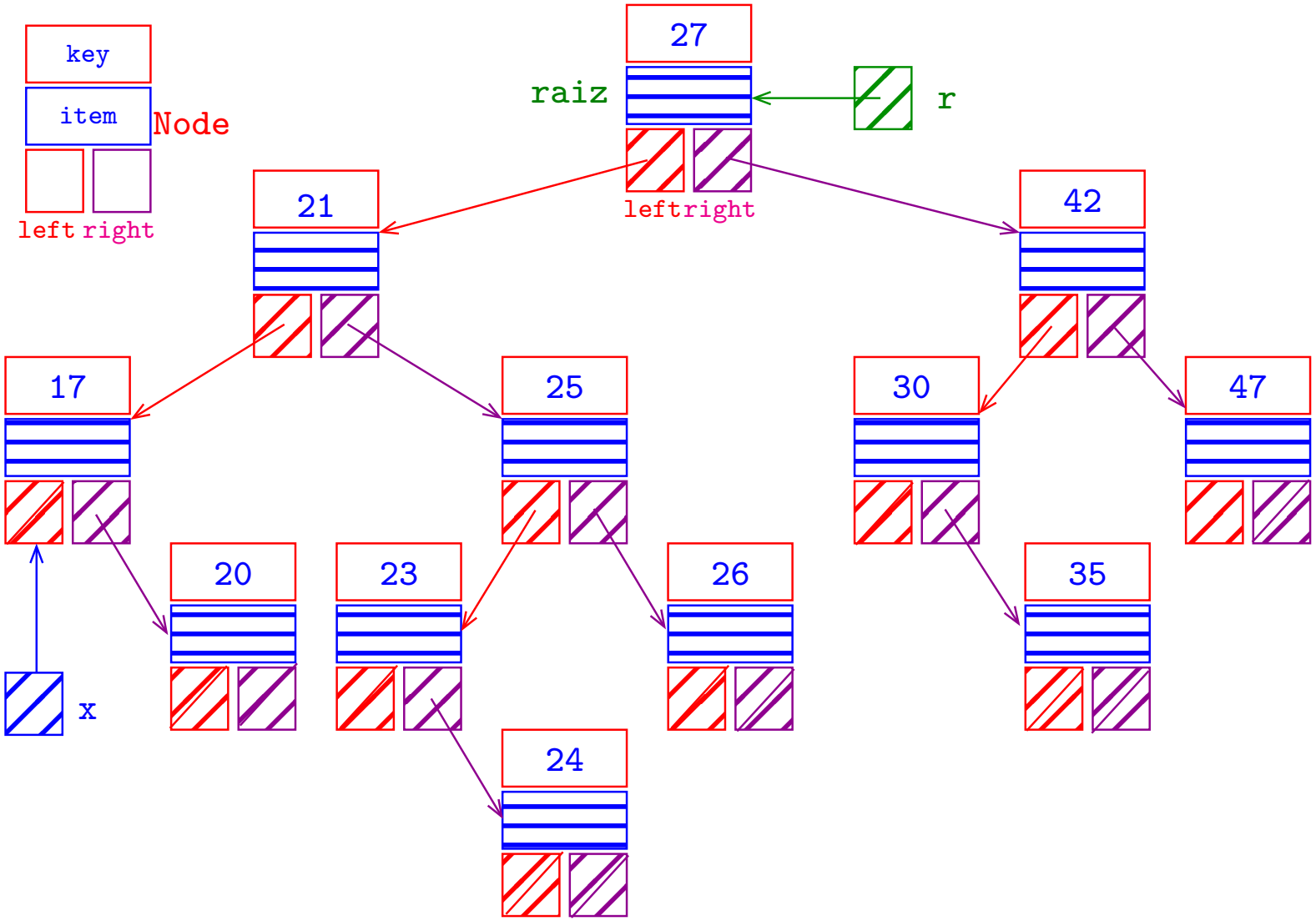


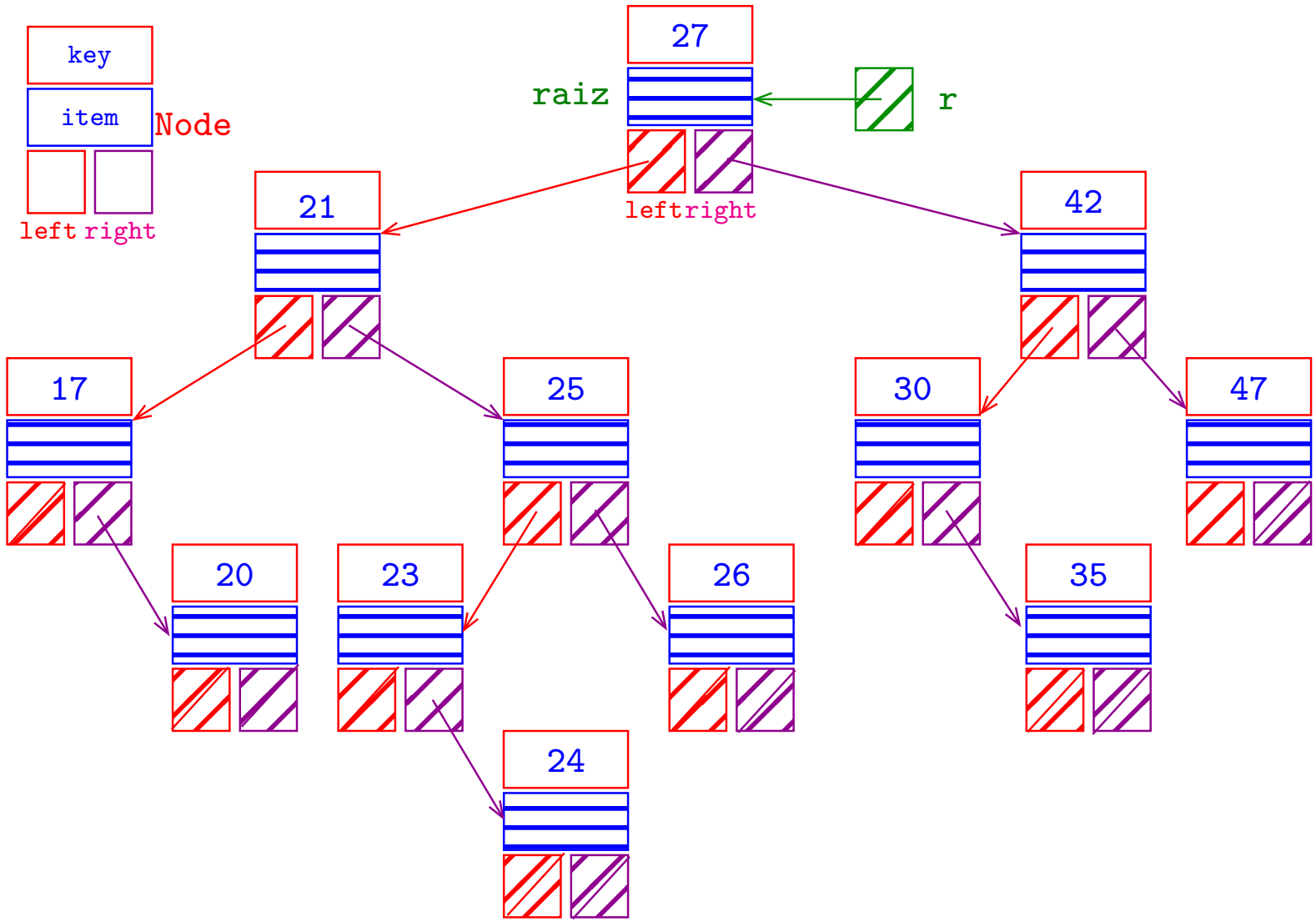
Ilustração de `min()`



BST: min()

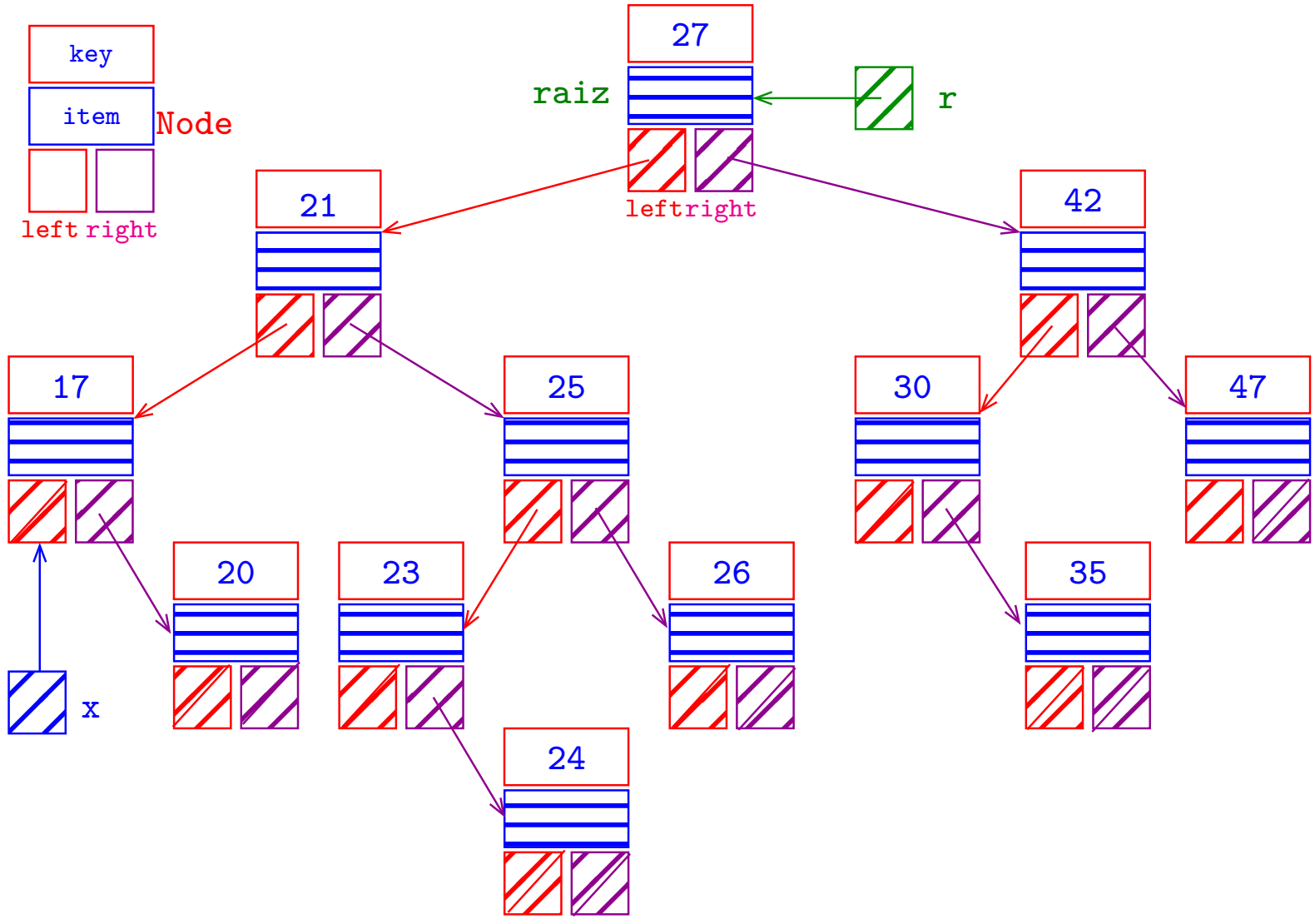
```
/* Retorna a menor chave na ST. */
Key min() {
   if (r == NULL) return NULL;
   return minTree(r)->key;
}
/* Retorna o nó da menor chave
   da subárvore cuja raiz é x.  */
static Node minTree(Node x) {
   if (x == NULL) return NULL;
   if (x->left == NULL) return x;
   return minTree(x->left);
}
```

Ilustração de deleteMin()



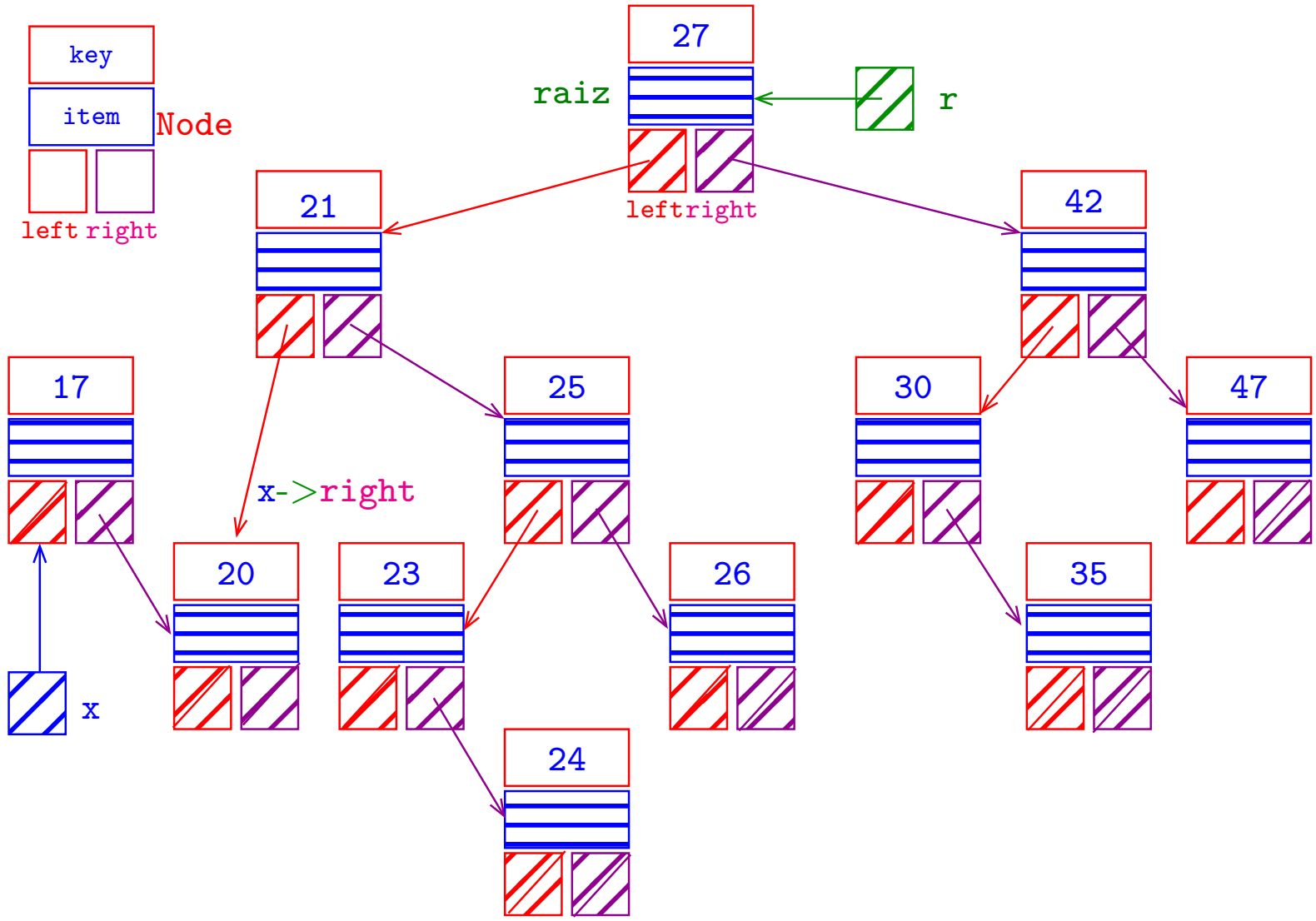
x ← nó com a menor chave

Ilustração de deleteMin()



remova x da BST

Ilustração de deleteMin()



coletor de lixo em ação

Ilustração de deleteMin()

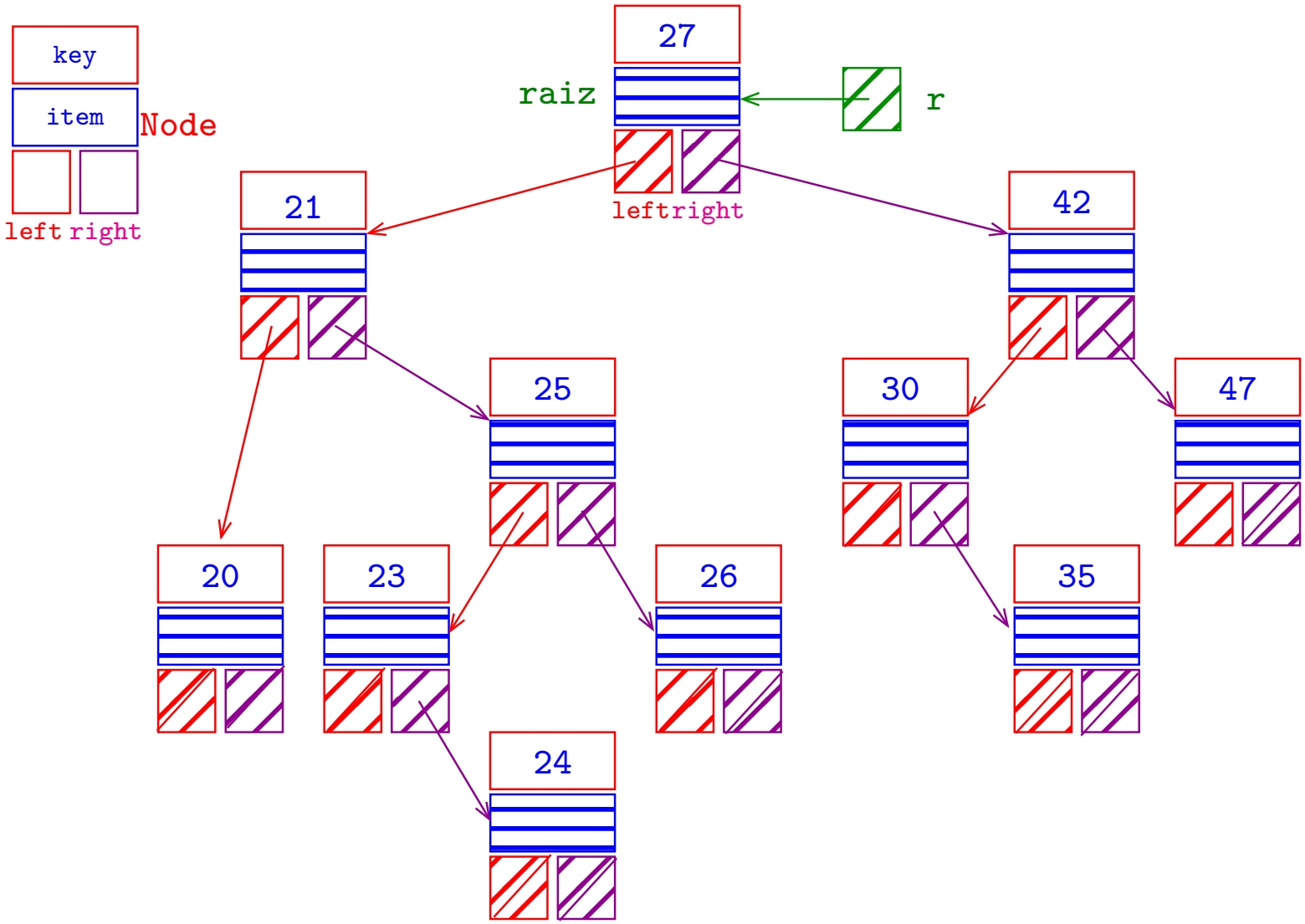
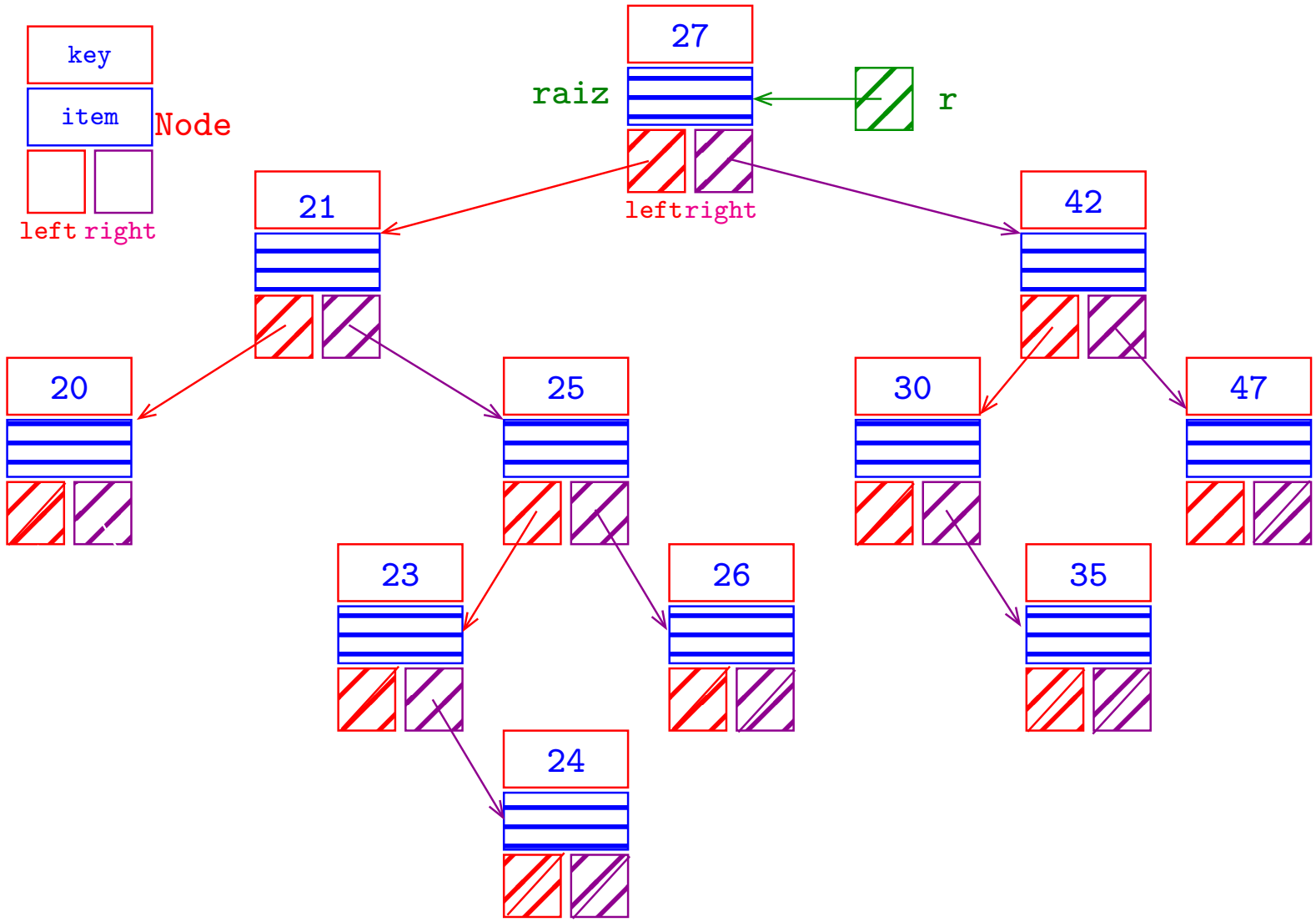


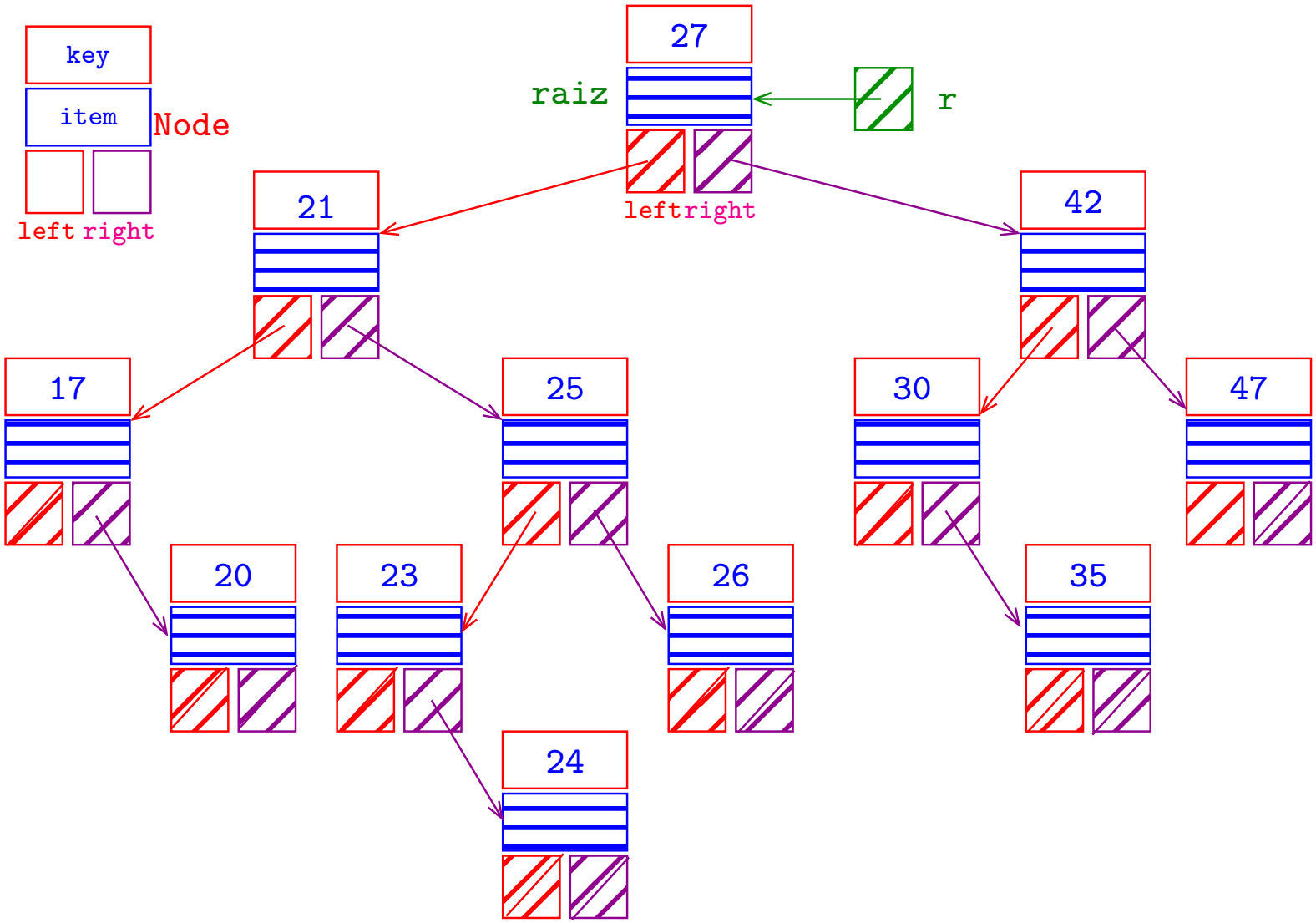
Ilustração de deleteMin()



BST: deleteMin()

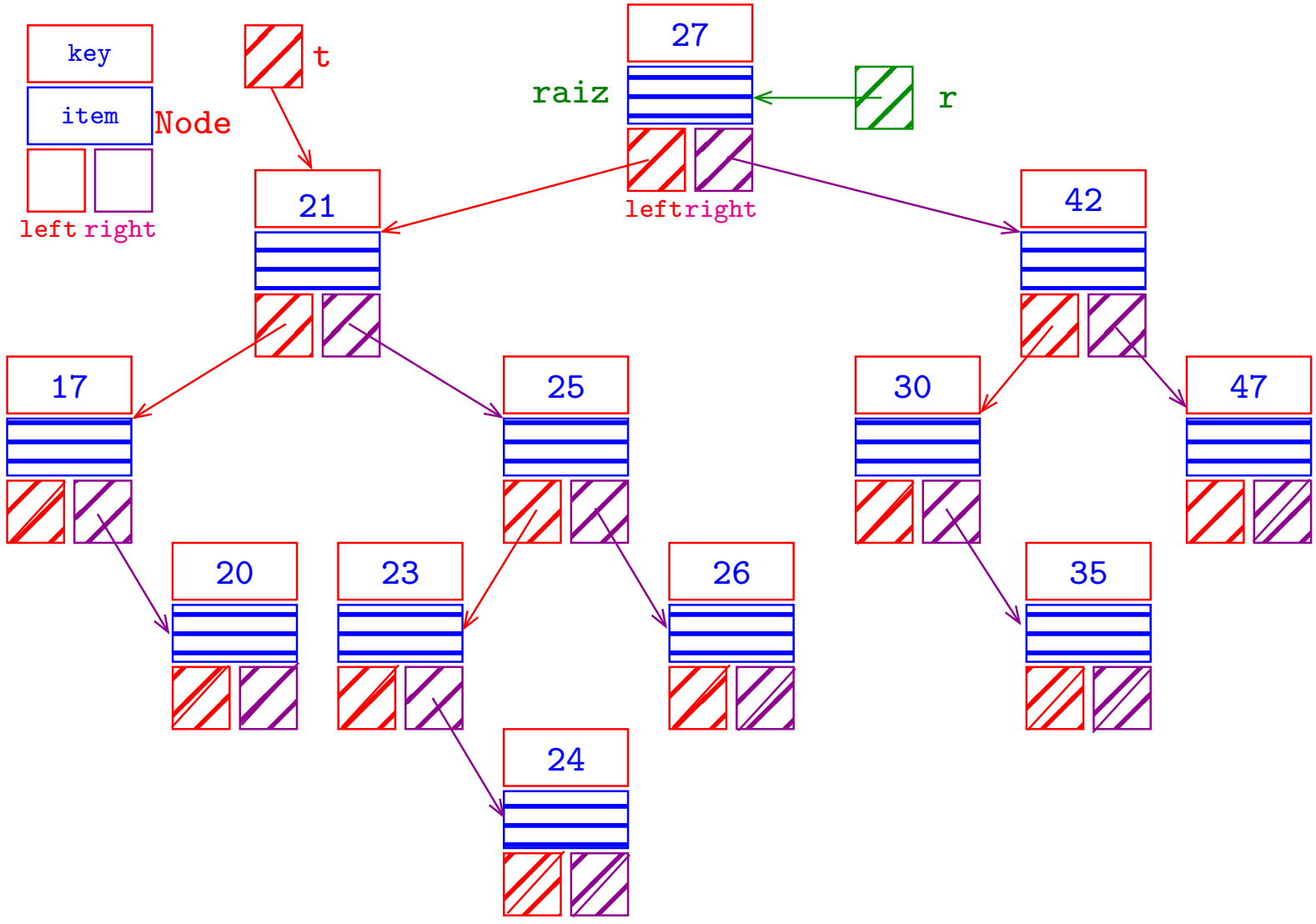
```
/* Remove o nó que tem a menor chave.  */
void deleteMin() {
   r = deleteMinTree(r);
}

/* Remove o nó que tem a menor chave */
/* na subárvore (não vazia) cuja raiz é x */
/* e retorna a raiz da subárvore resultante.  */
static Node deleteMinTree(Node x) {
   if (x == NULL) return NULL;
   if (x->left == NULL) return x->right;
   x->left = deleteMinTree(x->left);
   return x;
}
```

Ilustração de delete(21)



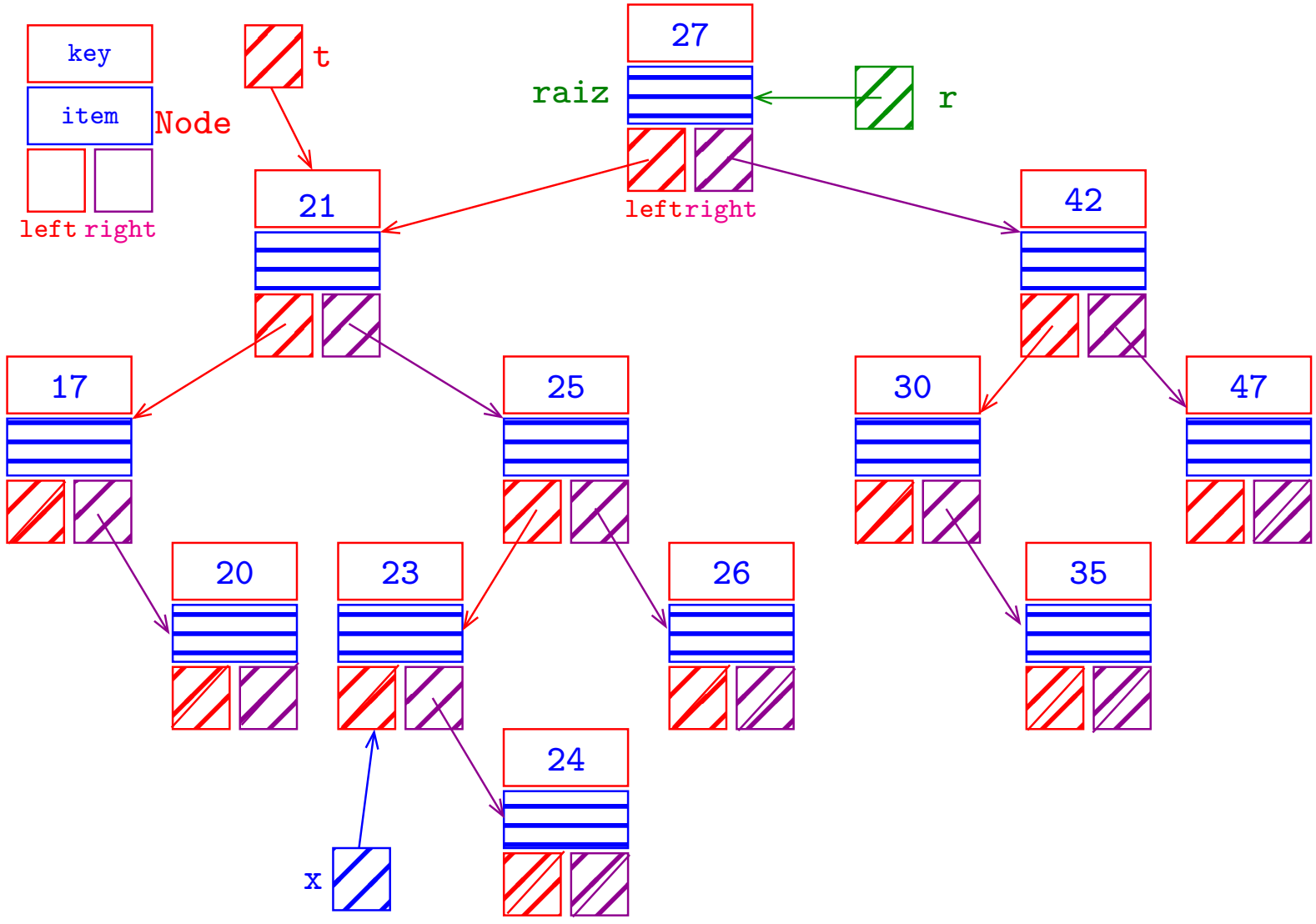
t ← nó com 21

Ilustração de delete(21)



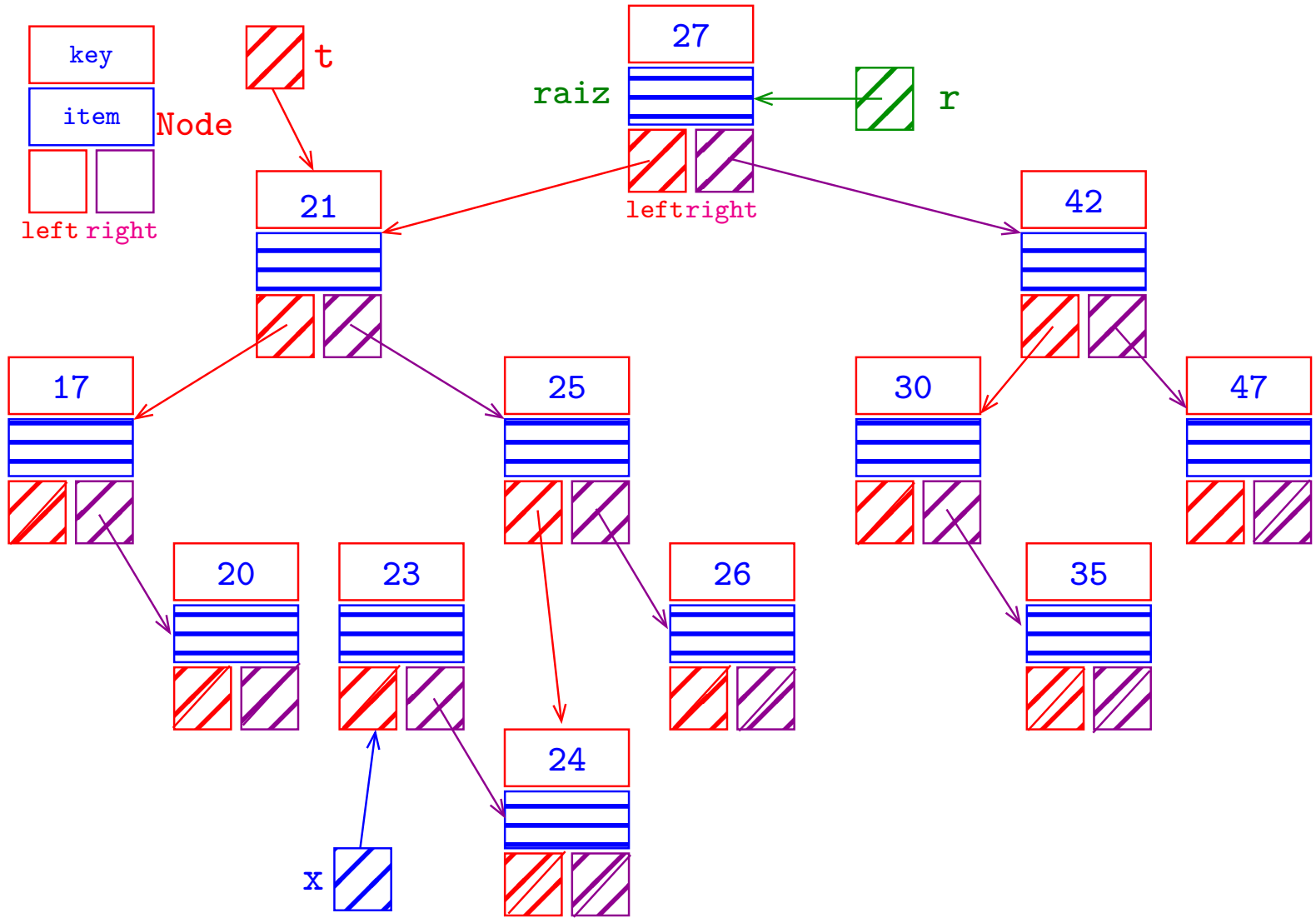
x ← menor nó em t->right (=minTree(t->right))

Ilustração de delete(21)



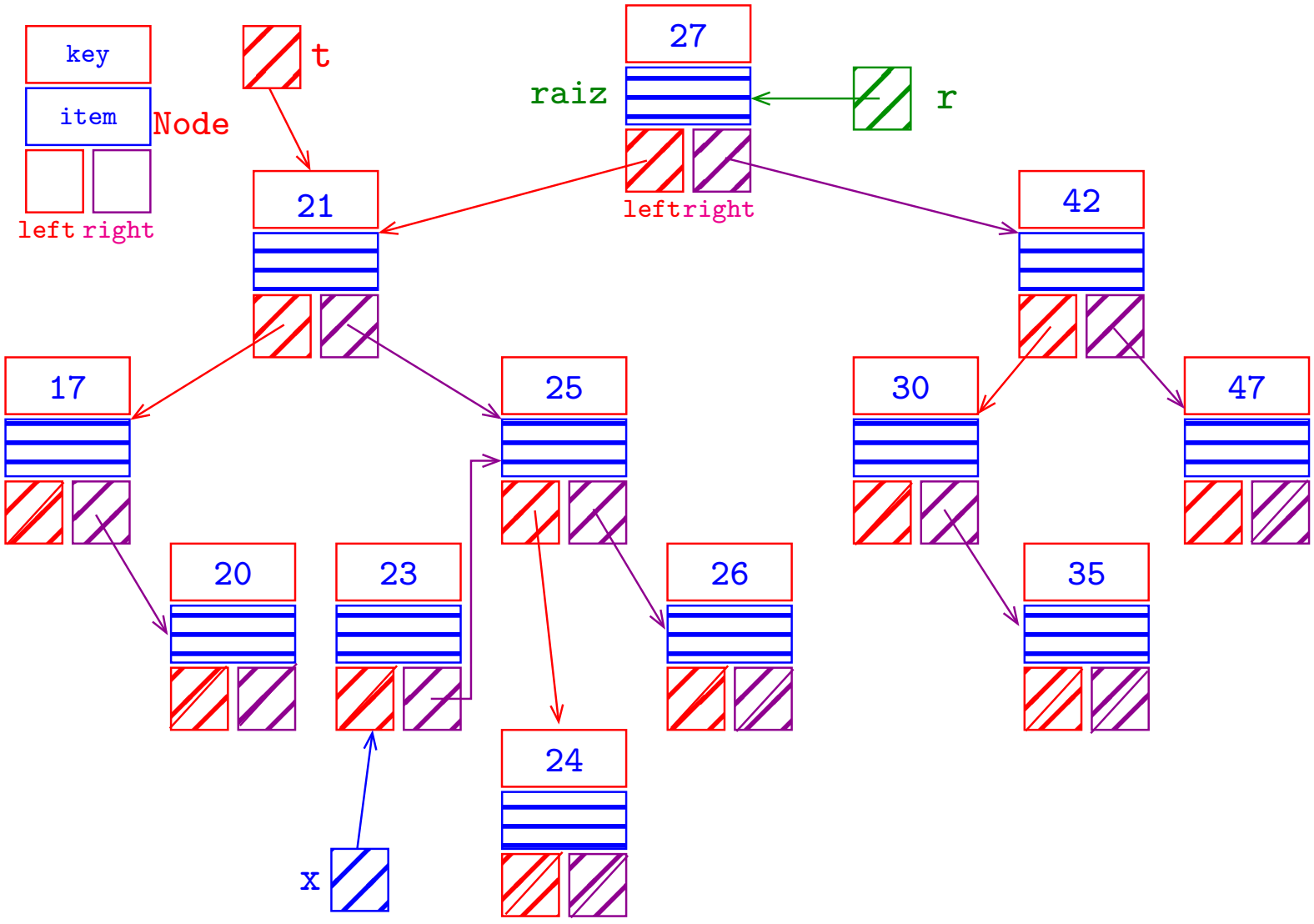
remova x da árvore (=deleteMinTree(t->right))

Ilustração de delete(21)



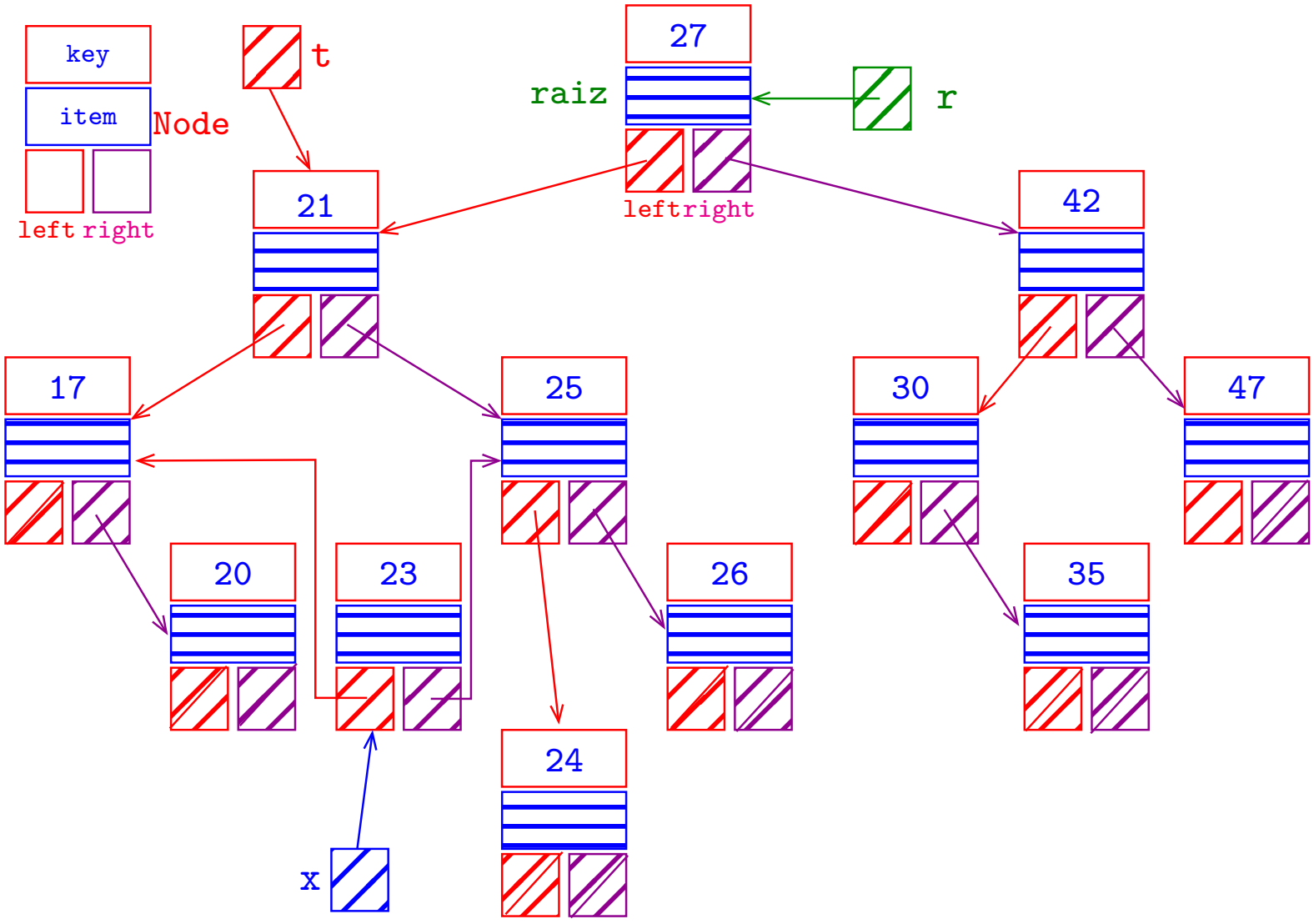
ajuste x->right (x->right ← deleteMinTree(t->right))

Ilustração de delete(21)



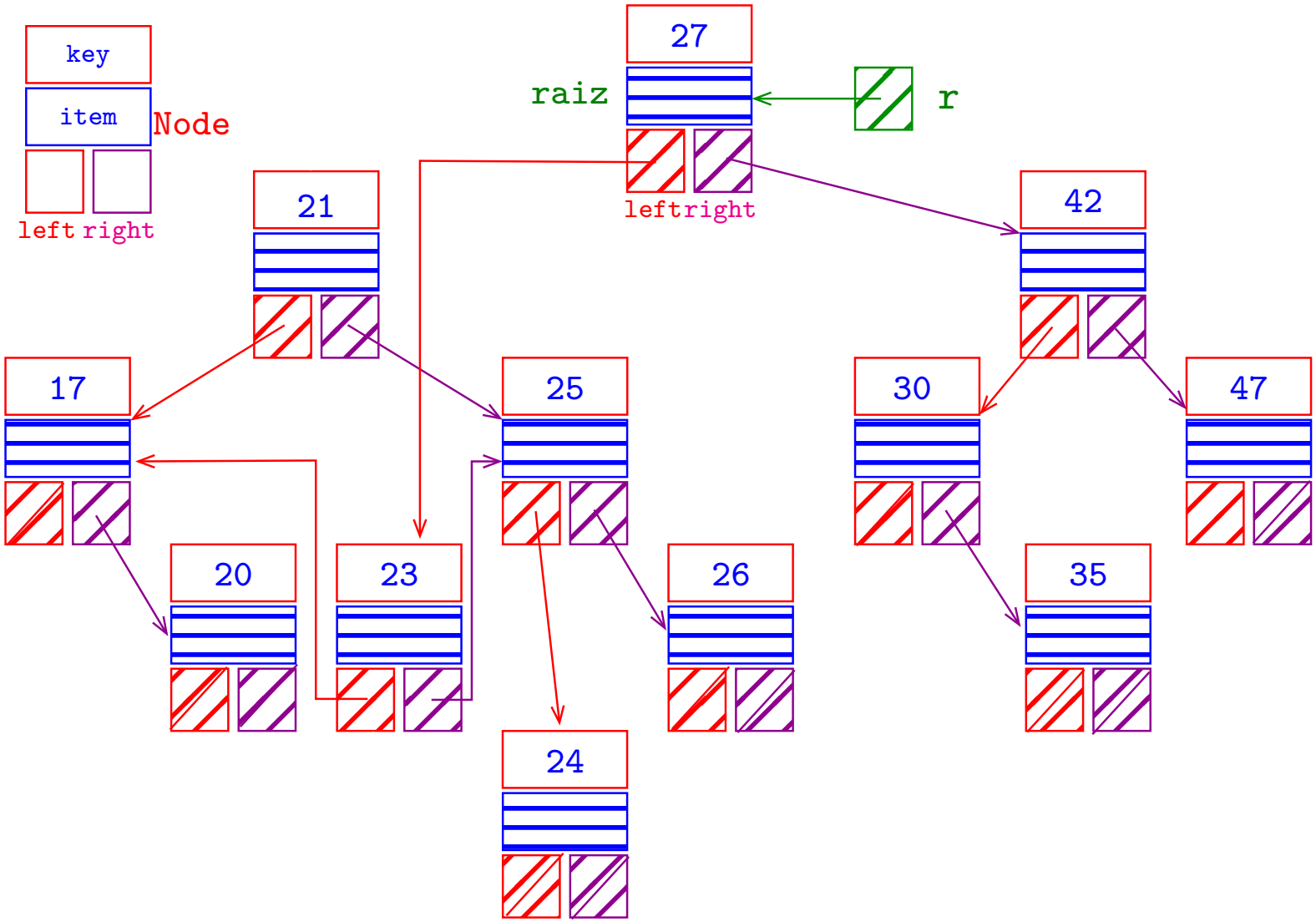
ajuste x->left (x->left ← t->left)

Ilustração de delete(21)



retorne x para ajustar a referência que era para t

Ilustração de delete(21)



coletor de lixo em ação

Ilustração de delete(21)

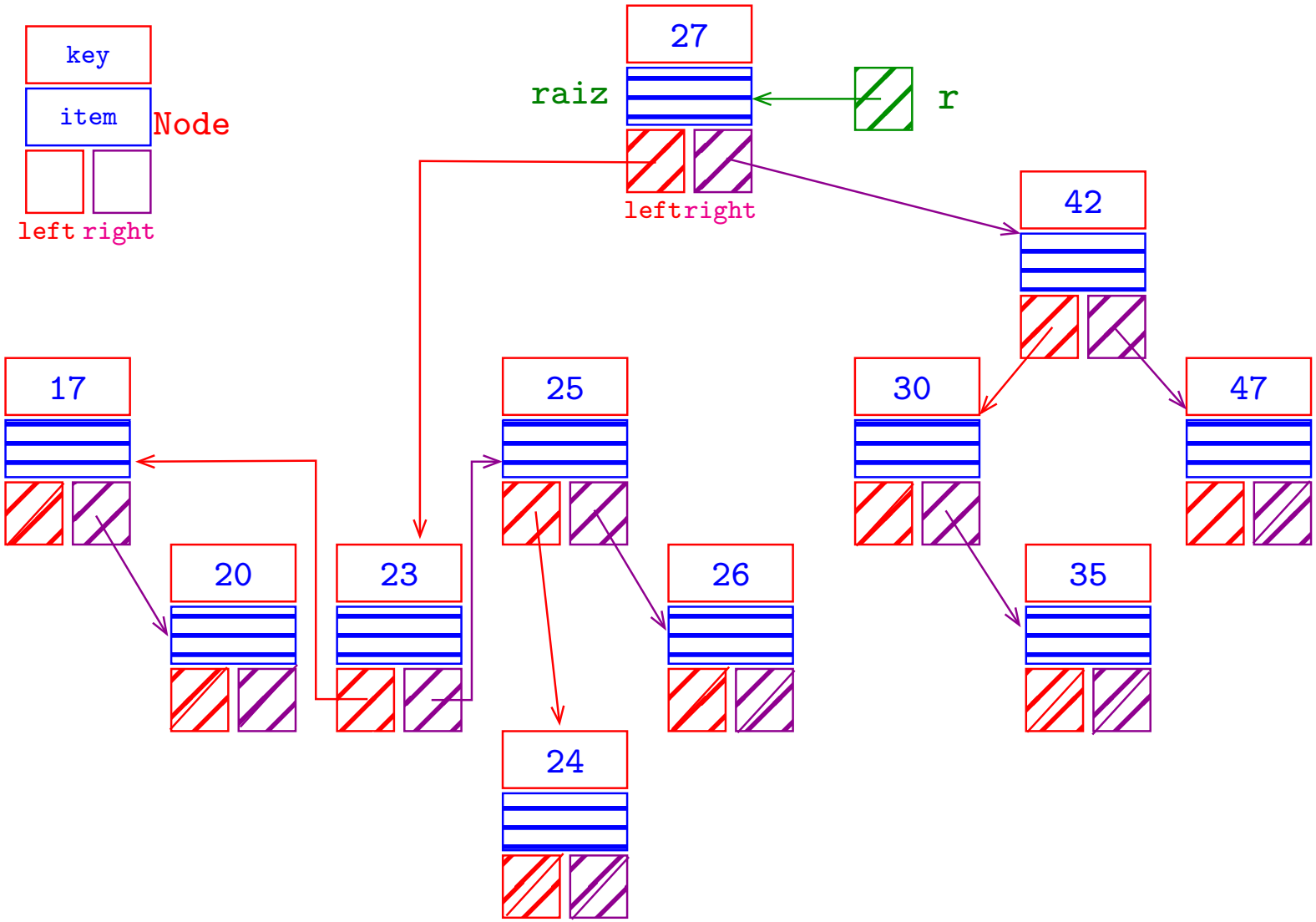
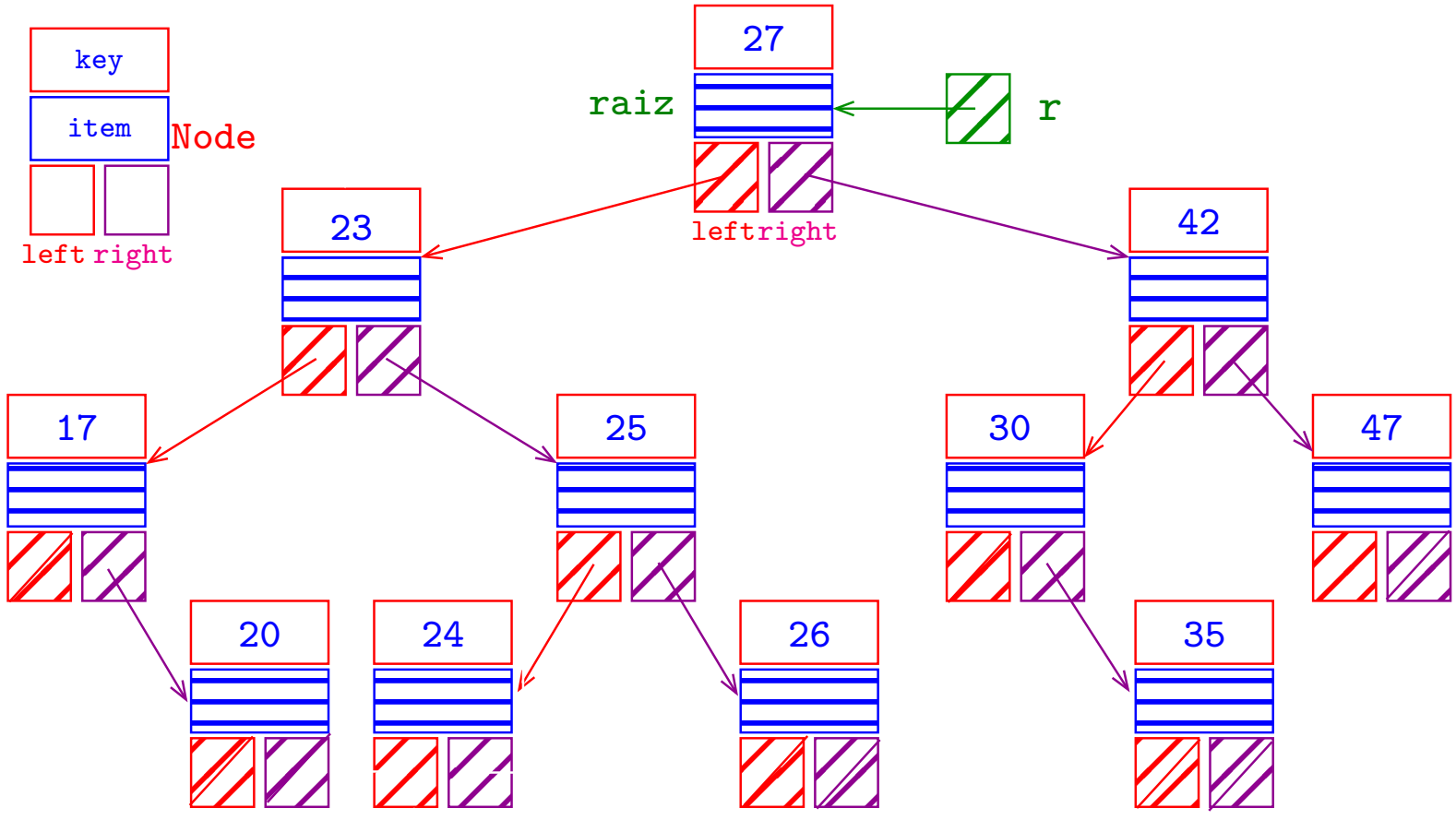


Ilustração de delete(21)



BST: delete(key)

Remove o nó que contém a chave key.
Se nenhum nó contém key, não faz nada.

```
void delete(Key key) {
   r = deleteTree(r, key);
}
```

BST: delete(key)

```
static Node deleteTree(Node x, Key key) {
   if (x == NULL) return NULL;
   int cmp = compare(key, x->key);
   if (cmp < 0)
      x->left = deleteTree(x->left, key);
   else if (cmp > 0)
      x->right = deleteTree(x->right, key);
   else{
      /* não tem algum filho */
      if (x->right == NULL) return x->left;
      if (x->left == NULL) return x->right;
```

BST: delete(key)

```
        /* x tem ambos os filhos */
        Node t = x;
        x = minTree(t->right); /* x->left == null */
        x->right = deleteMinTree(t->right);
        x->left = t->left;
    }
    return x;
}
```

Consumo de tempo

O consumo de tempo das funções `get()`, `put()` e `delete()` é, no pior caso, proporcional à altura da árvore.

Conclusão: interessa trabalhar com árvores balanceadas: árvores AVL, árvores rubro-negras, árvores ...

Mais experimentos

Consumo de tempo para se criar um `ST` em que a chaves são as palavras em `les_miserables.txt` e os valores o número de ocorrências.

estrutura	`ST`	tempo
vetor	não-ordenada	59.5
vetor MTF	não-ordenada	7.6
vetor	ordenada	1.5
lista ligada	não-ordenada	147.1
lista ligada MTF	não-ordenada	15.3
lista ligada	ordenada	115.2
skiplist	ordenada	1.1
arvore binária de busca ♡	ordenada	0.7

Tempos em segundos obtidos com `StopWatch`.