MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 6

Filas priorizadas

Uma **fila priorizada** (ou **fila com prioridades**) é um ADT (*abstract data type*) que generaliza tanto a fila quanto a pilha.

Uma fila priorizada decrescente ou **PQ de máximo** é um ADT que manipula um conjunto de itens por meio de duas operações fundamentais:

- inserção de um novo item no conjunto e
- remoção de um item máximo.

Isso significa que uma fila priorizada manipula itens comparáveis (de um conjunto com ordem).

Interface para PQ-máximo

Arquivo	MaxPQ.h	
void	MaxPQInit()	cria uma PQ
void	MaxPQInsert(Item x)	insere x nesta PQ
Item	MaxPQMax()	devolve um máximo
Item	MaxPQDelMax()	remove e devolve um máximo de PQ
int	MaxPQSize()	número de itens
bool	MaxPQEmpty()	PQ está vazia?
void	MaxPQFree()	destroi esta PQ

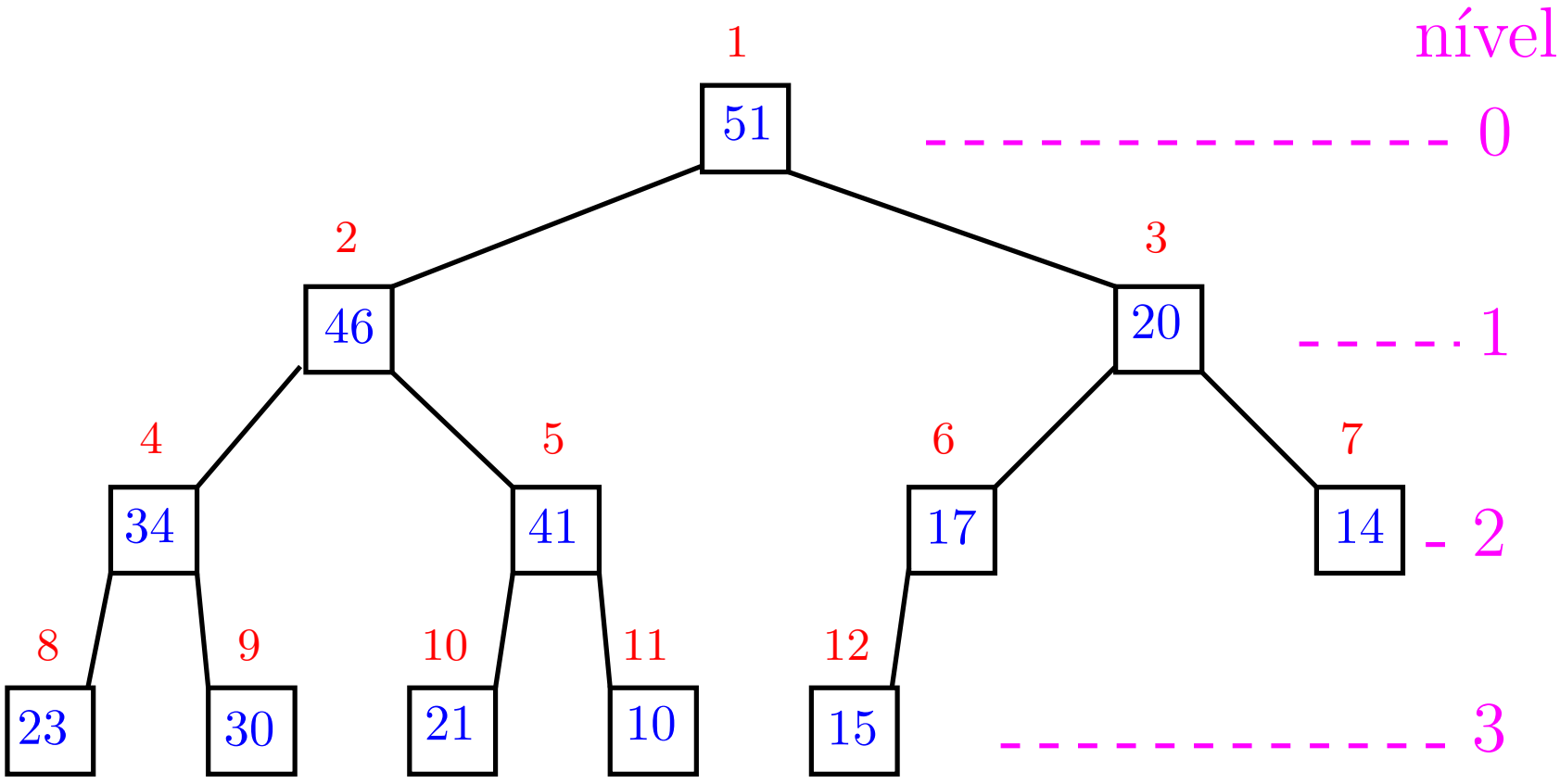
Implementações elementares

Pode-se implementar a classe `MaxPQ` ou `MinPQ` com

- vetor de itens não-ordenados;
- vetor de itens ordenados;
- lista ligada de itens ordenados;
- lista ligada de itens não ordenados.

Em todas essas implementações,
o consumo de tempo pode ser
proporcional ao número `n` de itens na fila.

Implementação com *binary heaps*



Implementação com *binary heaps*

A estrutura se apoia nas
operações administrativas `sink()` e `swim()`.

O consumo de tempo das operações
em uma fila priorizada implementada
como um binary heap é $\mathrm{O}(\lg \mathtt{n})$,
onde `n` é o número de itens na fila.

Construção de *binary heaps*

O consumo de tempo para
construir *dinamicamente (online)*
um binary heap com `n` itens é $\mathrm{O}(\mathtt{n} \lg \mathtt{n})$
(construção se apoia em `swim()`).

O consumo de tempo para
construir *estaticamente (offline)*
um binary heap com `n` itens é $\mathrm{O}(\mathtt{n})$
(construção se apoia na operação `sink()`).

Hmm

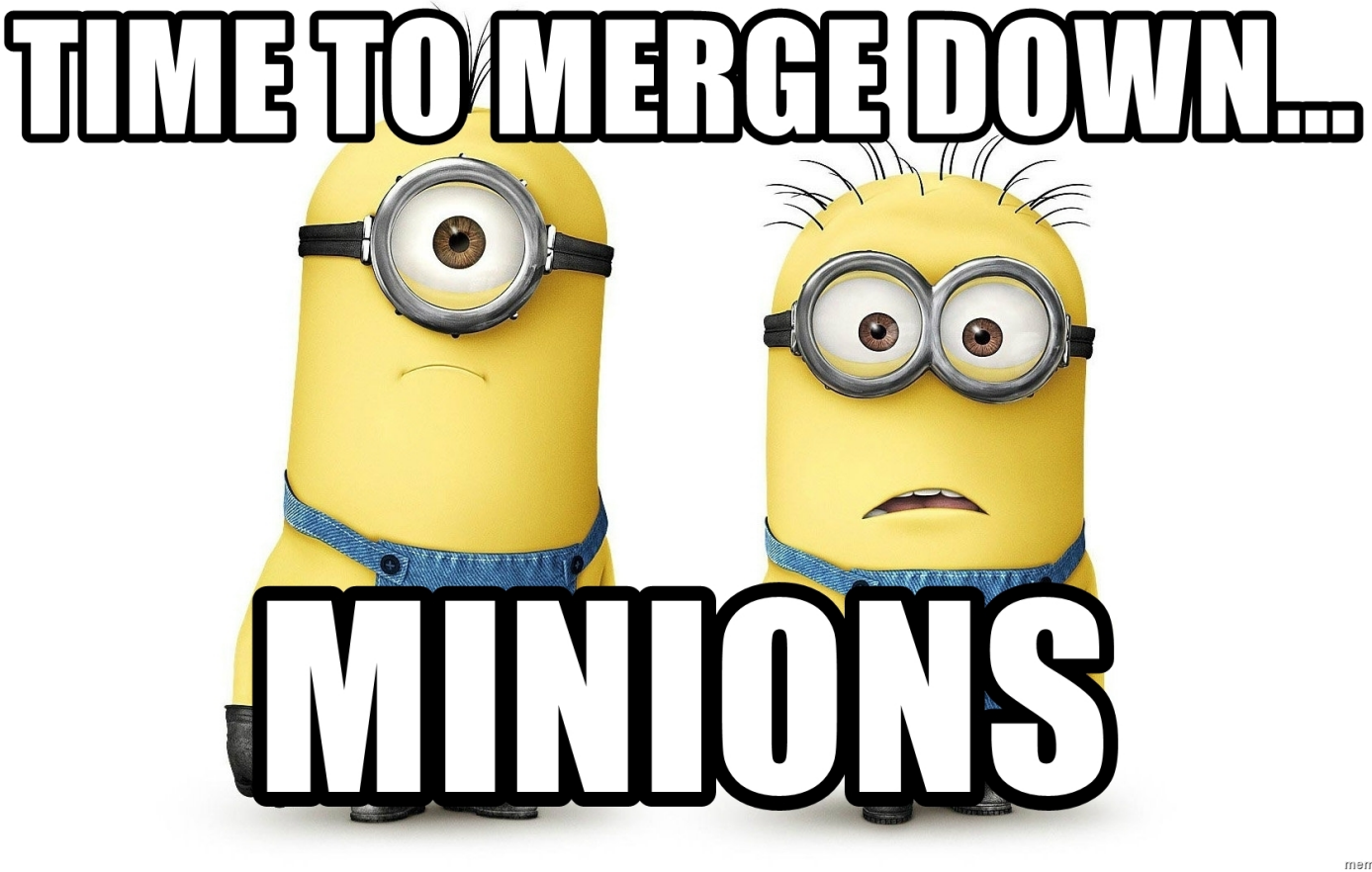
O slide anterior sugere que às vezes pode valer a pena sermos preguiçosos (*lazy data structure*) e aguardarmos mais itens chegarem dinamicamente, antes de montarmos o heap.

Pode não valer a pena sermos ansiosos (*eager data structure*).

Pelo menos do ponto de vista de consumo de tempo amortizado, pode valer a pena sermos preguiçosos.

AULA 7

Mergeable heaps



Fonte: https://memegenerator.net

Leftist heap

TAOCP 5.2.3 Vol. 3

Além das operações usuais de uma fila com prioridades, permite que a união ("merge") de duas filas seja feita eficientemente.

Estrutura simples, ultrapassada por outras, como *binomial heaps* (CLRS 19) e *fibonacci heaps* (CLRS 20).

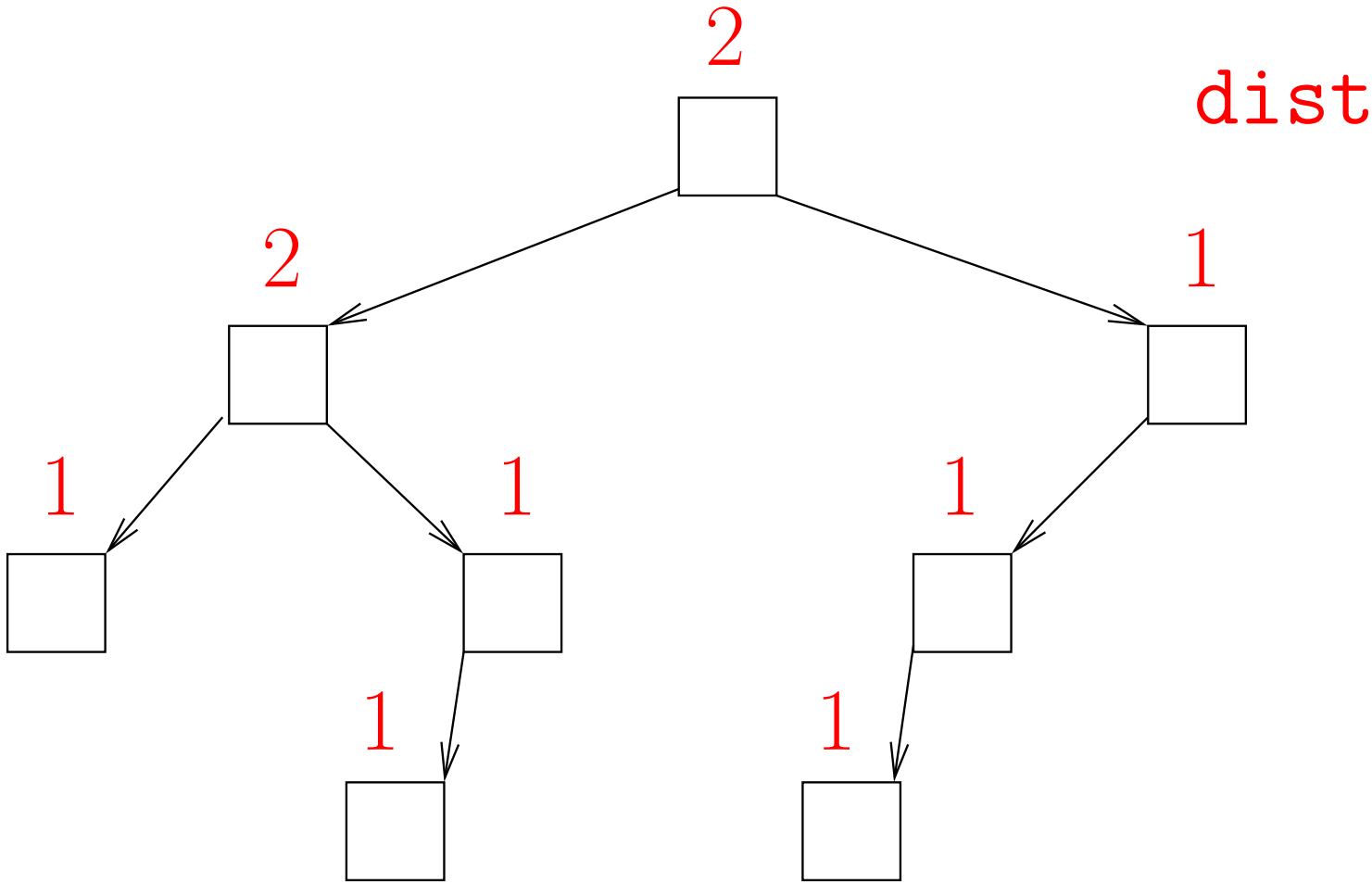
Árvores esquerdistas

Cada nó x tem quatro campos:

1. esq[x]: filho esquerdo de x;
2. dir[x]: filho direito de x;
3. dist[x]: menor comprimento de um caminho de x a NULL.

```
dist (x)
1   se x = NULL
2      então devolva 0
3      senão devolva
          1 + min{dist(esq[x]), dist(dir[x])}
```

Exemplo



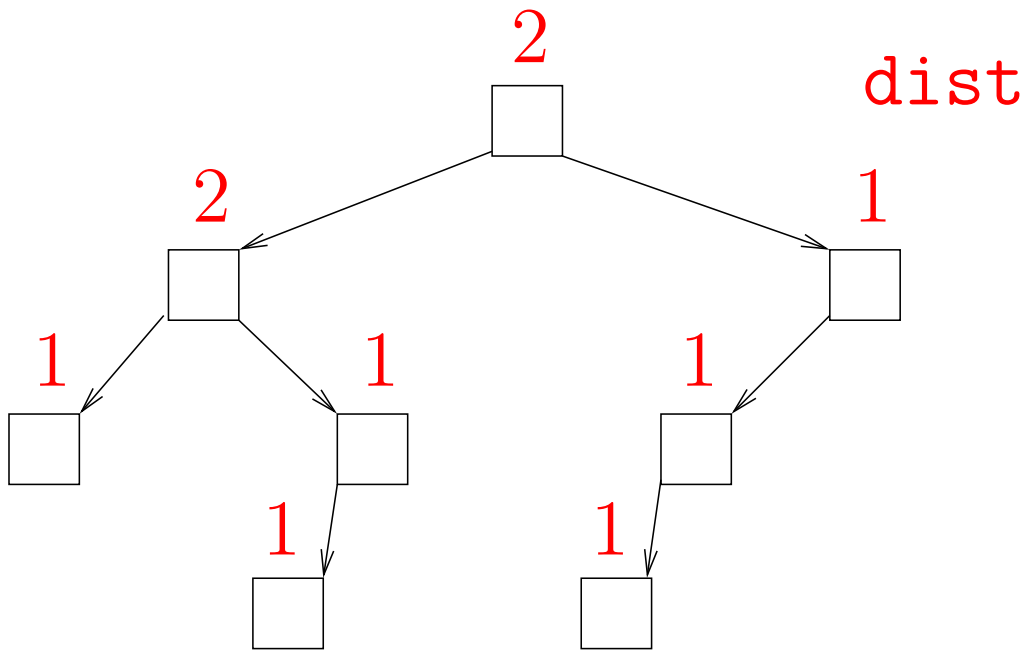
Árvores esquerdistas

Uma árvore é esquerdista se

$$\texttt{dist}[\texttt{esq}[\texttt{x}]] \geq \texttt{dist}[\texttt{dir}[\texttt{x}]]$$

para todo nó $\texttt{x}$ ($\texttt{dist}[\texttt{NULL}] = 0$).

Exemplo 1:



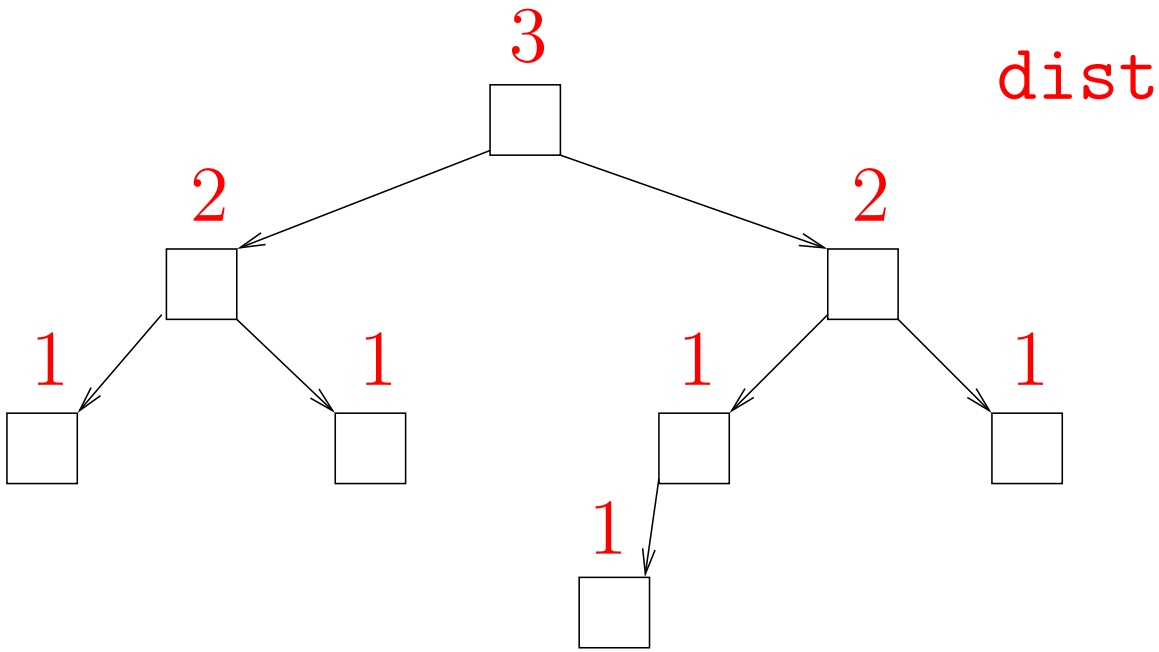
Árvores esquerdistas

Uma árvore é esquerdista se

$$\texttt{dist}[\texttt{esq}[\texttt{x}]] \geq \texttt{dist}[\texttt{dir}[\texttt{x}]]$$

para todo nó $\texttt{x}$ ($\texttt{dist}[\texttt{NULL}] = 0$).

Exemplo 2:



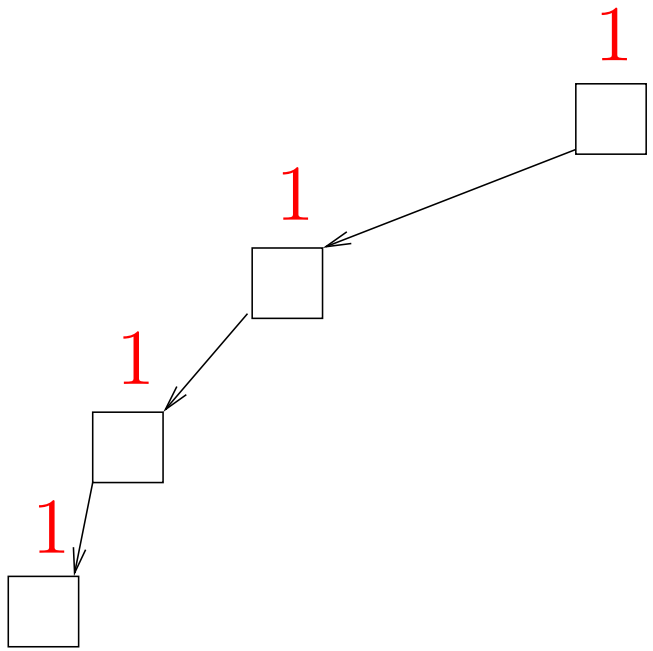
Árvores esquerdistas

Uma árvore é esquerdista se

$$\texttt{dist}[\texttt{esq}[\texttt{x}]] \geq \texttt{dist}[\texttt{dir}[\texttt{x}]]$$

para todo nó $\texttt{x}$ ($\texttt{dist}[\texttt{NULL}] = 0$).

Exemplo 3:



dist

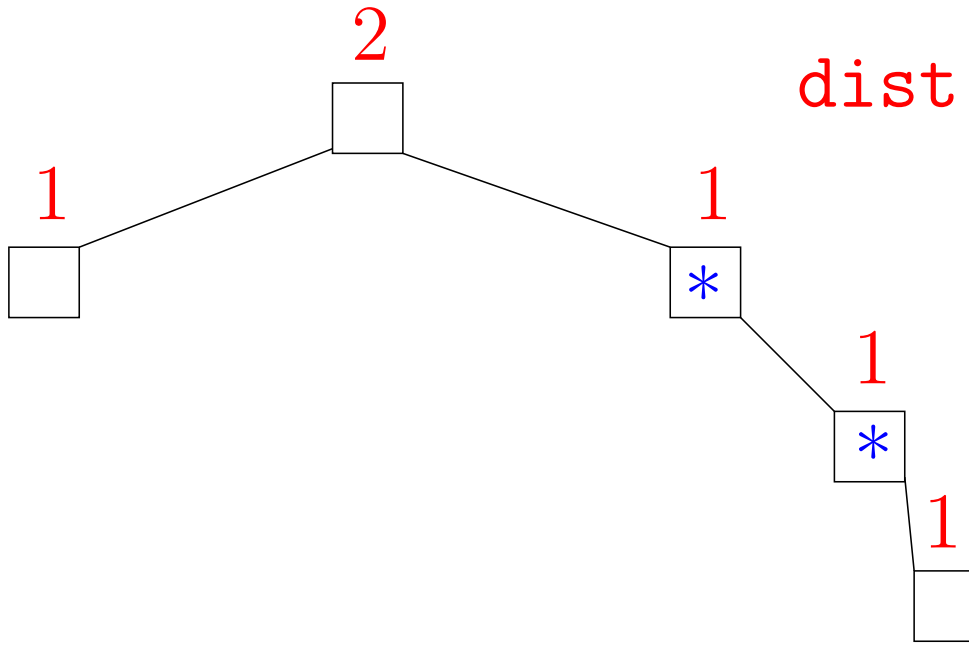
Árvores esquerdistas

Uma árvore é esquerdista se

$$\texttt{dist}[\texttt{esq}[\texttt{x}]] \geq \texttt{dist}[\texttt{dir}[\texttt{x}]]$$

para todo nó x ($\texttt{dist}[\texttt{NULL}] = 0$).

Exemplo 4: árvore não-esquerdista



Caminho direitista

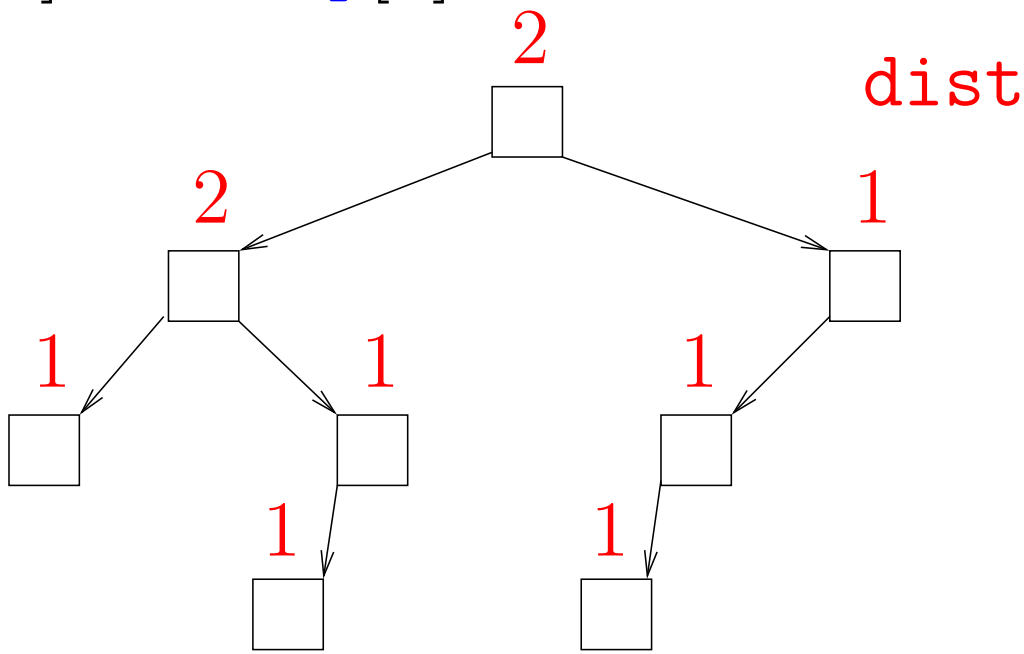
O **caminho direitista** de um nó x é a sequência

$$\langle x, \texttt{dir}[x], \texttt{dir}[\texttt{dir}[x]], \ldots, \texttt{NULL} \rangle.$$

dcomp[x] := núm. de nós no caminho direitista de x

tam[x] := número de nós na árvore de raiz x

Se x é um nó de uma árvore esquerdista,
então dist[x] = dcomp[x].



Fato estrutural

$\texttt{tam}[\texttt{x}] :=$ número de nós na árvore de raiz $\texttt{x}$

Se $\texttt{x}$ é um nó de uma árvore esquerdista, então

$$\texttt{tam}[\texttt{x}] \geq 2^{\texttt{dist}[\texttt{x}]} - 1.$$

Fato estrutural

$\texttt{tam}[\texttt{x}]$:= número de nós na árvore de raiz $\texttt{x}$

Se $\texttt{x}$ é um nó de uma árvore esquerdista, então

$$\texttt{tam}[\texttt{x}] \geq 2^{\texttt{dist}[\texttt{x}]} - 1.$$

Prova: Seja $\texttt{d} := \texttt{dist}[\texttt{x}]$.

Se $\texttt{d} = 1$, então $\texttt{tam}[\texttt{x}] \geq 1 = 2^{\texttt{d}} - 1$.

Suponha que $\texttt{d} \geq 2$ e que
a desigualdade vale para $\texttt{d} - 1$.

Temos que $\texttt{dist}[\texttt{dir}[\texttt{x}]] = \texttt{d} - 1$ e que existe um nó $\texttt{y}$ na árvore de raiz $\texttt{esq}[\texttt{x}]$ tal que $\texttt{dist}[\texttt{y}] = \texttt{d} - 1$.

Fato estrutural

$\texttt{tam}[\texttt{x}] :=$ número de nós na árvore de raiz $\texttt{x}$

Fato 2. Se $\texttt{x}$ é um nó de uma árvore esquerdista, então

$$\texttt{tam}[\texttt{x}] \geq 2^{\texttt{dist}[\texttt{x}]} - 1.$$

Prova: (continuação)
Logo,

$$\begin{aligned}
\texttt{tam}[\texttt{x}] &= \texttt{tam}[\texttt{esq}[\texttt{x}]] + \texttt{tam}[\texttt{dir}[\texttt{x}]] + 1 \\
&\geq \texttt{tam}[\texttt{y}] + \texttt{tam}[\texttt{dir}[\texttt{x}]] + 1 \\
&\overset{\text{hi}}{\geq} 2^{\texttt{d}-1} - 1 + 2^{\texttt{d}-1} - 1 + 1 \\
&= 2^{\texttt{d}} - 1.
\end{aligned}$$

Consequência

Se x é um nó de uma árvore esquerdista, então
$\texttt{dist}[\texttt{x}] = \texttt{dcomp}[\texttt{x}] \leq \lfloor \lg(\texttt{tam}[\texttt{x}]+1) \rfloor = \mathrm{O}(\lg \texttt{tam}[\texttt{x}])$.

Em particular:

Se x é raiz de uma
árvore esquerdista com m nós,

$$\texttt{dist}[\texttt{x}] = \texttt{dcomp}[\texttt{x}] \leq \lfloor \lg(\texttt{m}+1) \rfloor = \mathrm{O}(\lg \texttt{m}).$$

Prova: $\texttt{m} \geq 2^{\texttt{d}} - 1 \Rightarrow \texttt{m}+1 \geq 2^{\texttt{d}} \Rightarrow \lfloor \lg(\texttt{m}+1) \rfloor \geq \texttt{d}$.

Heap esquerdista

$\texttt{H}$:= árvore

$\texttt{raiz}[\texttt{H}]$:= raiz de $\texttt{H}$

$\texttt{prior}[\texttt{x}]$:= prioridade do nó $\texttt{x}$

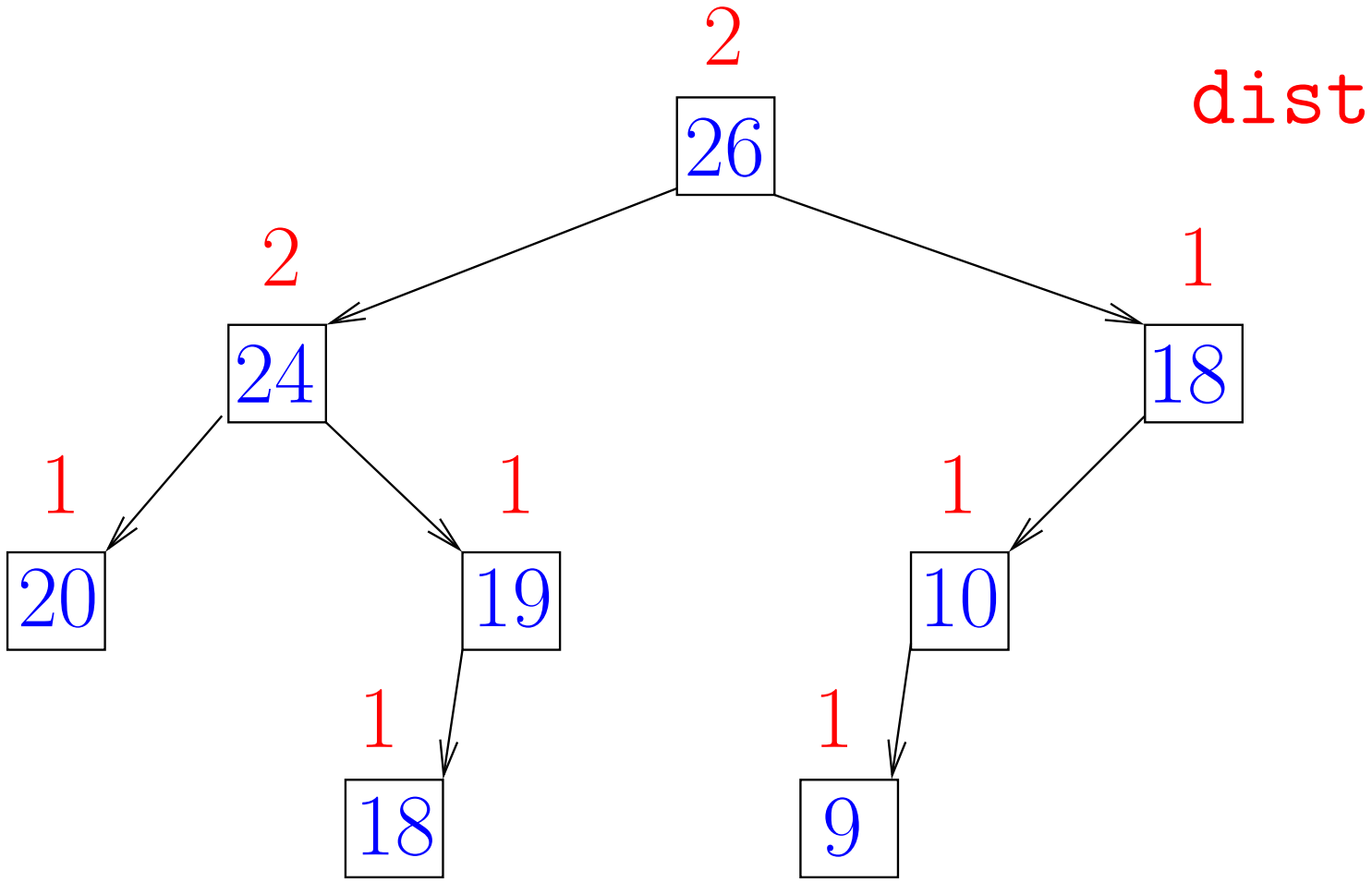
$\texttt{pai}[\texttt{x}]$:= pai do nó $\texttt{x}$

Um **heap esquerdista** $\texttt{H}$ é
uma árvore esquerdista que satisfaz

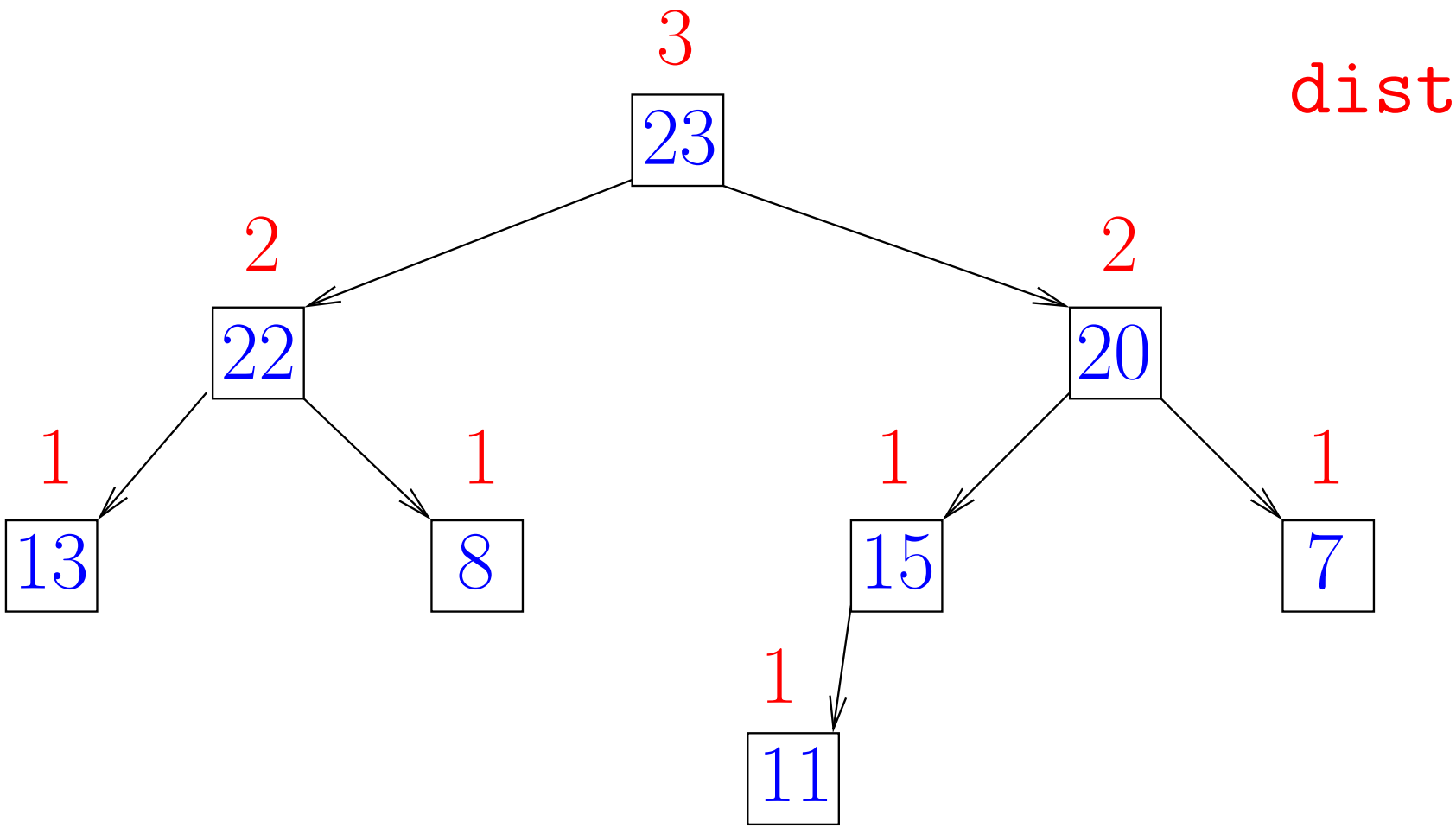
$$\texttt{prior}[\texttt{pai}[\texttt{x}]] \geq \texttt{prior}[\texttt{x}]$$

para todo nó $\texttt{x} \neq \texttt{raiz}[\texttt{H}]$.

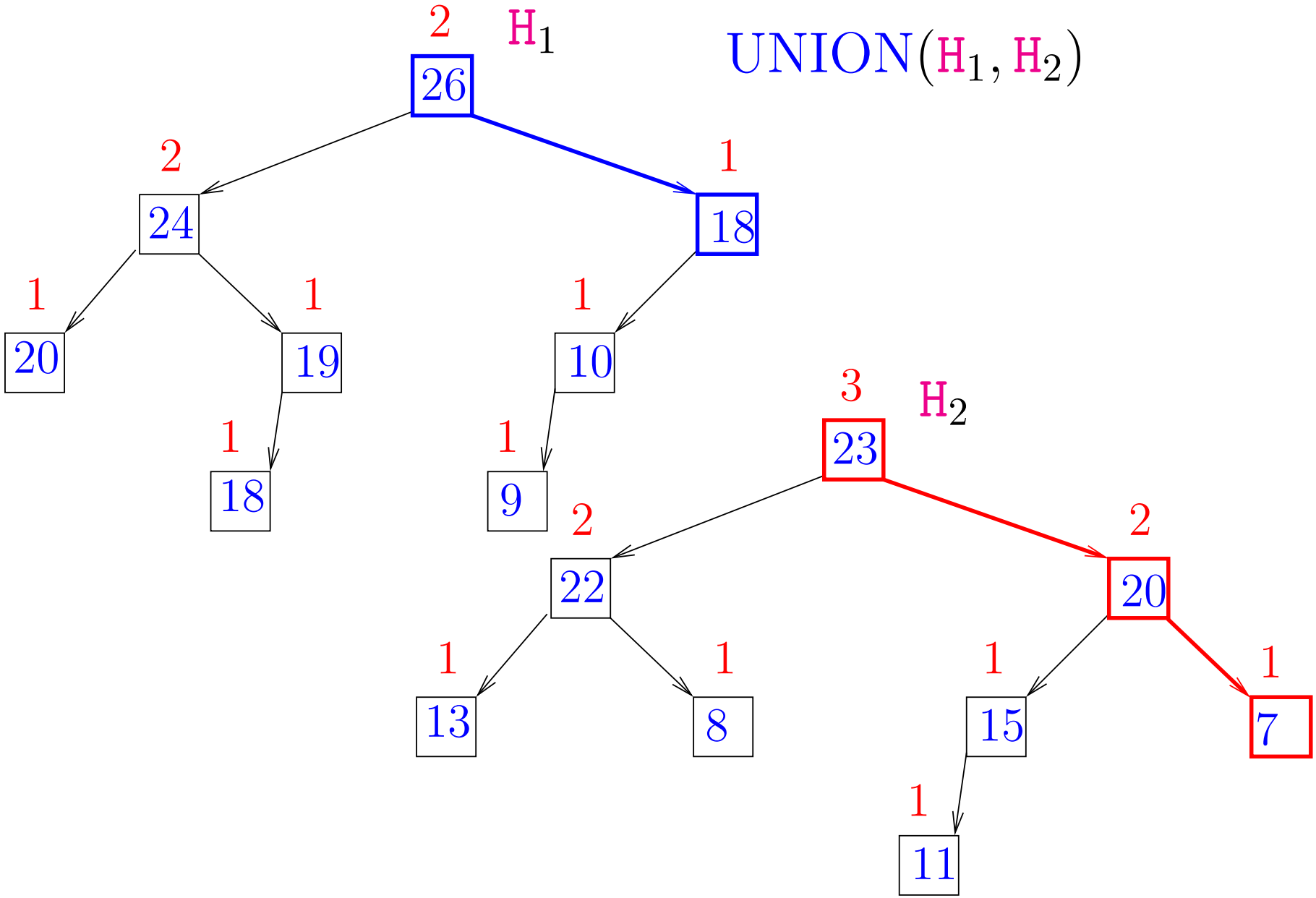
Heap esquerdista



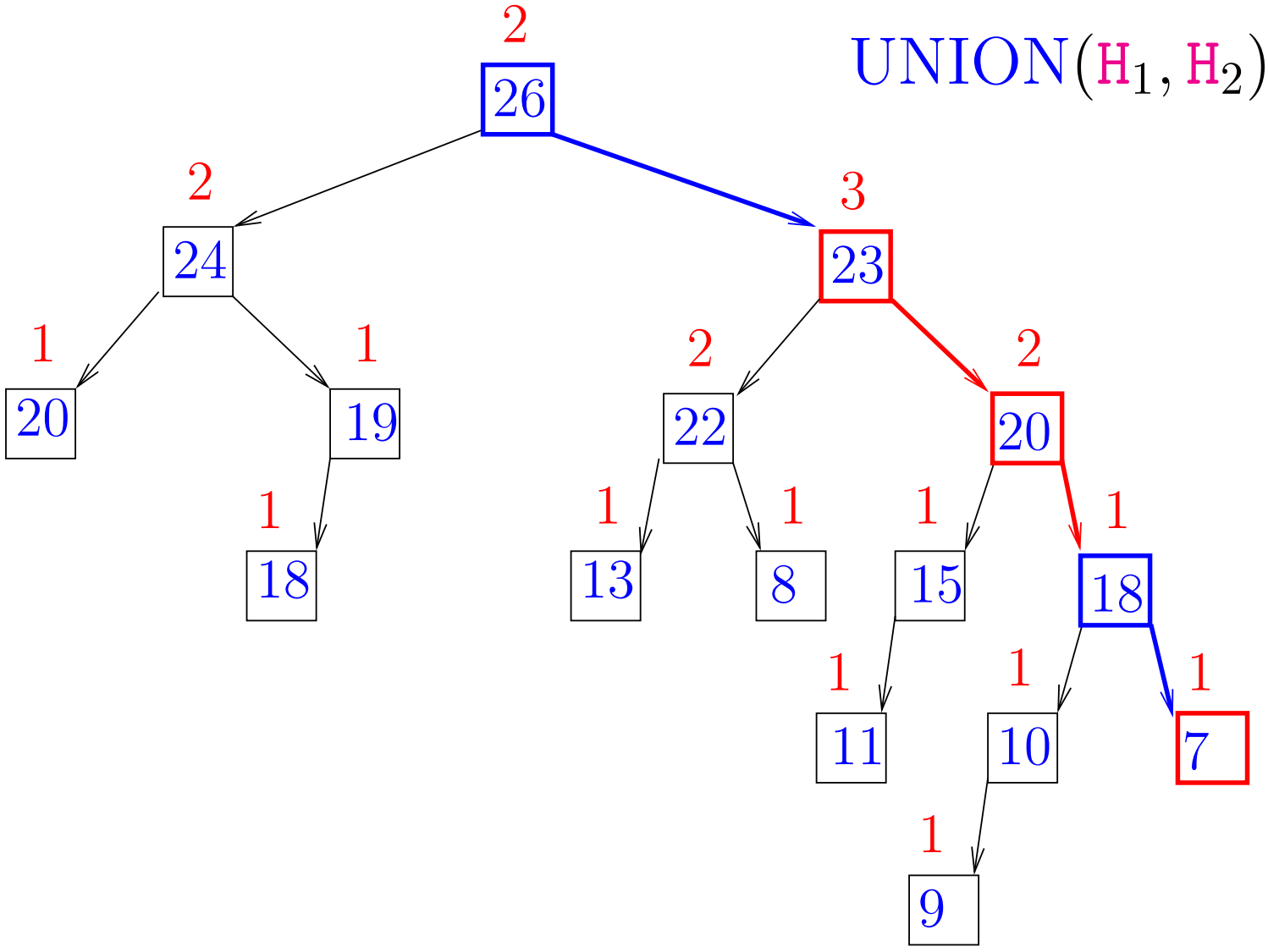
Heap esquerdista



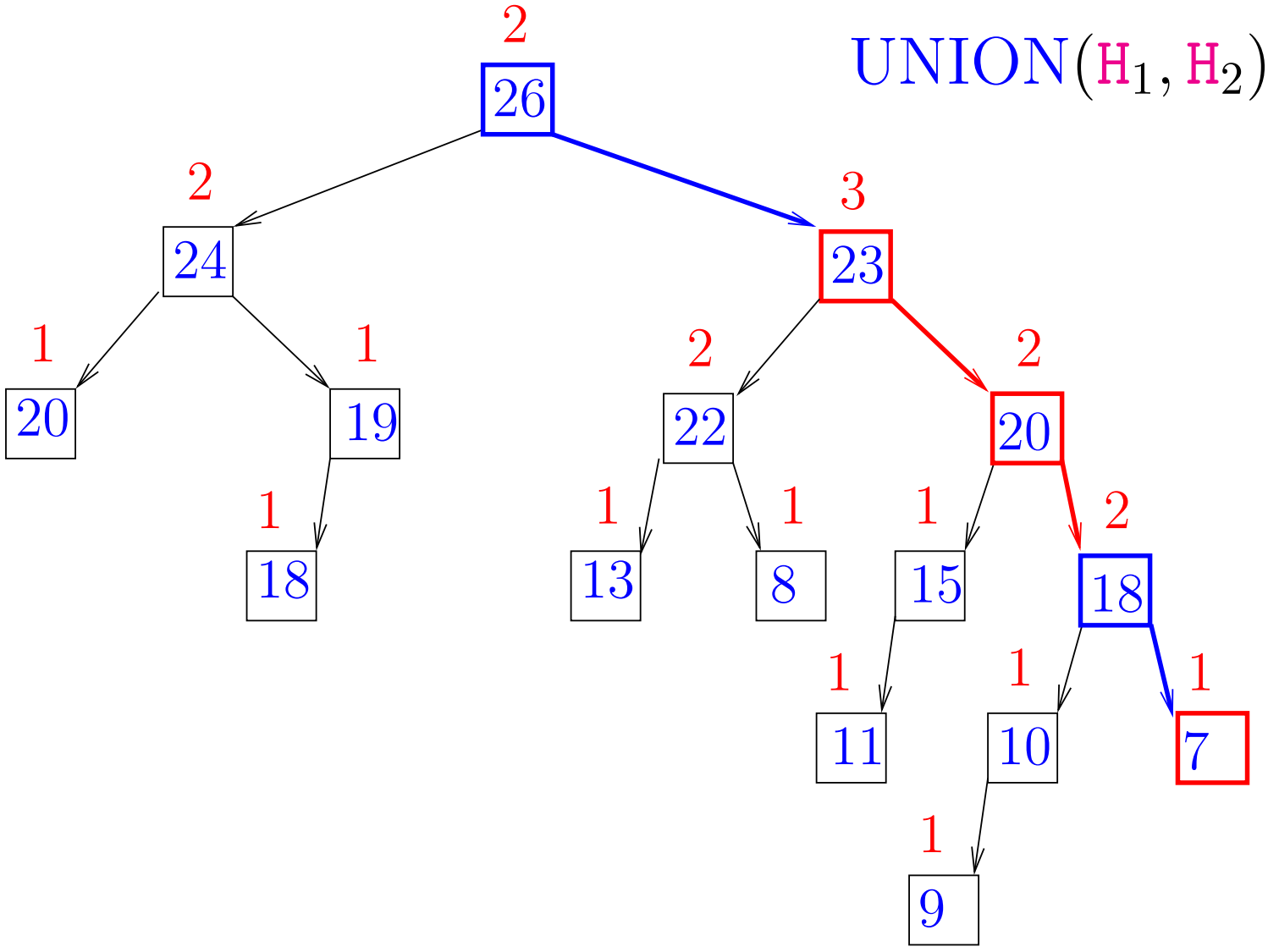
Rotina básica de manipulação



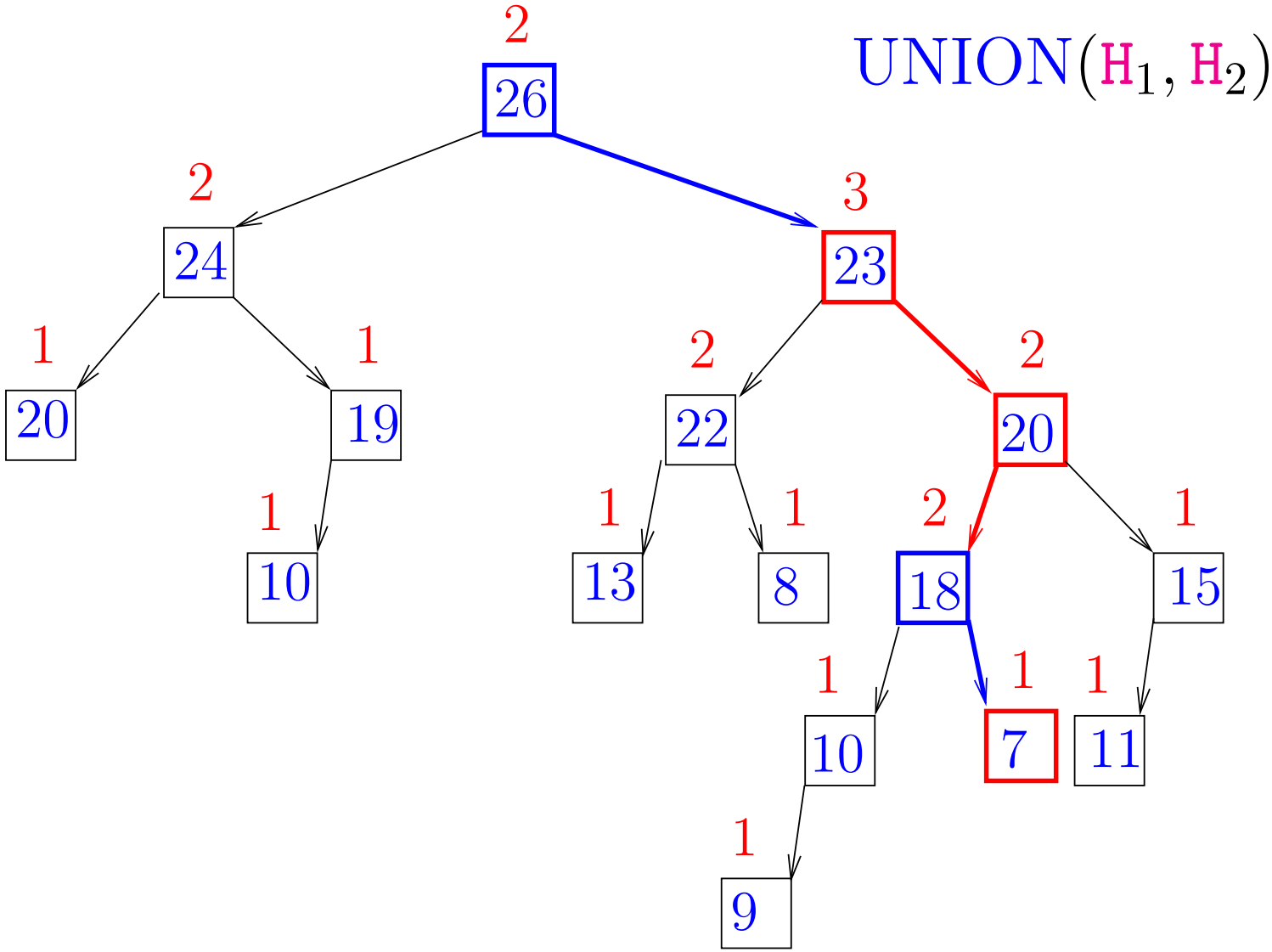
Rotina básica de manipulação



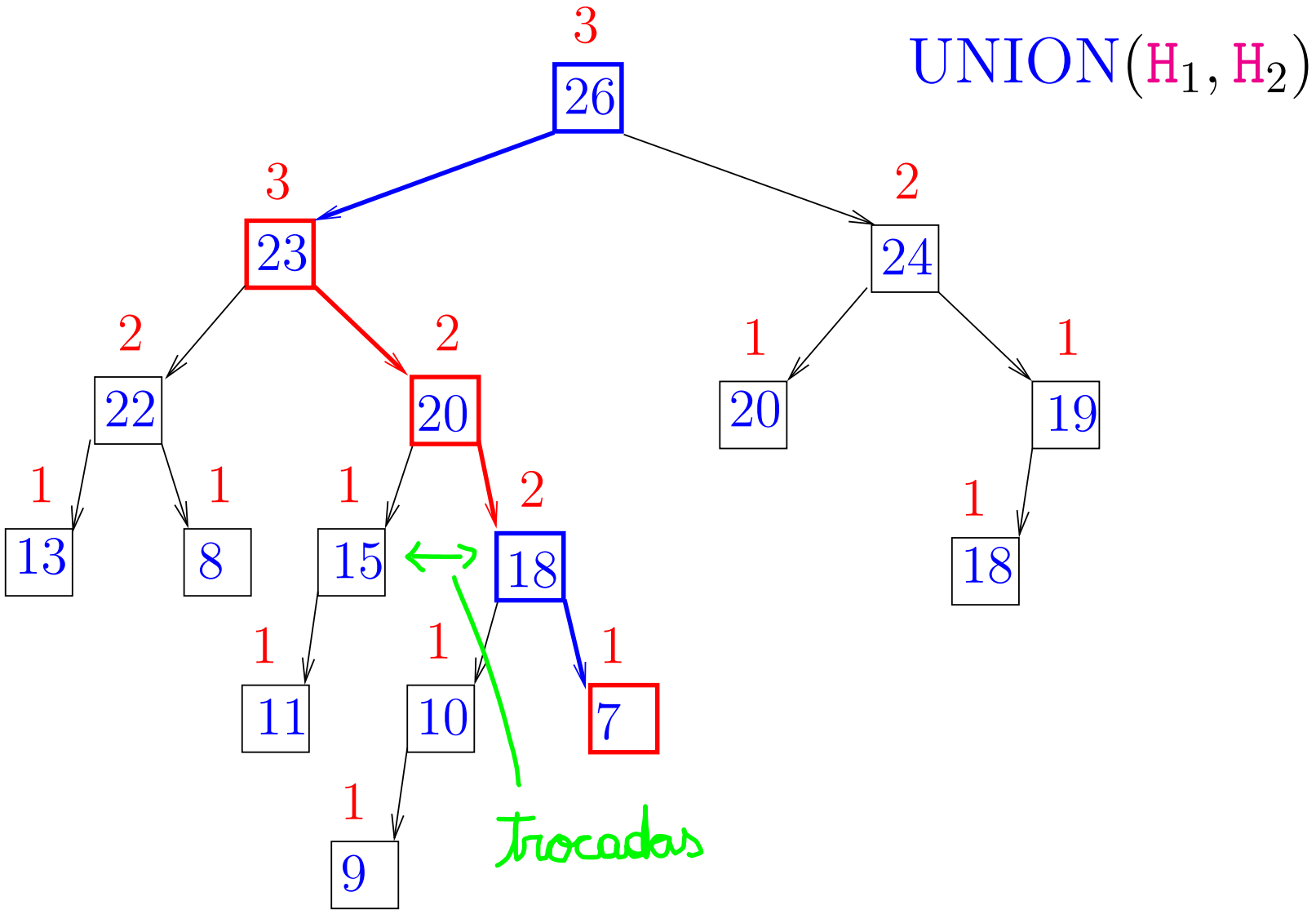
Rotina básica de manipulação



Rotina básica de manipulação



Rotina básica de manipulação



Rotina básica de manipulação

```
LEFTIST-HEAP-UNION (H1, H2)
 1    se raiz[H1] = NULL então devolva H2
 2    se raiz[H2] = NULL então devolva H1
 3    se prior[raiz[H1]] < prior[raiz[H2]]
 3        então H1 ↔ H2
 4    x1 ← raiz[H1]   x2 ← raiz[H2]
 5    se esq[x1] = NULL então esq[x1] ← x2
 6    senão H' ← MakeLeftistHeap()
 7          raiz[H'] ← dir[x1]
 8          H' ← LeftistHeapMerge(H', H2)
 9          dir[x1] ← raiz[H']
10          se dist[esq[x1]] < dist[dir[x1]]
11              então esq[x1] ↔ dir[x2]
12          dist[x1] ← dist[dir[x1]] + 1
13   devolva H1
```

Consumo de tempo

O consumo de tempo do algoritmo `LeftistHeapMerge` no pior caso é proporcional a

$$\texttt{dcomp}(\texttt{raiz}[\texttt{H}_1]) + \texttt{dcomp}(\texttt{raiz}[\texttt{H}_2]) = \mathrm{O}(\lg \texttt{m})$$

onde $\texttt{m} = \texttt{tam}(\texttt{raiz}[\texttt{H}_1]) + \texttt{tam}(\texttt{raiz}[\texttt{H}_2])$

O consumo de tempo do algoritmo `LeftistHeapMerge` é $\mathrm{O}(\lg \texttt{m})$.

Fila com heap esquerdista

Rotina que insere um nó x
em um heap esquerdista H.

A rotina supõe que prior[x] já foi definido.

```
LEFTIST-HEAP-INSERT (H, x)
1   H′ ← MakeLeftistHeap()
2   esq[x] ← NULL
3   dir[x] ← NULL
4   dist[x] ← 1
5   raiz[H′] ← x
6   H ← LeftistHeapMerge(H, H′)
```

Consome tempo $O(\lg m)$.

Fila com heap esquerdista

Rotina que remove e devolve o nó de maior prioridade de um heap esquerdista $\mathtt{H}$.

```
LEFTIST-HEAP-EXTRACT (H)
1   x ← raiz[H]
2   H′ ← MakeLeftistHeap()
3   raiz[H′] ← esq[x]
4   raiz[H] ← dir[x]
5   H ← LeftistHeapMerge(H, H′)
6   devolva x
```

Consome tempo $\mathrm{O}(\lg \mathtt{m})$.

Cliente MinPQ: TopM

```
int main(int argc, char *argv) {
  int M = atoi(argv[1]); Transaction *t;
  MinPQInit(M+1);
  while ((t = readT()) != NULL) {
     MinPQInsert(t); /* Mantemos M maiores transacoes na PQ */
     if (MinPQSize() > M) {
        t = MinPQDelMin();   freeT(t);
     }
  }
  stackInit(); /* Pilha para imprimir da maior para a menor */
  while (!MinPQEmpty()) stackPush(MinPQDelMin());
  while (!stackEmpty()) {
     t = stackPop();   printT(t);   freeT(t);
  }
  stackFree();   MinPQFree();
}
```

Interface para PQ-mínimo

Arquivo	MinPQ.h	
void	MinPQInit(int)	cria uma PQ
void	MinPQInsert(Item x)	insere x nesta PQ
Item	MinPQMin()	devolve um mínimo
Item	MinPQDelMin()	remove e devolve um mínimo de PQ
int	MinPQSize()	número de itens
bool	MinPQEmpty()	PQ está vazia?
void	MinPQFree()	destroi esta PQ

Vamos mostrar uma implementação dessa interface com árvores esquerdistas.

LMinPQ: struct node, Link e newNode

Cada nó da árvore esquerdista tem quatro campos:

```
typedef struct node *Link;
struct node {
    Transaction *item;
    Link left, right;
    int dist;
};
Link newNode(Transaction *item,
             Link left, Link right, int dist) {
    Link p = mallocSafe(sizeof(*p));
    p->item = item;
    p->left = left;   p->right = right;
    p->dist = dist;
    return p;
}
```

Arquivo LMinPQ.c: esqueleto

```
#include "MinPQ.h"

static Link root;
static int n;

void MinPQInit(int max) {...}
void MinPQInsert(Item item) {...}
Item MinPQMin() {...}
Item MinPQDelMin() {...}
bool MinPQEmpty() {...}
int  MinPQSize() {...}
void MinPQFree() {...}

static Link merge(Link r1, Link r2) {...}
```

LMinPQ: init(), empty() e size()

```
void MinPQInit(int m) {
   root = NULL;
   n = 0;
}

int MinPQSize() {
   return n;
}

bool MinPQEmpty() {
   return n == 0;
}
```

LMinPQ: insert() e delMin()

```
void MinPQInsert(Item item) {
   Link s = newNode(item, NULL, NULL, 1);
   root = merge(root, s);
   n++;
}
```

LMinPQ: insert() e delMin()

```
void MinPQInsert(Item item) {
   Link s = newNode(item, NULL, NULL, 1);
   root = merge(root, s);
   n++;
}
Item MinPQDelMin() {
   Item item = root->item;
   Link s = root;
   root = merge(root->left, root->right);
   freeNode(s);
   n--;
   return item;
}
```

LMinPQ: merge()

merge(r1, r2) *essencialmente* intercala as listas encadeadas dos caminhos direitistas de r1 e r2:

```
static Link merge(Link r1, Link r2) {
    if (r1 == null) return r2;
    if (r2 == null) return r1;
    if(less(r2->item,r1->item)) {
        Link t = r1;  r1 = r2;  r2 = t;
    }
    r1->right = merge(r1->right, r2);
    return r1;
}
```

O *essencialmente* é devido ao fato de ser necessário acertamos os campos dist durante a volta da recursão, como é feito mais adiante.

LMinPQ: merge()

```
static Link merge(Link r1, Link r2) {
   Link t;
   if (r1 == null) return r2;
   if (r2 == null) return r1;
   if(less(r2->item,r1->item)) {
      t = r1;  r1 = r2;  r2 = t;
   }
   /* r1 aponta para o menor item */
   if (r1->left == NULL) r1->left = r2;
   else {
```

LMinPQ: merge()

```
  else {
     r1->right = merge(r1->right, r2);
     if (r1->left->dist < r1->right->dist) {
       t = r1->left;
       r1->left = r1->right;
       r1->right = t;
     }
     r1->dist = r1->right->dist + 1;
  }
  return r1;
}
```

Conclusão

O consumo de tempo das operações em uma fila priorizada implementada com `LMinPQ` é $O(\lg n)$, onde `n` é o número de itens na fila.

O consumo de tempo dos métodos da classe `LMinPQ` é $O(\lg n)$, onde `n` é o número de itens na fila.

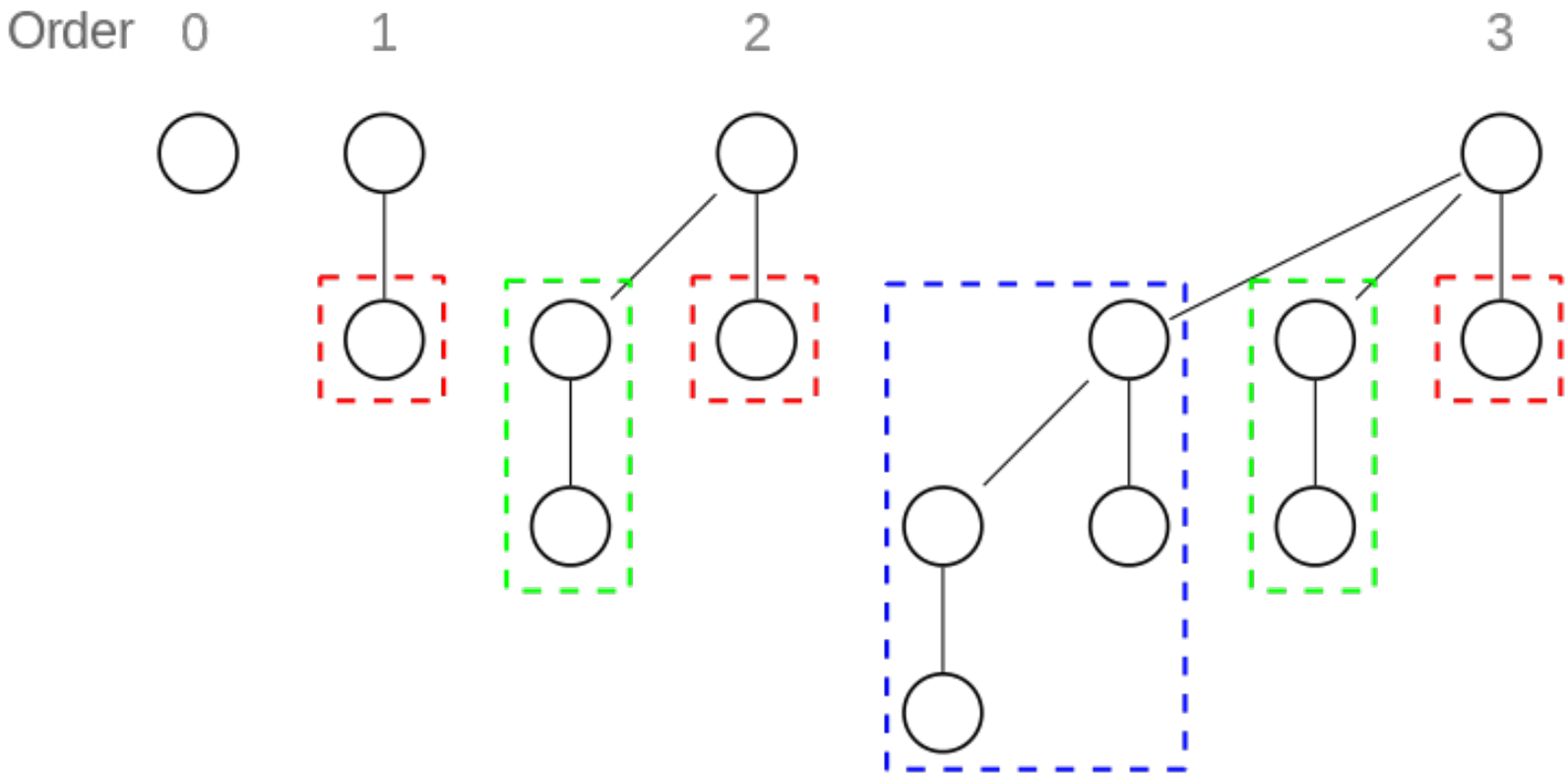
Conclusão

O consumo de tempo das operações em uma fila priorizada implementada com `LMinPQ` é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na fila.

O consumo de tempo dos métodos da classe `LMinPQ` é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na fila.

Ademais, podemos adicionar uma operação extra à biblioteca, de união de duas filas priorizadas.

Binomial heaps



Fontes: Wikipedia,

Foundations of Data Science

CLRS 19

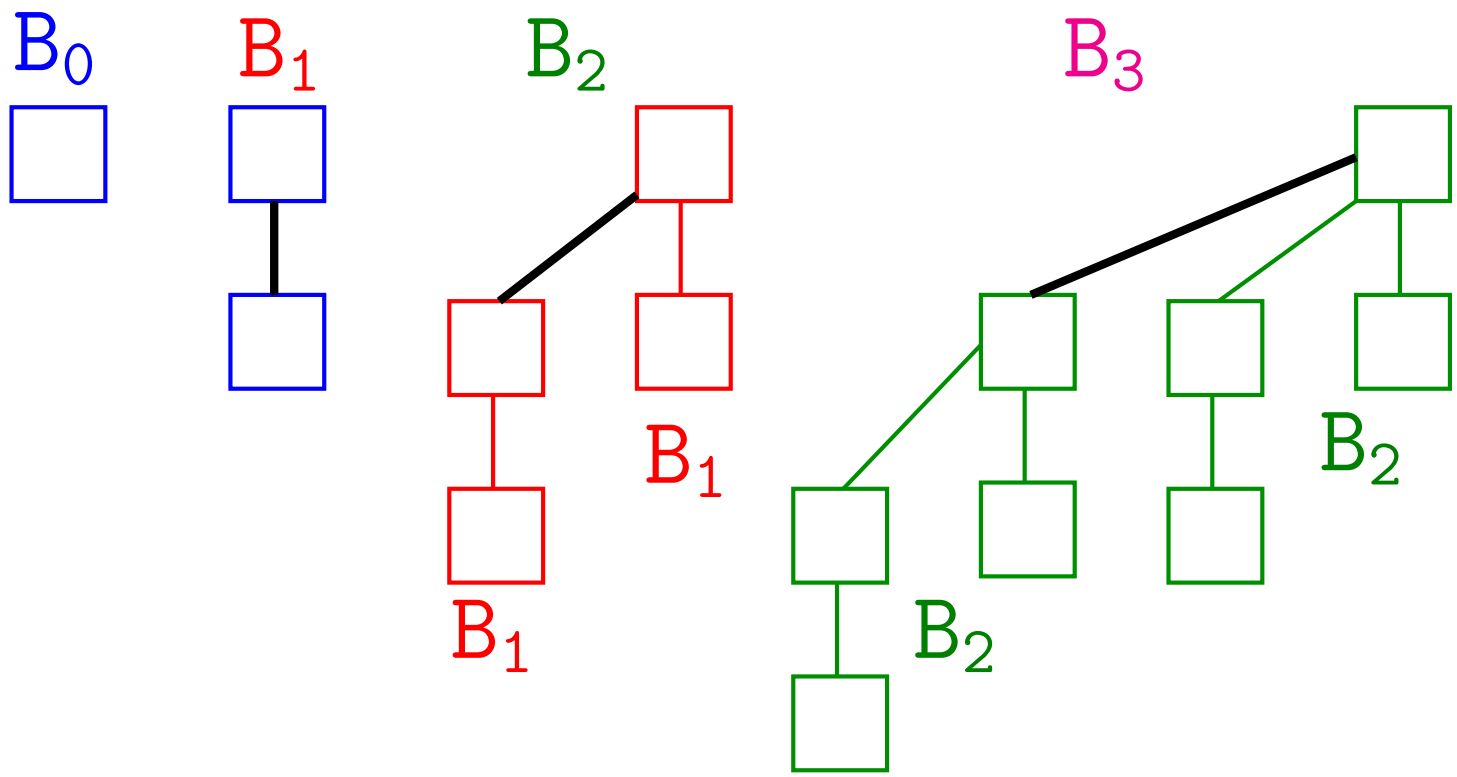
Binomial trees

Os nós de uma **árvore ordenada** têm
seus filhos ordenados: primeiro, segundo, ...

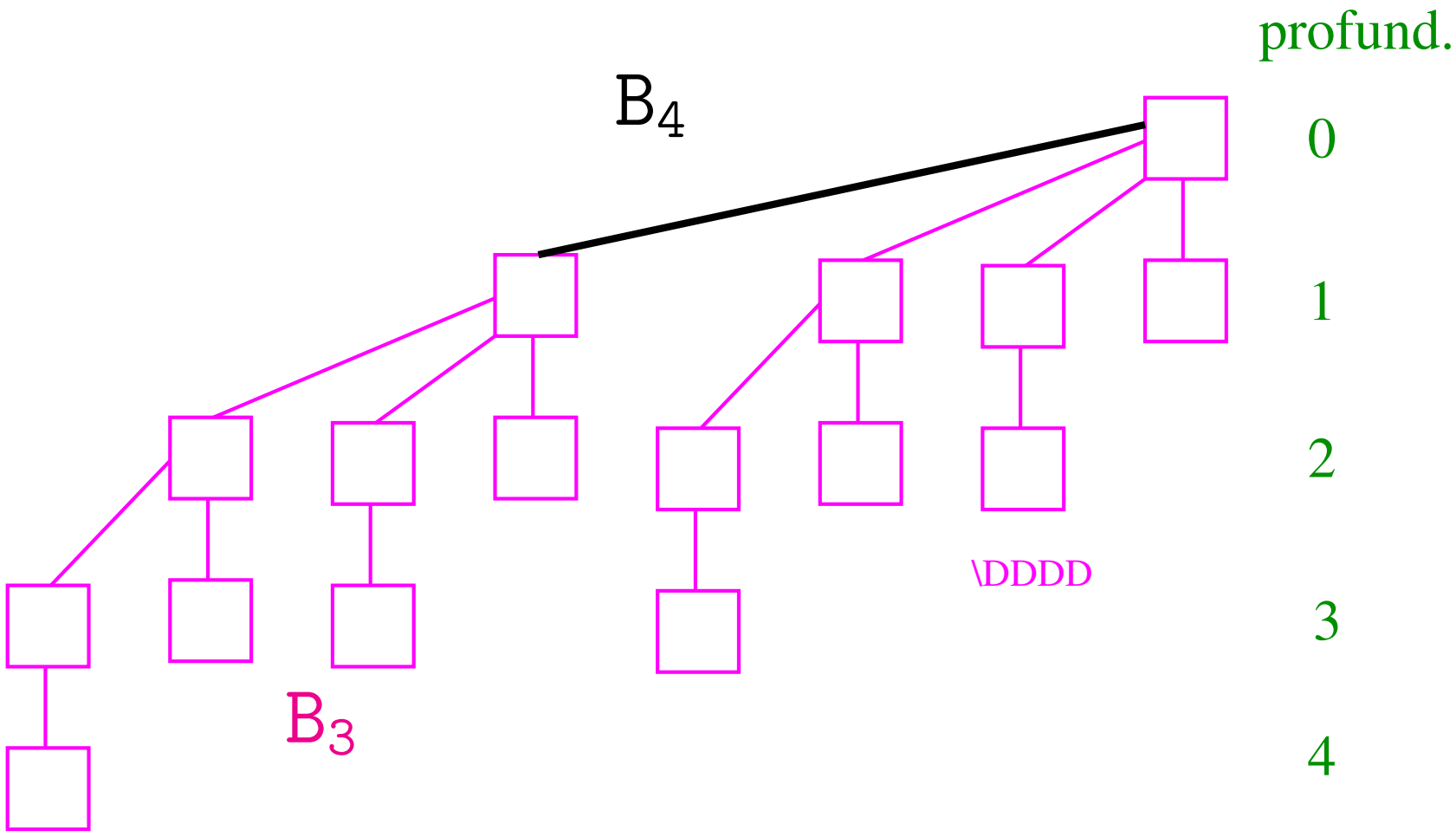
Árvores binomiais são definadas recursivamente:

- um nó é a árvore binomial B_0 de ordem 0.
- para $k = 1, 2, \ldots$
  a árvore binomial B_k de ordem k consiste de
  duas árvores B_{k-1} ligadas: a raiz de uma é
  o filho mais à esquerda da raiz da outra.

Binomial trees



Binomial trees



Binomial trees: estrutura

Em uma árvore binomial B_k de **ordem k**:

- a raiz tem k filhos;
- a árvore tem 2^k nós;
- a árvore tem altura k;
- há exatamente $\binom{k}{i}$ com profundidade i;
- os filhos da raiz são
  as árvores binomiais B_{k-1}, B_{k-2}, B_{k-3},...

Demonstração: por indução em k.

Binomial trees: mais estrutura

O seguinte fato será fundamental para o consumo de tempo das operações de um binomial heap.

> O grau máximo de qualquer nó em uma árvore binomial com $\texttt{n}$ nós é $\lg \texttt{n}$.

Os filhos de nó serão representados através de uma lista ligada ordenada pelos graus (`order`) dos filhos.

Binomial heap

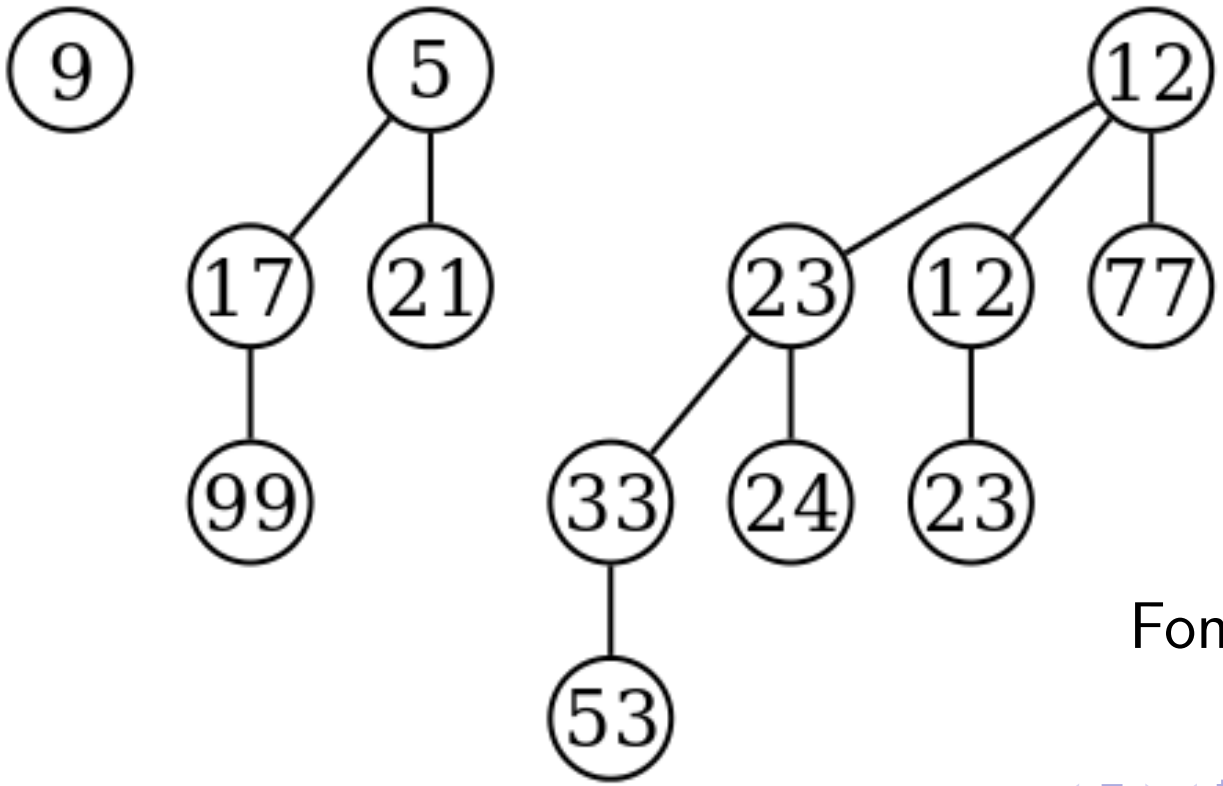
Uma **binomial heap** `H` é uma coleção de binomial trees que satisfaz as seguintes propriedades:

- cada binomial tree em `H` é uma `MinPQ`/`MaxPQ`:
  o valor associado ao `item` de cada nó
  é menor ou igual (maior ou igual)
  ao valor associado aos seus filhos;
- `H` possui no máximo
  uma binomial tree de cada ordem.

Binomial heap

Em uma binomial heap `H` com `n itens`,
os dígitos na representação binária de `n` indicam
a ordem das binomial trees que formam `H`:

$$\texttt{n} = 13 = \texttt{1} \times 2^0 + \texttt{0} \times 2^1 + \texttt{1} \times 2^2 + \texttt{1} \times 2^3.$$

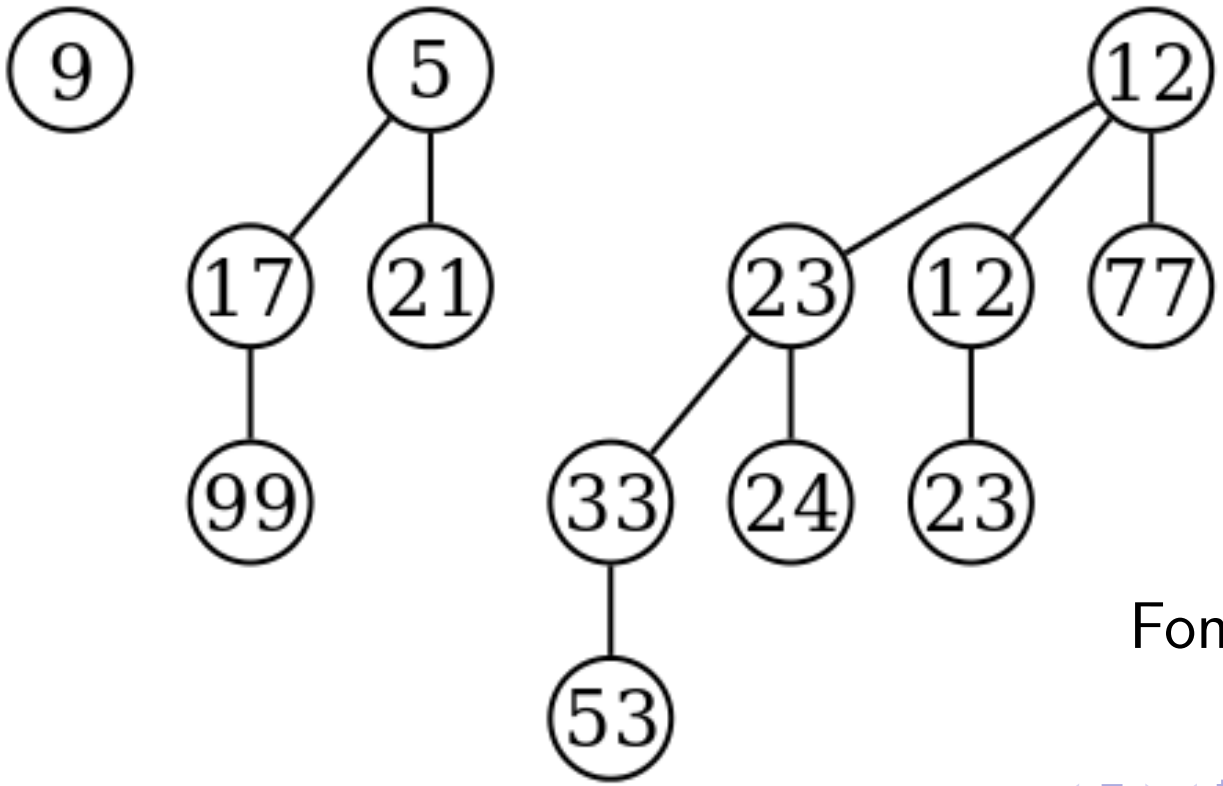


Fonte: Wikipedia

Binomial heap

A maior binomial tree em uma binomial heap com `n itens` tem ordem $\lfloor \lg \mathtt{n} \rfloor$ e no máximo $1+\lfloor \lg \mathtt{n} \rfloor$ árvores.

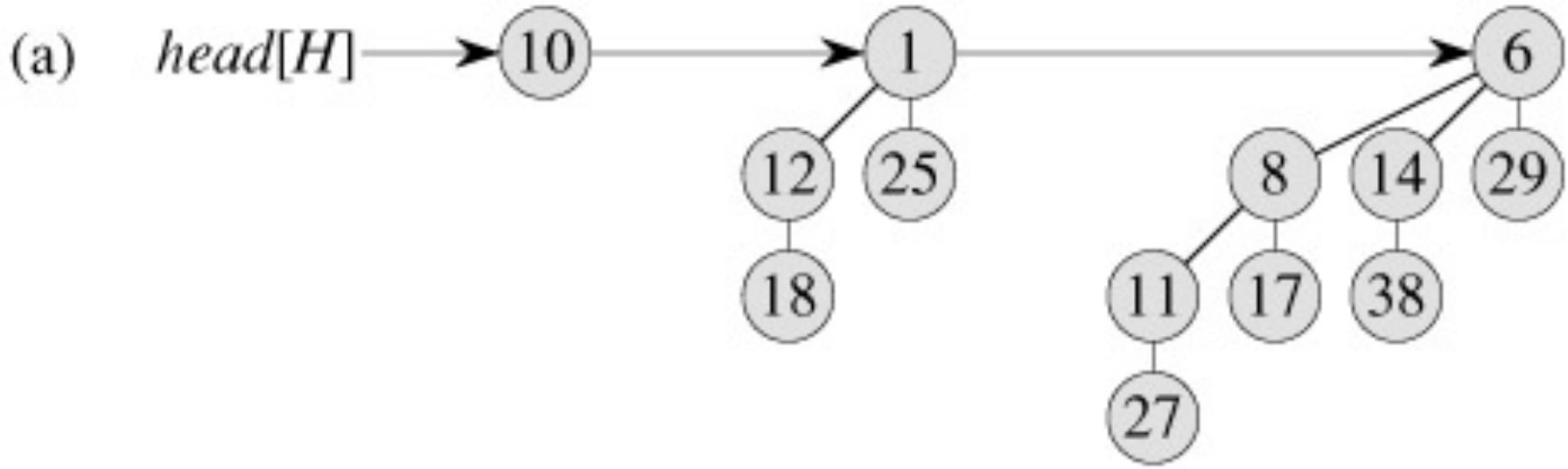
$$\mathtt{n} = 13 = \mathtt{1} \times 2^0 + \mathtt{0} \times 2^1 + \mathtt{1} \times 2^2 + \mathtt{1} \times 2^3.$$



Fonte: Wikipedia

Binomial heap: estrutura de dados

Fonte: CLRS

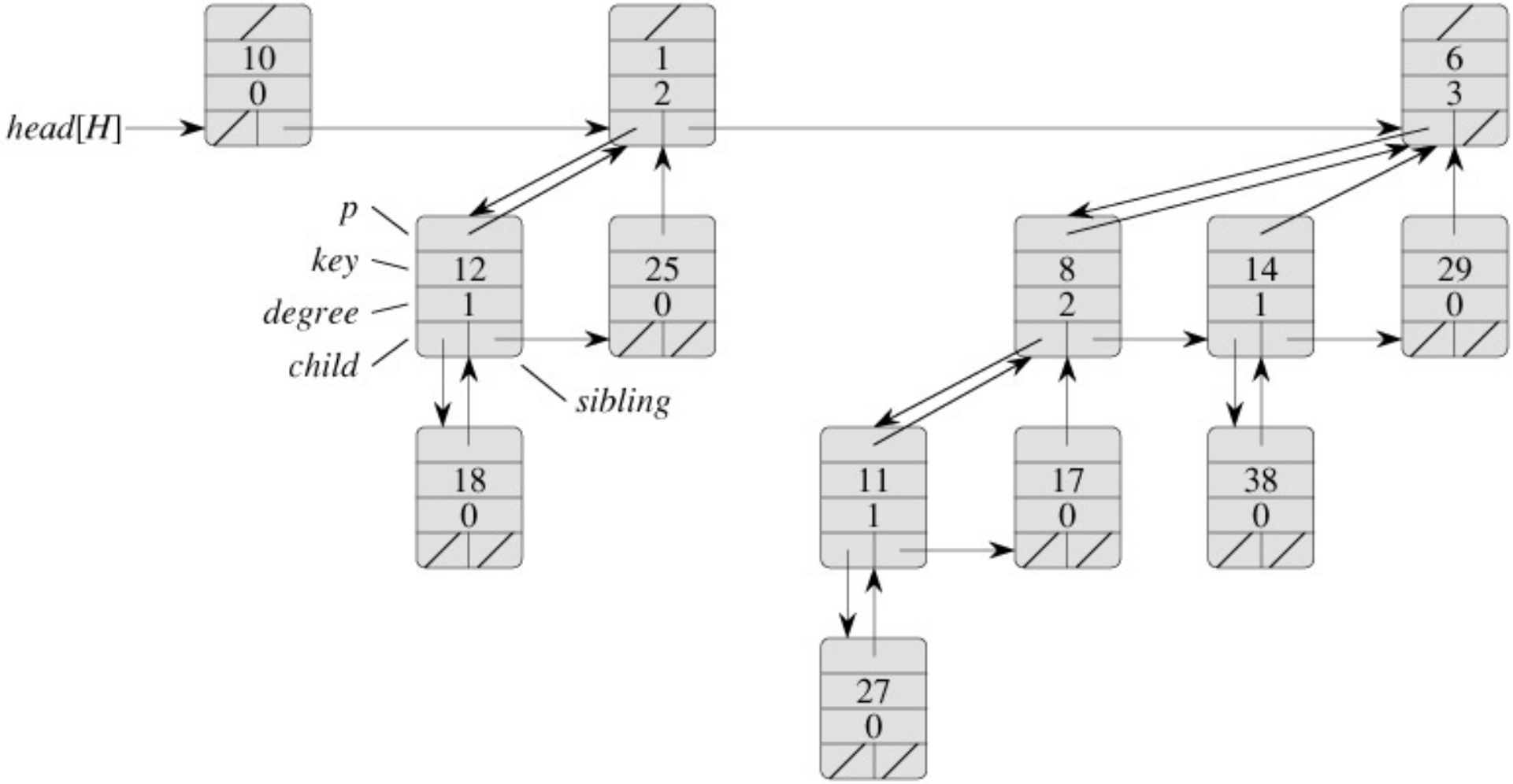


BinomialMinPQ: struct node e Link

Representação de um nó de uma binomial tree.

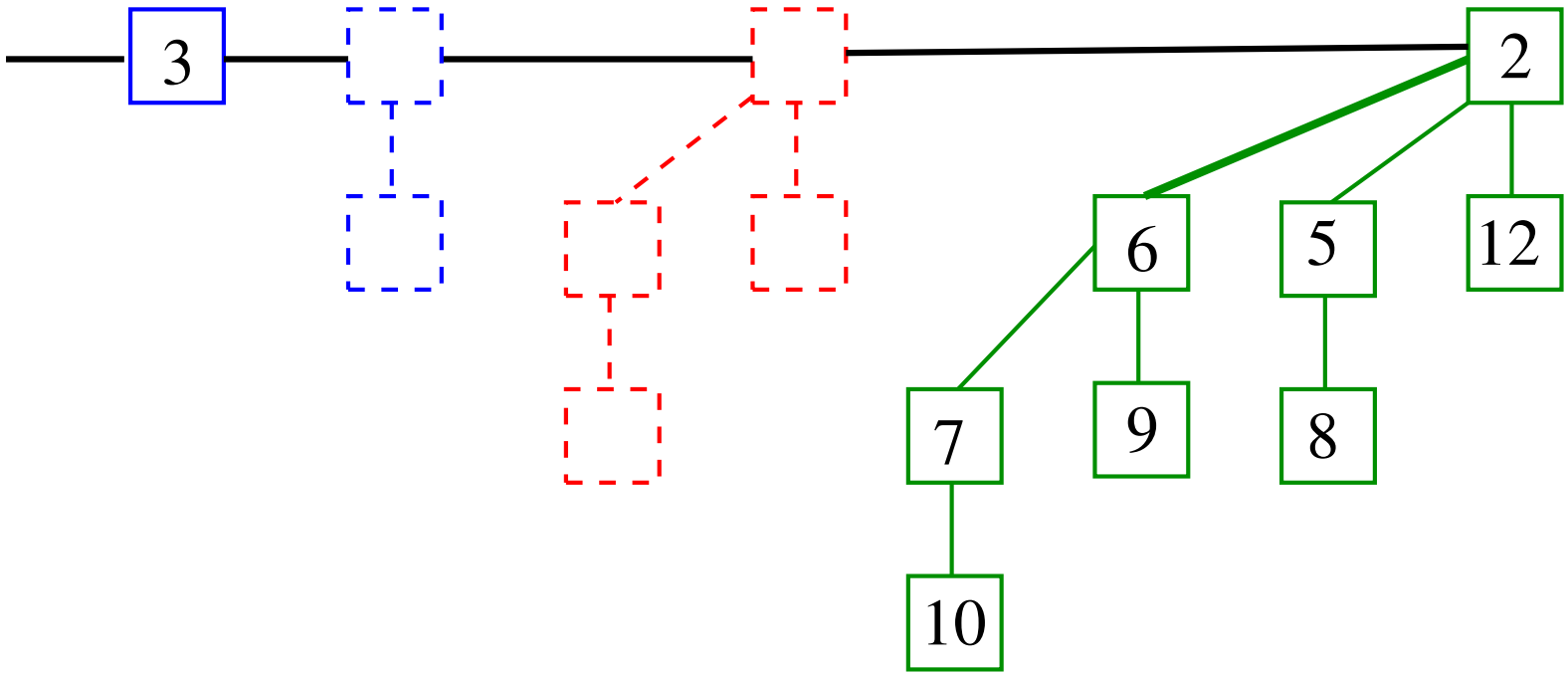
```
typedef struct node *Link;
struct node {
   Item item;
   Link child,    /* filho mais a esquerda */
   Link sibling, /* lista de irmãos */
   Link parent;  /* pai */
   int  order;   /* ou grau, degree */
};
```

A lista de irmão está em ordenada de acordo com o grau dos nós (ordem das árvores).

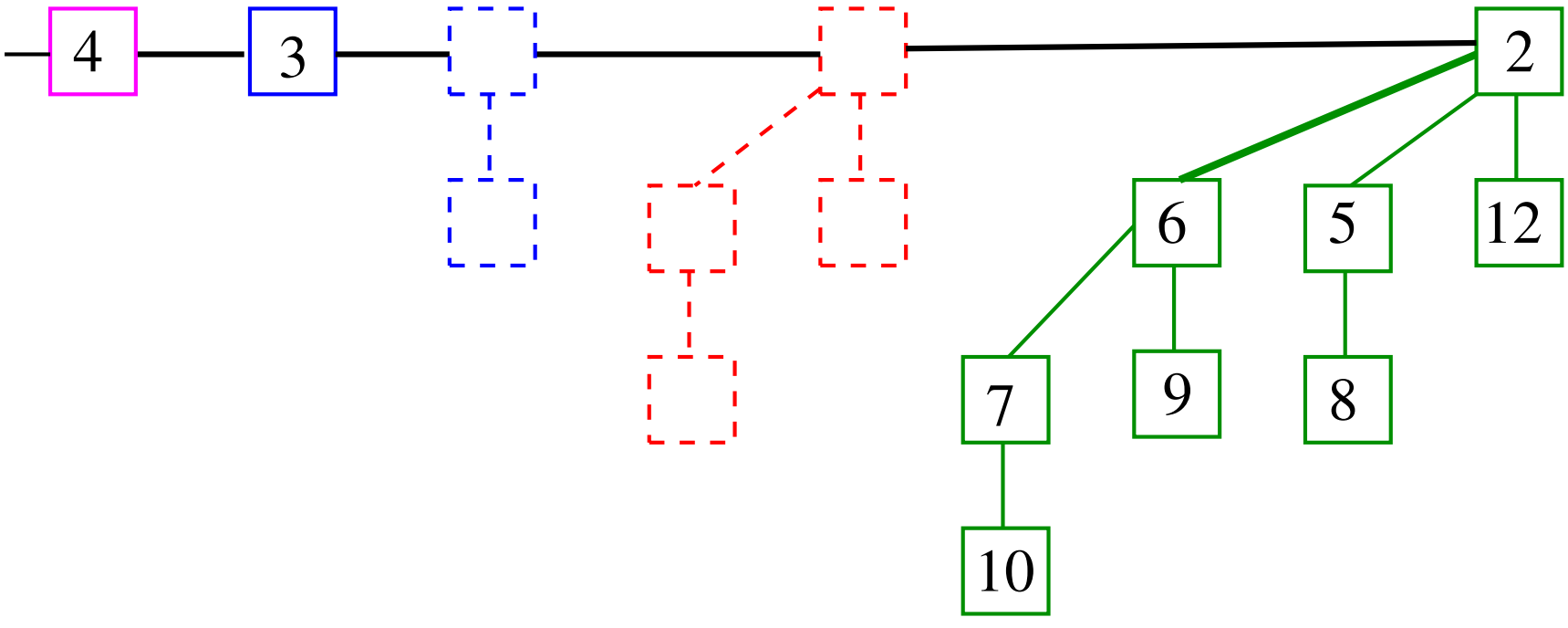

head[H]
10
0
1
2
6
3
p
key
degree
child
sibling
12
1
25
0
8
2
14
1
29
0
18
0
11
1
17
0
38
0
27
0

Fonte: CLRS

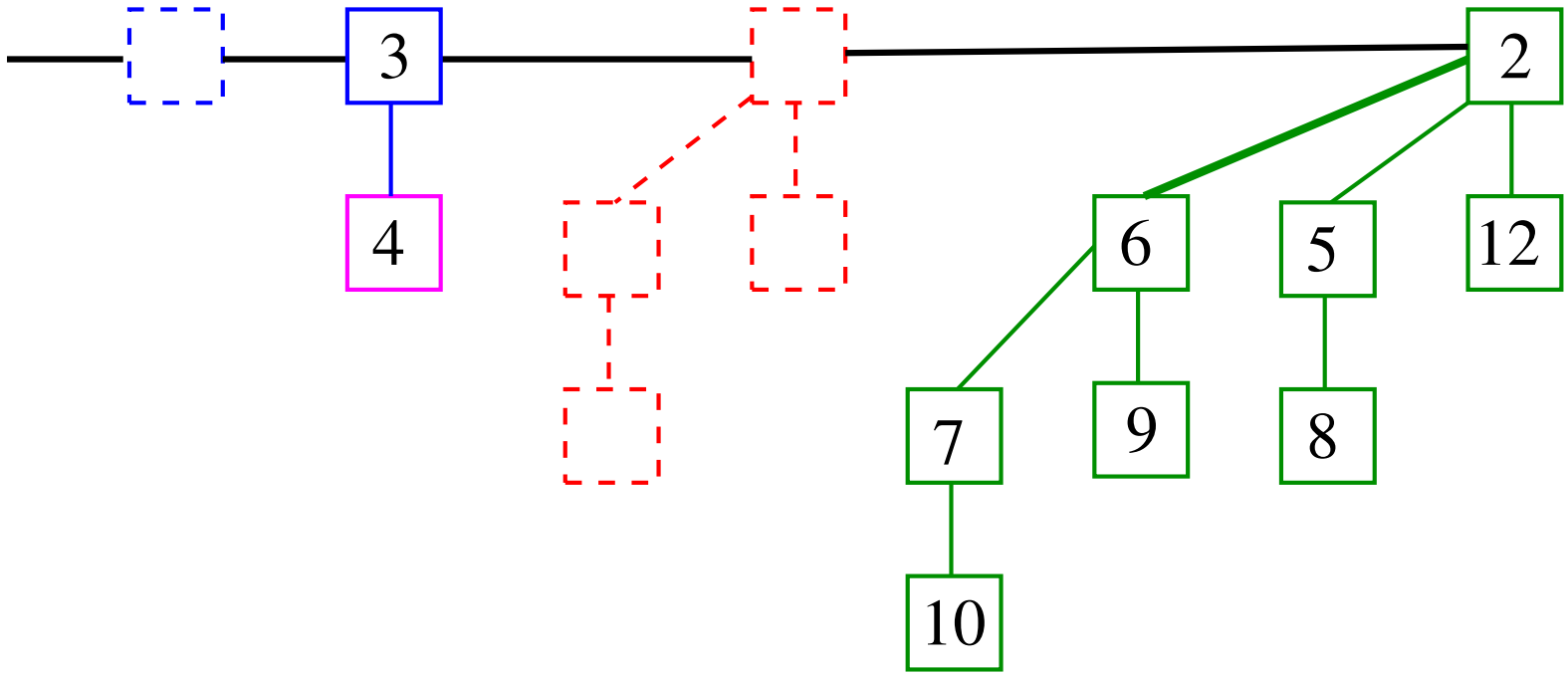
BinomialMinPQ: insert()



BinomialMinPQ: insert()



BinomialMinPQ: insert()



BinomialMinPQ: insert()

Como existem $1 + \lfloor \lg n \rfloor$ binomial trees
que podem ser unidas concluímos o seguinte.

No pior caso, o consumo de tempo
da operação insert() é $O(\lg n)$,
onde n é o número de itens na binomial heap.