Complementando a aula passada

AULA 19

Altura

A **altura** de um nó `i` é o maior comprimento de um caminho de `i` a uma folha.

Em outras palavras, a altura de um nó `i` é o maior comprimento de uma sequência da forma

$$\langle \texttt{filho}(\texttt{i}), \texttt{filho}(\texttt{filho}(\texttt{i})), \texttt{filho}(\texttt{filho}(\texttt{filho}(\texttt{i}))), \ldots \rangle$$

onde $\texttt{filho}(\texttt{i})$ vale $2\texttt{i}$ ou $2\texttt{i} + 1$.

Os nós que têm **altura zero** são as folhas.

A altura de um nó `i` é $\lfloor \lg(m/\texttt{i}) \rfloor$.

Exercício

A altura de um nó $\texttt{i}$ é o comprimento da sequência

$$\langle 2\,\texttt{i}, 2^2\,\texttt{i}, 2^3\,\texttt{i}, \ldots, 2^h\,\texttt{i} \rangle$$

onde $2^h\,\texttt{i} \leq m < 2^{(h+1)}\,\texttt{i}$. Assim,

$$\begin{array}{rcccl}
2^h\,\texttt{i} & \leq & m & < & 2^{h+1}\texttt{i} \quad \Rightarrow \\
2^h & \leq & m/\texttt{i} & < & 2^{h+1} \quad \Rightarrow \\
\lg 2^h & \leq & \lg(m/\texttt{i}) & < & \lg 2^{h+1} \quad \Rightarrow \\
h & \leq & \lg(m/\texttt{i}) & < & h+1
\end{array}$$

Portanto, a altura de $\texttt{i}$ é $h = \lfloor \lg(m/\texttt{i}) \rfloor$.

Exercício 10.B

$A[1\,..\,m]$ tem no máximo $\lceil m/2^{h+1} \rceil$ nós com altura h.

Exercício 10.B

$A[1\,..\,m]$ tem no máximo $\lceil m/2^{h+1} \rceil$ nós com altura h.

Exemplo: N_h = número de nós à altura h

m	$\lfloor m/2^{0+1} \rfloor$	N_0	$\lceil m/2^{0+1} \rceil$	$\lfloor m/2^{1+1} \rfloor$	N_1	$\lceil m/2^{1+1} \rceil$
16	8	8	8	4	4	4
17	8	9	9	4	4	5
18	9	9	9	4	5	5
19	9	10	10	4	5	5
20	10	10	10	5	5	5
21	10	11	11	5	5	6
22	11	11	11	5	6	6
23	11	12	12	5	6	6
24	12	12	12	6	6	6

Solução

Prova: Um nó i tem altura $h = \lfloor \lg(m/\mathtt{i}) \rfloor$. Logo,

$$\begin{aligned} 2^h &\leq m/\mathtt{i} < 2^{h+1} \\ m/2^{h+1} &< \mathtt{i} \leq m/2^h \\ \lfloor m/2^{h+1} \rfloor + 1 &\leq \mathtt{i} \leq \lfloor m/2^h \rfloor \end{aligned}$$

O número de possíveis valores para i é

$$\begin{aligned} &= \lfloor m/2^h \rfloor - \lfloor m/2^{h+1} \rfloor \\ &= \lfloor m/2^h \rfloor - \lfloor \lfloor m/2^h \rfloor /2 \rfloor \\ &= \lceil \lfloor m/2^h \rfloor /2 \rceil \\ &\leq \lceil \lceil m/2^h \rceil /2 \rceil = \lceil m/2^{h+1} \rceil . \end{aligned}$$

Função insereHeap

Inseção de um elemento x em um max-heap v[1..n]

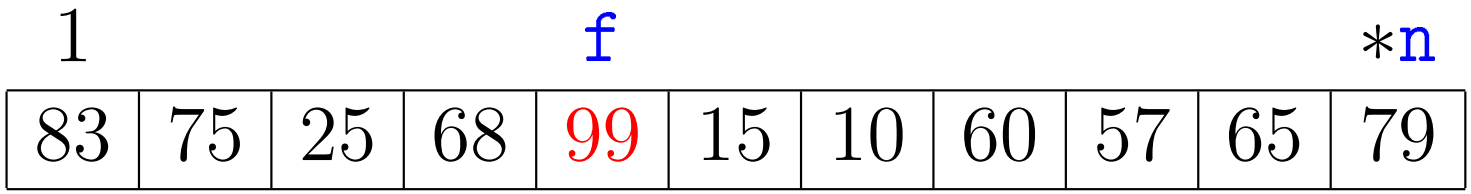
```
void insereHeap (int x, int *n, int v[]) {
    int f /* filho */, p/* pai */, t;
1   *n += 1; f = *n; p = f / 2; v[f] = x;
2   while/*D*/ (f > 1 && v[p] < v[f]) {
3       t = v[p];
4       v[p] = v[f];
5       v[f] = t;

        /* pai no papel de filho */
6       f = p; p = f / 2;
    }
 }
```

Função `insereHeap`

Relações invariantes: Em /*D*/ vale que:

(i0) $\texttt{v}[1\,..\,*\texttt{n}]$ é uma permutação do vetor original

(i1) $\texttt{v}[\texttt{i}/2] \geq \texttt{v}[\texttt{i}]$ para todo $\texttt{i} = 2, \ldots, *\texttt{n}$ diferente de `f`.



Conclusão

O consumo de tempo da função `insereHeap` é proporcional a $\lg n$, onde n é o número de elementos no `max-heap`.

O consumo de tempo da função `heapSort` é $O(n)$, onde n é o número de elementos no `max-heap`.

Mais análise experimental

Algoritmos implementados:

`mergeR`	`mergeSort` recursivo.
`mergeI`	`mergeSort` iterativo.
`quick`	`quickSort` recursivo.
`heap`	`heapSort`.

Mais análise experimental

A plataforma utilizada nos experimentos foi um computador rodando Ubuntu GNU/Linux 3.5.0-17

Compilador:

```
gcc -Wall -ansi -O2 -pedantic
-Wno-unused-result.
```

Computador:

```
 model name:  Intel(R) Core(TM)2 Quad CPU Q6600 @
2.40GHz
cpu MHz :  1596.000
cache size:  4096 KB
MemTotal :  3354708 kB
```

Aleatório: média de 10

n	mergeR	mergeI	quick	heap
8192	0.00	0.00	0.00	0.00
16384	0.00	0.00	0.00	0.00
32768	0.01	0.01	0.01	0.00
65536	0.01	0.01	0.01	0.01
131072	0.02	0.02	0.02	0.03
262144	0.05	0.04	0.04	0.06
524288	0.10	0.08	0.08	0.12
1048576	0.21	0.20	0.17	0.28
2097152	0.44	0.43	0.35	0.70
4194304	0.92	0.90	0.73	1.73
8388608	1.90	1.87	1.51	4.13

Tempos em segundos.

Decrescente

n	mergeR	mergeI	quick	heap
1024	0.00	0.00	0.00	0.00
2048	0.00	0.00	0.00	0.00
4096	0.01	0.00	0.01	0.00
8192	0.00	0.00	0.03	0.00
16384	0.00	0.00	0.14	0.00
32768	0.00	0.01	0.57	0.00
65536	0.01	0.01	2.27	0.01
131072	0.02	0.01	9.06	0.02
262144	0.03	0.03	36.31	0.04

Tempos em segundos.

Para n=524288 quickSort dá Segmentation fault (core dumped)

Crescente

n	mergeR	mergeI	quick	heap
1024	0.00	0.00	0.00	0.00
2048	0.00	0.00	0.00	0.00
4096	0.00	0.00	0.00	0.00
8192	0.00	0.00	0.03	0.00
16384	0.00	0.00	0.14	0.01
32768	0.01	0.00	0.57	0.01
65536	0.00	0.01	2.26	0.01
131072	0.02	0.02	9.05	0.02
262144	0.03	0.02	36.21	0.04

Tempos em segundos.

Para n=524288 quickSort dá Segmentation fault (core dumped)

Resumo

função	consumo de tempo	observação
`bubble`	$O(n^2)$	todos os casos
`insercao`	$O(n^2)$ $O(n)$	pior caso melhor caso
`insercaoBinaria`	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
`selecao`	$O(n^2)$	todos os casos
`mergeSort`	$O(n \lg n)$	todos os casos
`quickSort`	$O(n^2)$ $O(n \lg n)$	pior caso melhor caso
`heapSort`	$O(n \lg n)$	todos os casos

Animação de algoritmos de ordenação

Criados por Nicholas André Pinho de Oliveira:
http://nicholasandre.com.br/sorting/

Criados na Sapientia University (Romania):

https://www.youtube.com/channel/UCIqiLefbVHsOAXDAxQJH7Xw

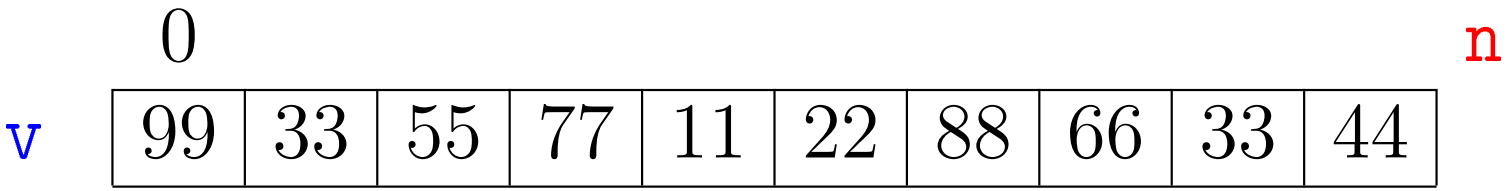
AULA 20

k-ésimo menor elemento

x é o k-**ésimo menor elemento** de um vetor v[0..n−1] se em um rearranjo crescente de v, x é o valor na posição v[k-1].

Problema: encontrar k-ésimo menor elemento de um vetor v[0 . . n-1], supondo $1 \leq k \leq n$.

Exempo: **33** é o 4o. menor elemento de:



k-ésimo menor elemento

x é o k-**ésimo menor elemento** de um vetor v[0..n−1] se em um rearranjo crescente de v, x é o valor na posição v[k-1].

Problema: encontrar k-ésimo menor elemento de um vetor v[0 . . n-1], supondo $1 \leq k \leq n$.

Exempo: **33** é o 4o. menor elemento de:

	0									n
v	99	33	55	77	11	22	88	66	33	44

pois: no vetor ordem crescente temos

	0									n
v	11	22	33	**33**	44	55	66	77	88	99

Solução inspirada em selecao()

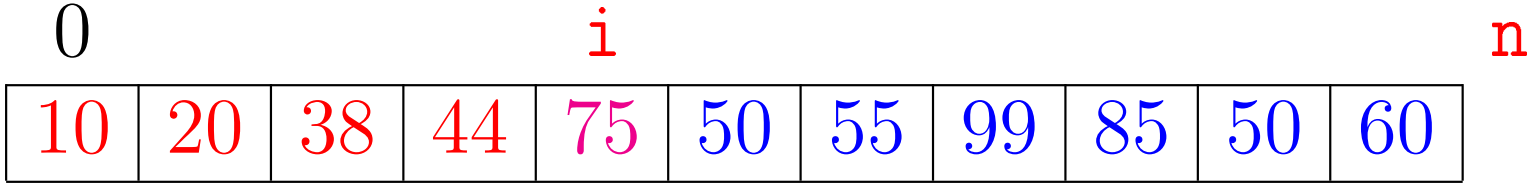
Algoritmo baseado em ordenação por seleção.
Ao final o k-ésimo menor elemento está em v[k-1].

```
void
kEsimo (int k, int n, int v[]) {
    int i, j, min, x;
1   for (i = 0; i < k; i++) {
2       min = i;
3       for (j = i+1; j < n; j++)
4           if (v[j] < v[min]) min = j;
5       x=v[i]; v[i]=v[min]; v[min]=x;
    }
}
```

Invariantes

Relações **invariantes** chaves dizem que em /*A*/ vale que:

♡ (i0) $\mathtt{v}[0\,..\,\mathtt{i}-1]$ é crescente e
$\mathtt{v}[0\,..\,\mathtt{i}-1] \le \mathtt{v}[\mathtt{i}\,..\,\mathtt{n}-1]$

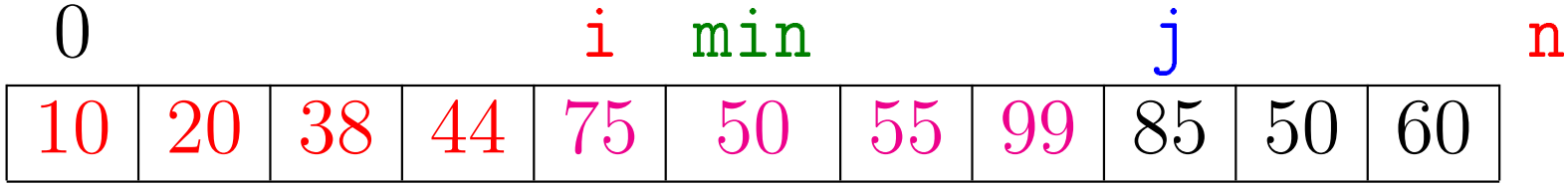


Supondo que a invariantes valem.
Correção do algoritmo é evidente.

No início da última iteração das linhas 1–5 tem-se que $\mathtt{i} = \mathtt{k}$.
Da invariante conclui-se que $\mathtt{v}[0\,..\,\mathtt{k}-1]$ é crescente.
e que $\mathtt{v}[\mathtt{k}-1] \le \mathtt{v}[\mathtt{k}\,..\,\mathtt{n}-1]$.

Mais invariantes

Na linha 1 vale que: (i1) $\texttt{v}[\texttt{i}] \leq \texttt{v}[\texttt{i}+1\,..\,\texttt{n}-1]$;
Na linha 3 vale que: (i2) $\texttt{v}[\texttt{min}] \leq \texttt{v}[\texttt{i}\,..\,\texttt{j}-1]$



invariantes (i1),(i2)
+ condição de parada do `for` da linha 3
+ troca linha 5 $\Rightarrow$ validade (i0)

Verifique!

Consumo de tempo

Se a execução de cada linha de código consome 1 unidade de tempo o consumo total é:

linha	todas as execuções da linha		
1	$=$	$\mathtt{k}+1$	$=$ $\mathrm{O}(\mathtt{k})$
2	$=$	$\mathtt{k}$	$=$ $\mathrm{O}(\mathtt{k})$
3	$=$	$\mathtt{n}+(\mathtt{n}-1)+\cdots+(\mathtt{n}-\mathtt{k}+1)$	$=$ $\mathrm{O}(\mathtt{kn})$
4	$=$	$(\mathtt{n}-1)+(\mathtt{n}-2)+\cdots+(\mathtt{n}-\mathtt{k}+1)$	$=$ $\mathrm{O}(\mathtt{kn})$
5	$=$	$\mathtt{k}$	$=$ $\mathrm{O}(\mathtt{k})$
total	$=$	$\mathrm{O}(2\mathtt{kn}+3\mathtt{k})$	$=$ $\mathrm{O}(\mathtt{kn})$

Conclusão

O consumo de tempo do algoritmo `kEsimo` no pior caso e no no melhor caso é proporcional a `kn`.

O consumo de tempo do algoritmo `kEsimo` é $O(\texttt{kn})$.

Conclusão

O consumo de tempo do algoritmo `kEsimo` no pior caso e no no melhor caso é proporcional a `kn`.

O consumo de tempo do algoritmo `kEsimo` é $O(\texttt{kn})$.

Consegue algo melhor usando ideias do heapsort?

Qual será o consumo de tempo em função de `n` e `k`?

Solução inspirada em quickSort()

Ao final o k-ésimo menor elemento está em v[k-1].
Primeira chamada: kEsimo(k,0,n,v);.

```
void kEsimo (int k, int p, int r, int v[])
{
1   int q = separa(p, r, v);
2   if (q == k-1) return;
3   if (q >= k  ) kEsimo(k, p  , q, v);
4   if (q <  k-1) kEsimo(k, q+1, r, v);
}
```

Consumo de tempo?

kEsimo: versão iterativa

```
void kEsimo (int k, int n, int v[]){
1   int p = 0;
2   int r = n;
3   int q = separa(p,r,v);
4   while (q != k-1) {
5       if (q >= k) r = q;
6       if (q <  k) p = q + 1;
7       q = separa(p, r, v);
    }
}
```

Exercícios

Qual o consumo de tempo no melhor caso do algoritmo `kEsimo` inspirado em `quickSort`?

Qual o consumo de tempo no pior caso do algoritmo `kEsimo` inspirado em `quickSort`?

Tente determinar experimentalmente o consumo de tempo do algoritmo `kEsimo` inspirado em `quickSort`.

Sob *hipóteses razoáveis* é possível mostrar que o consumo de tempo esperado do algoritmo `kEsimo` inspirado no `quickSort` é proporcional a `n`.

Arvores binárias

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

PF 14

http://www.ime.usp.br/~pf/algoritmos/aulas/bint.html

Árvore binárias

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

Cada nó tem três campos:

```
typedef struct celula Celula;
struct celula {
   int conteudo; /* tipo devia ser Item*/
   Celula *esq;
   Celula *dir;
};
typedef Celula No;
No x, y;
```

Pais e filhos

Os campos `esq` e `dir` dão estrutura à árvore.

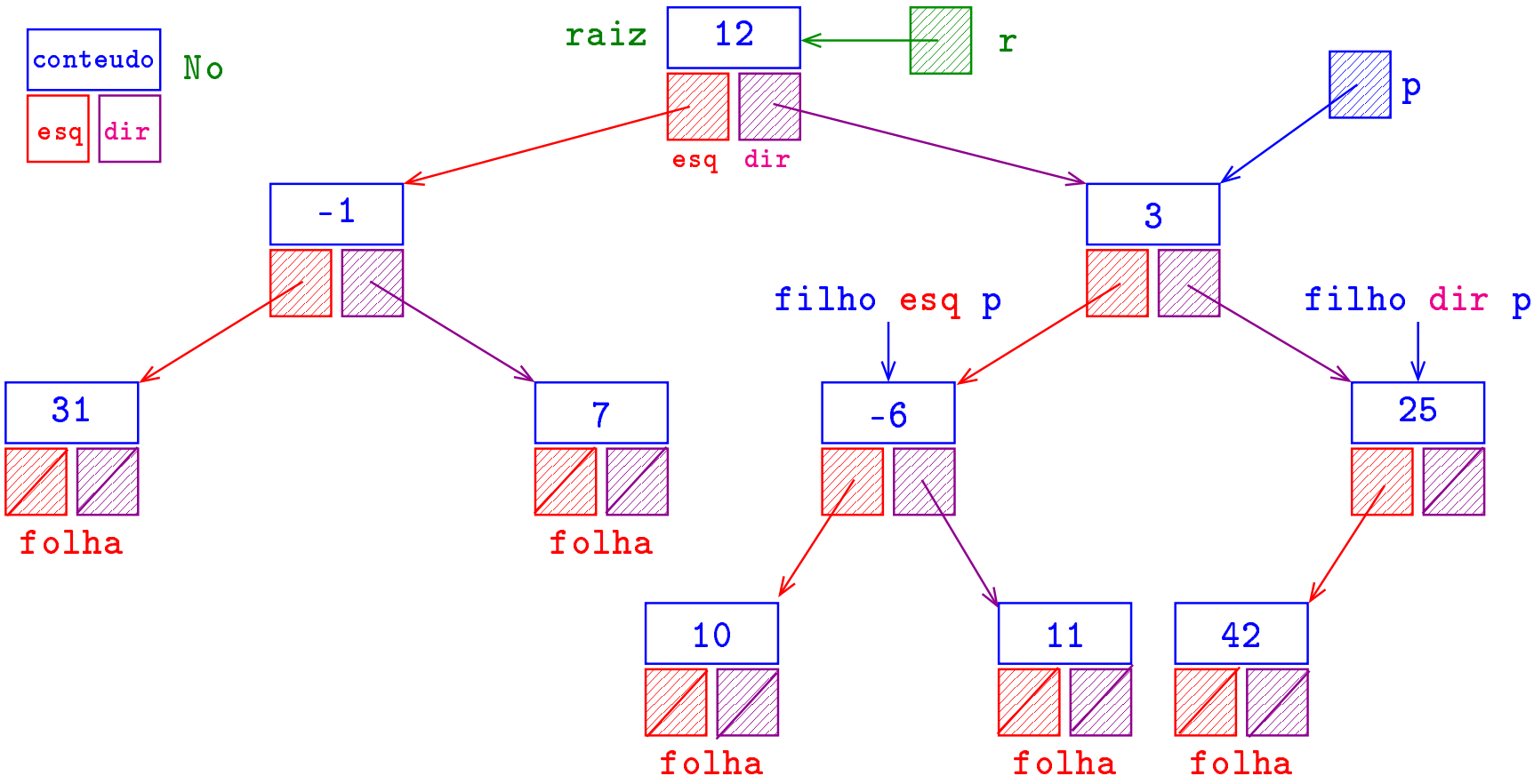
Se `x.esq == y`, então `y` é o **filho esquerdo** de `x`.

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

Assim, `x` é o **pai** de `y` se `x.esq == y` ou `x.dir == y`.

Uma **folha** é um nó sem filhos.
Ou seja, se `x.esq == NULL` e `x.dir == NULL` então `x` é uma folha

Ilustração de uma árvore binária



Árvores e subárvores

Suponha que `r` e `p` são (endereços de/ponteiros para) nós.

`p` é **descendente** de `r`
se `p` pode ser alcançado pela iteração dos comandos

```
p = p->esq;        p = p->dir;
```

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->esq` é a raiz da subárvore esquerda de `p` e `p->dir` é a raiz da subárvore direita de `p`.

Endereço de uma árvore

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

```
typedef No *Arvore;
Arvore r;
```

Endereço de uma árvore

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

```
typedef No *Arvore;
Arvore r;
```

Um objeto `r` é uma árvore binária se

- `r == NULL` ou
- `r->esq` e `r->dir` 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);

esquerda-raiz-direita

Visitamos

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

esquerda-raiz-direita

Visitamos

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

esquerda-raiz-direita

Visitamos

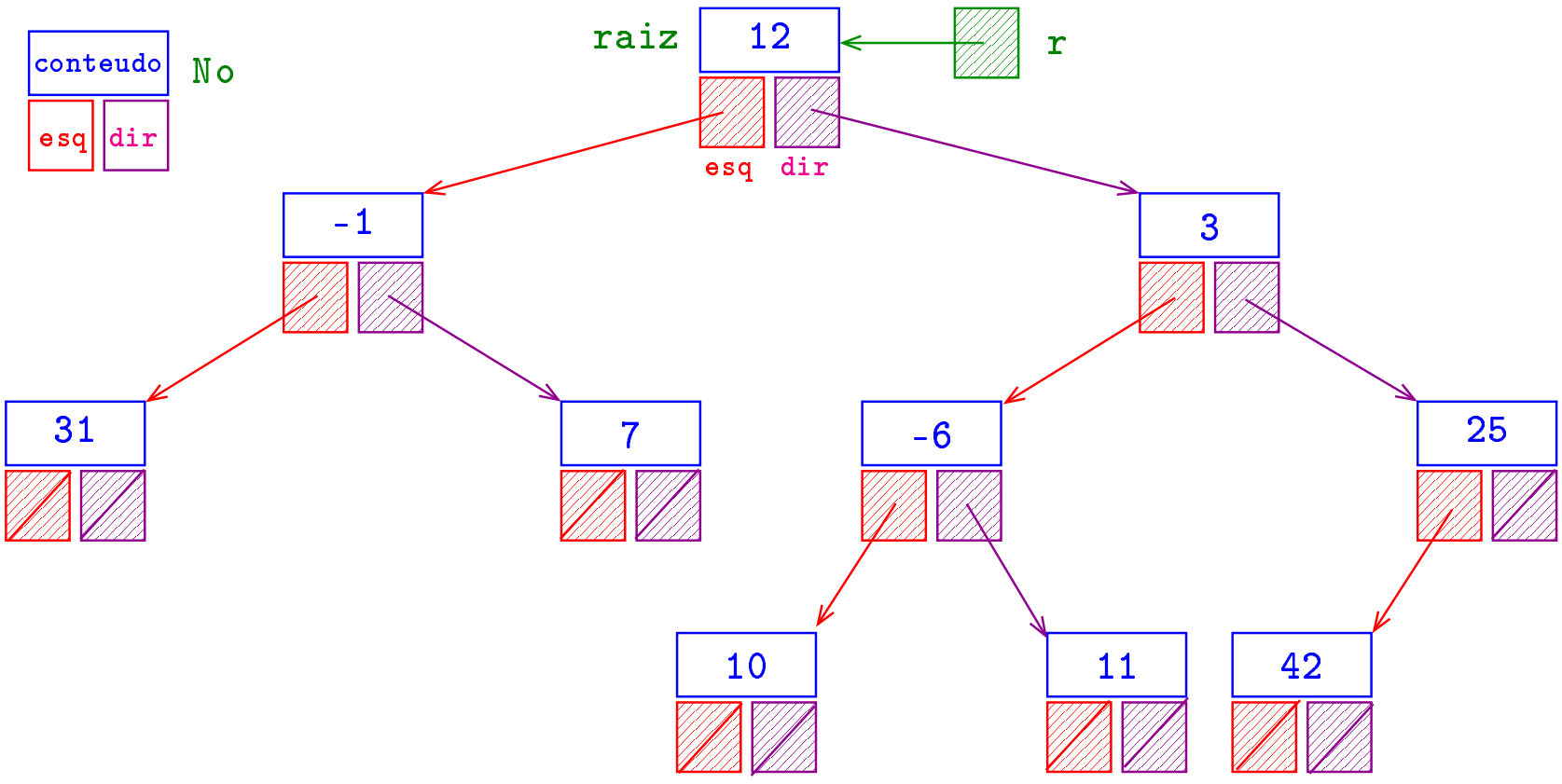
1. a subárvore esquerda da raiz, em ordem e-r-d;
2. depois a raiz;
3. a subárvore direita da raiz, em ordem e-r-d;

```
void inOrdem(Arvore r) {
   if (r != NULL) {
       inOrdem(r->esq);
       printf("%d\n", r->conteudo);
       inOrdem(r->dir);
   }
}
```

esquerda-raiz-direita versão iterativa

```
void inOrdem(Arvore r) {
   stackInit();
   while (r != NULL || !stackEmpty()) {
       if (r != NULL) {
           stackPush(r);
           r = r->esq;
       }
       else {
           r = stackPop();
           printf("%d\n", r->conteudo);
           r = r->dir;
       }
   }
}
```

Ilustração de percursos em árvores binárias

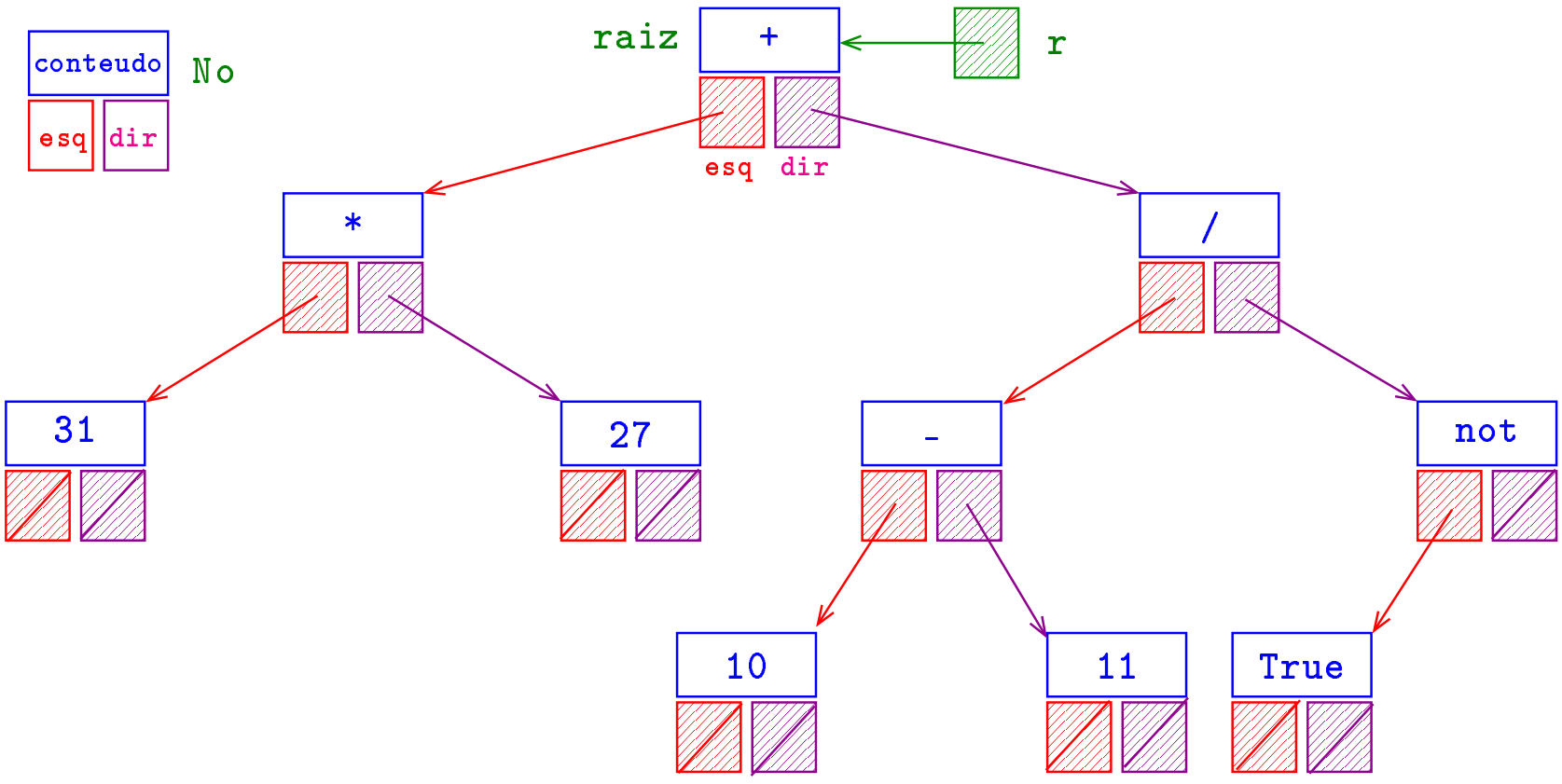


in-ordem (e-r-d): 31 -1 7 12 10 -6 11 3 42 25

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 / +`