MAC0323 Algoritmos e Estruturas de Dados II

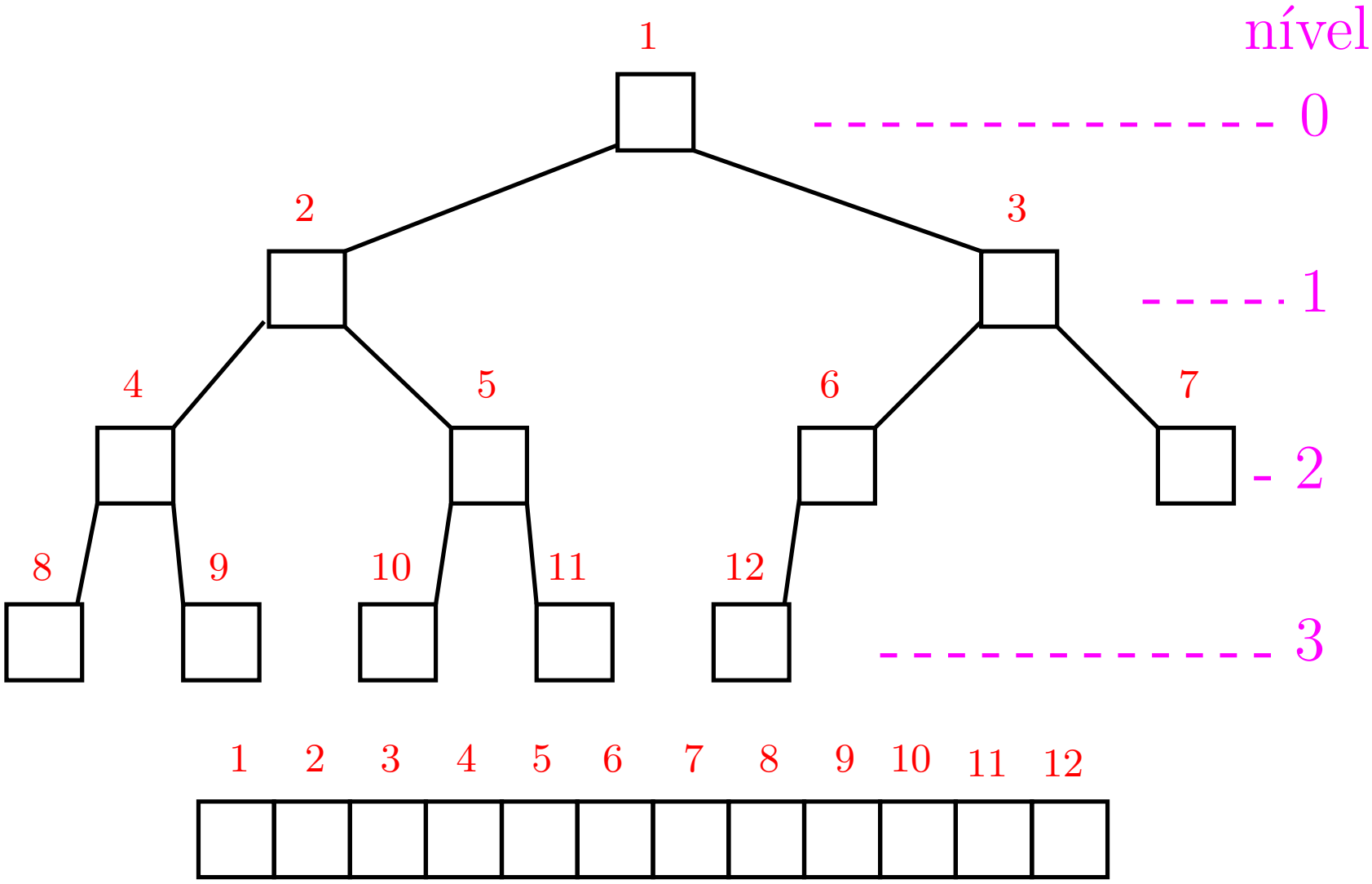
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 5

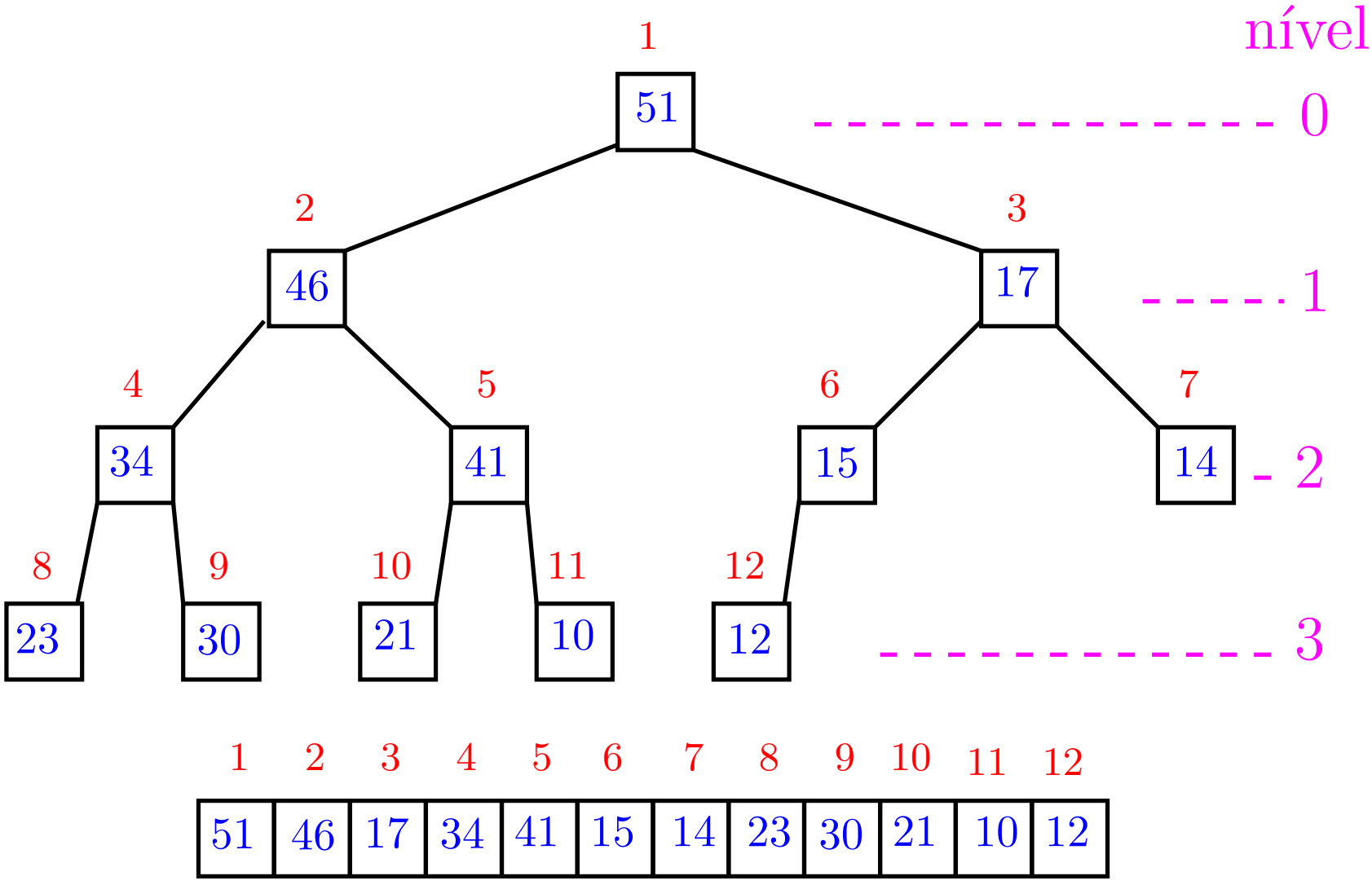
Representação de árvores em vetores



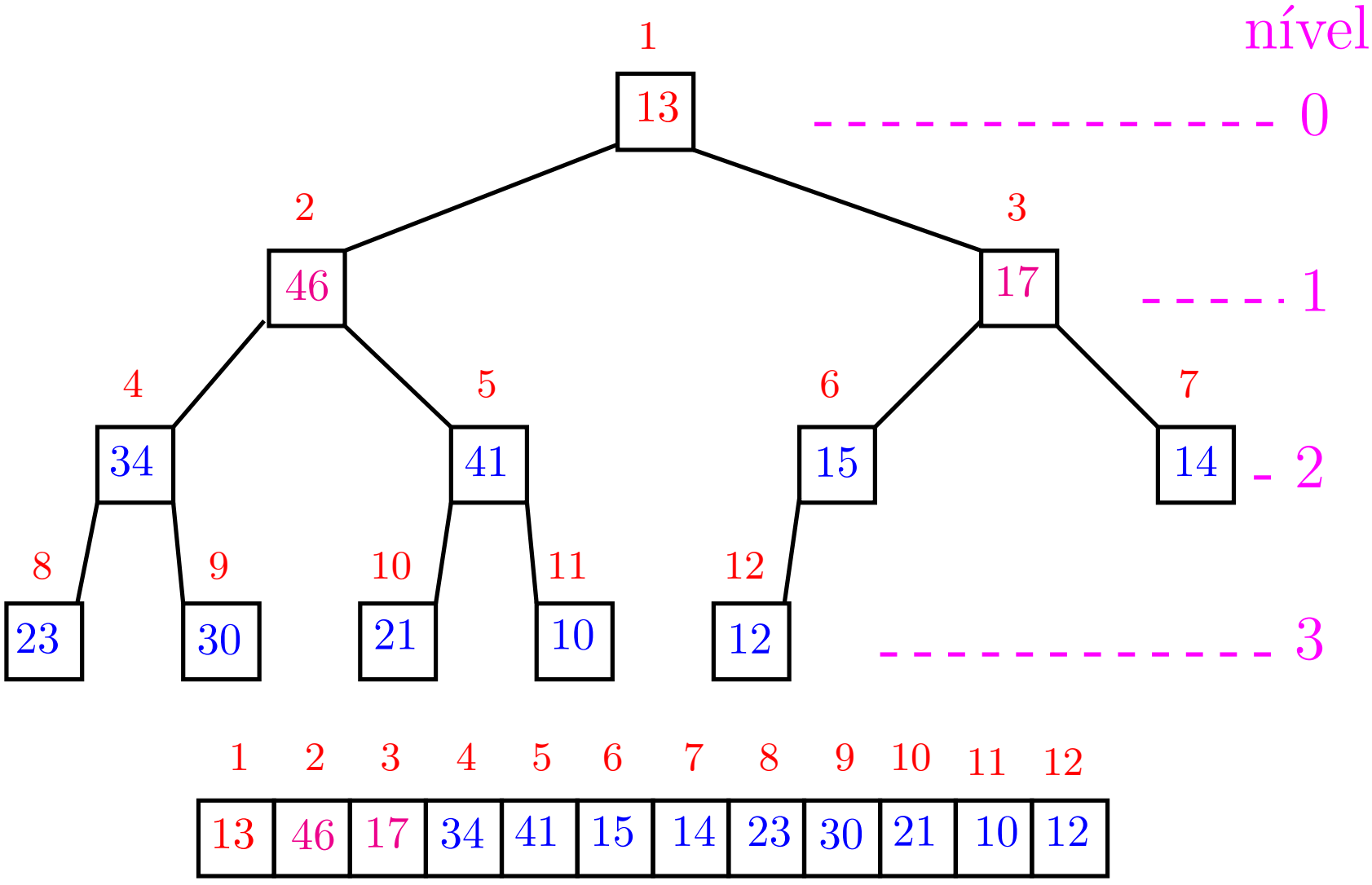
Resumão da estrutura

filho esquerdo de i:	$2\,i$
filho direito de i:	$2\,i + 1$
pai de i:	$\lfloor i/2 \rfloor$
nível da raiz:	0
nível de i:	$\approx \lg i$
altura da raiz:	$\approx \lg m$
altura da árvore:	$\approx \lg m$
altura de i:	$\approx \lg(m/i)$
altura de uma folha:	0

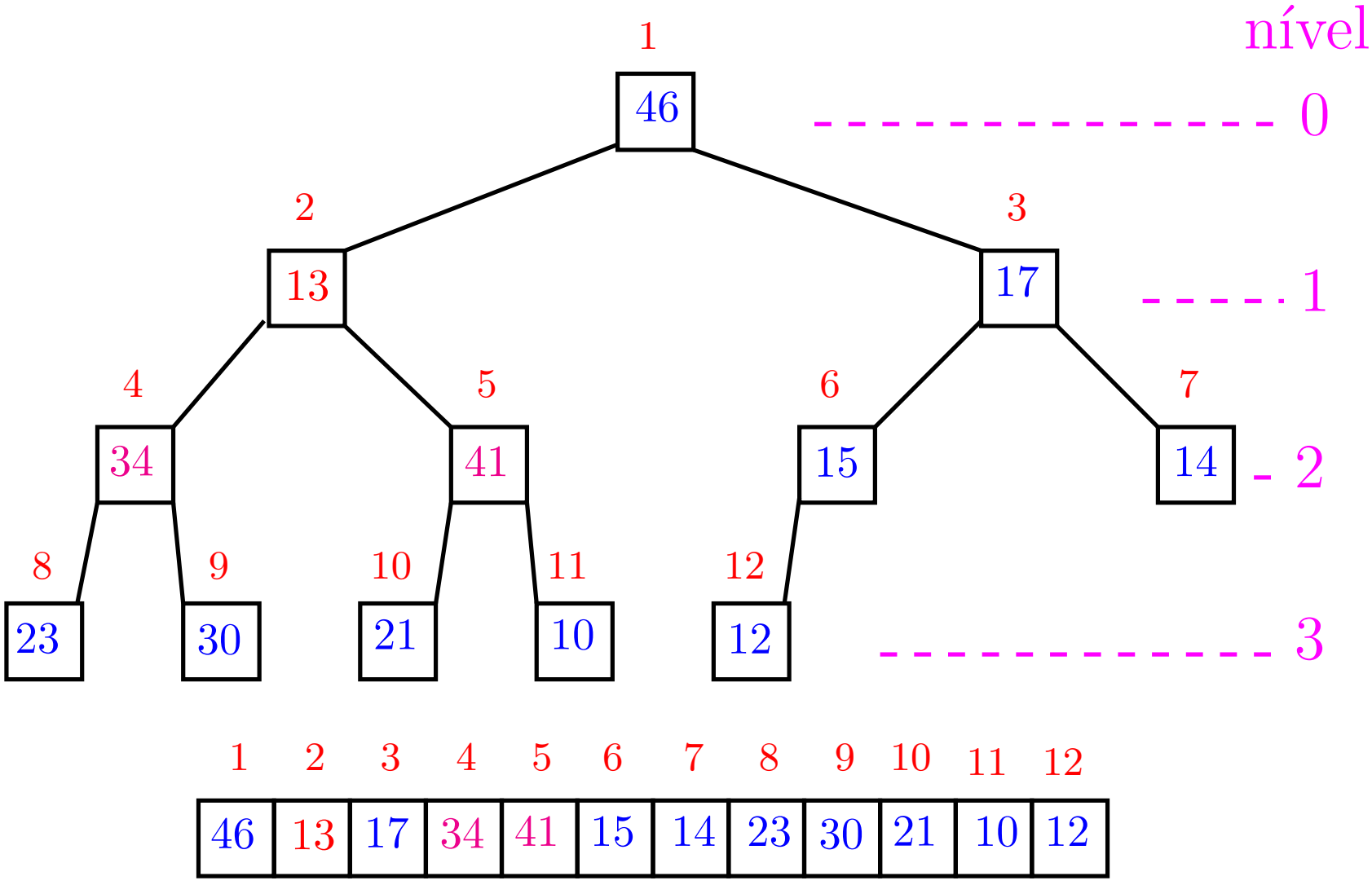
max-heap


nível
1
51
0
2
46
3
17
1
4
34
5
41
6
15
7
14
2
8
23
9
30
10
21
11
10
12
12
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 12

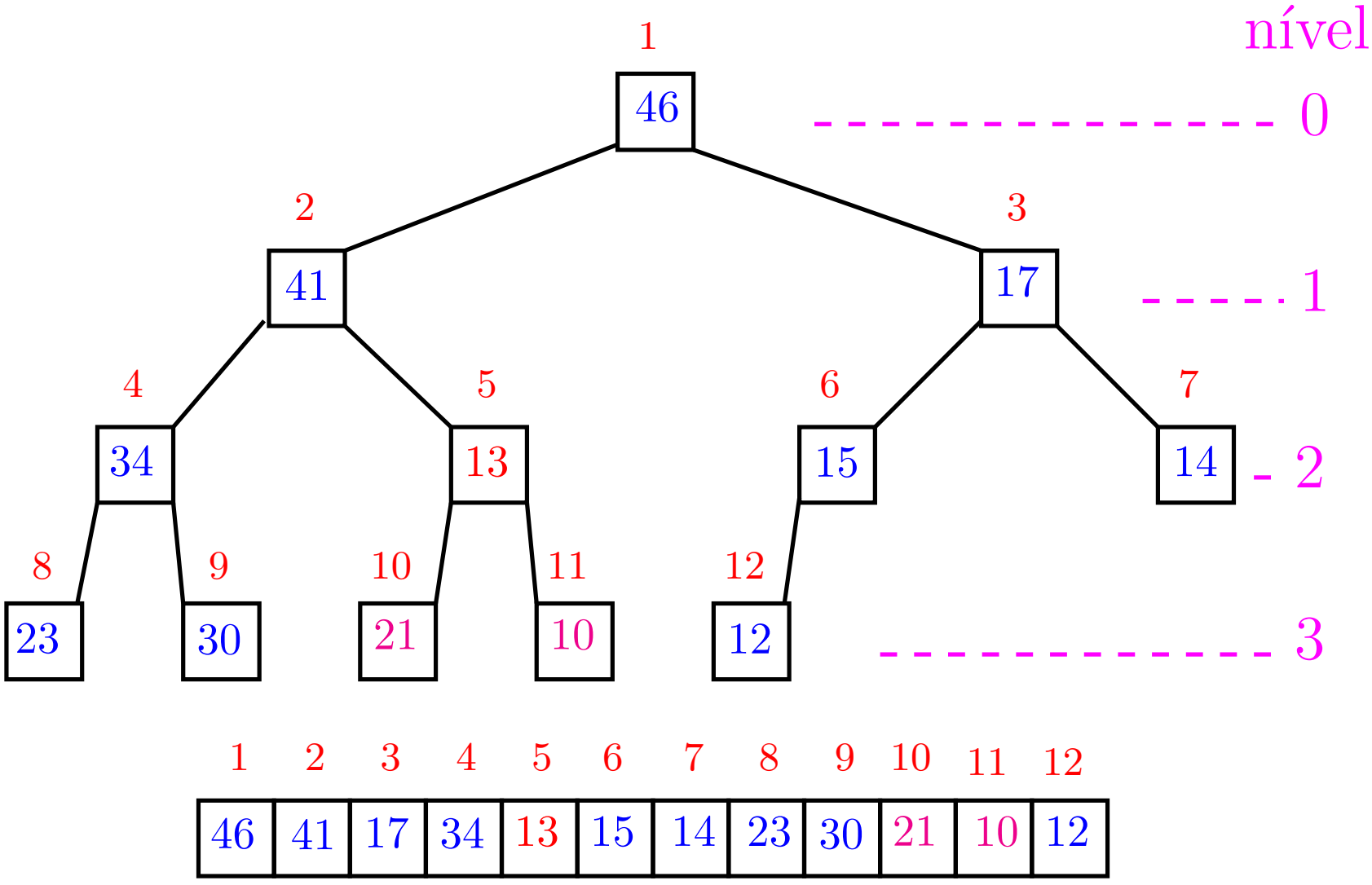
Função básica: `sink()`



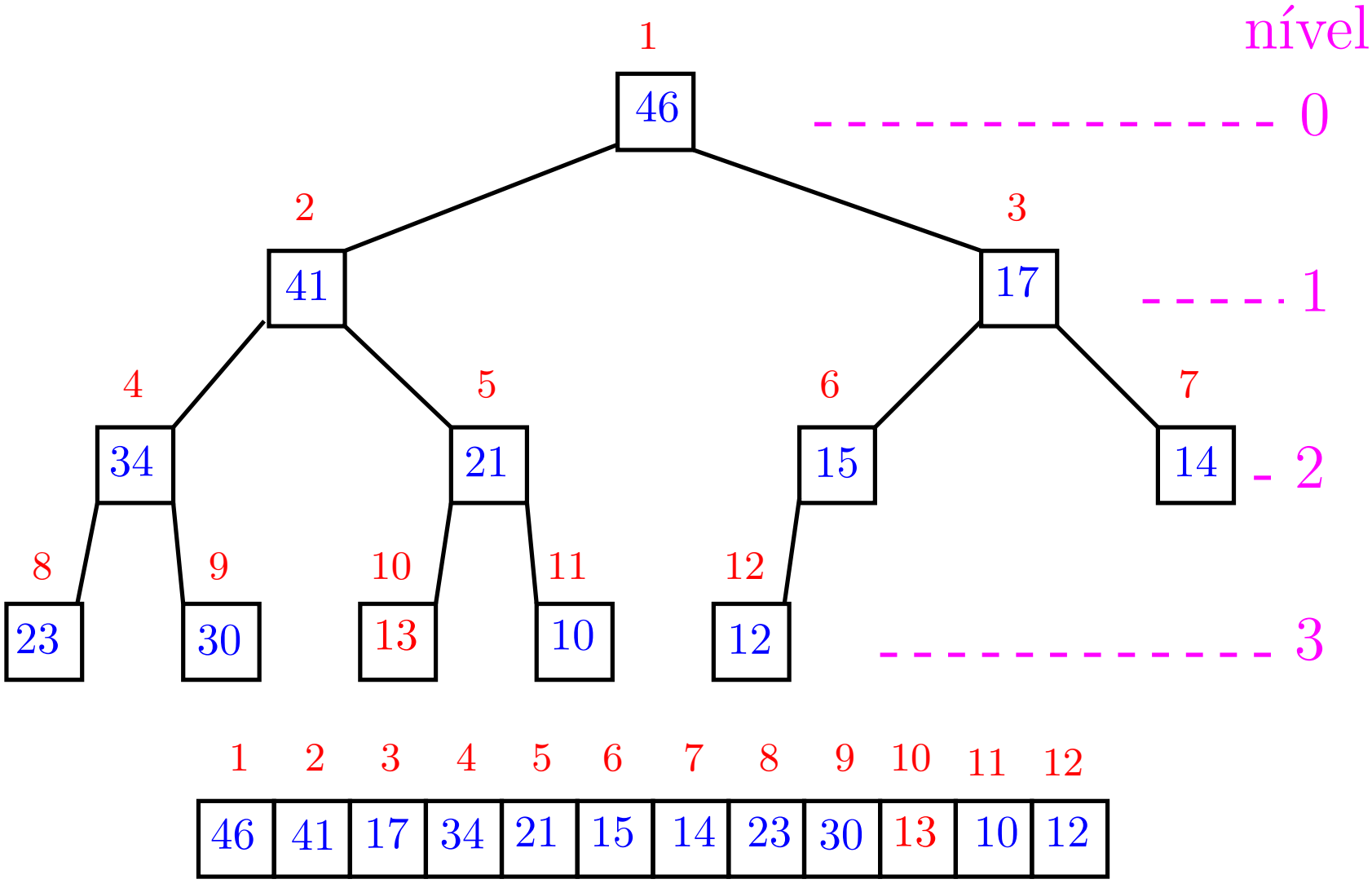
Função básica: `sink()`



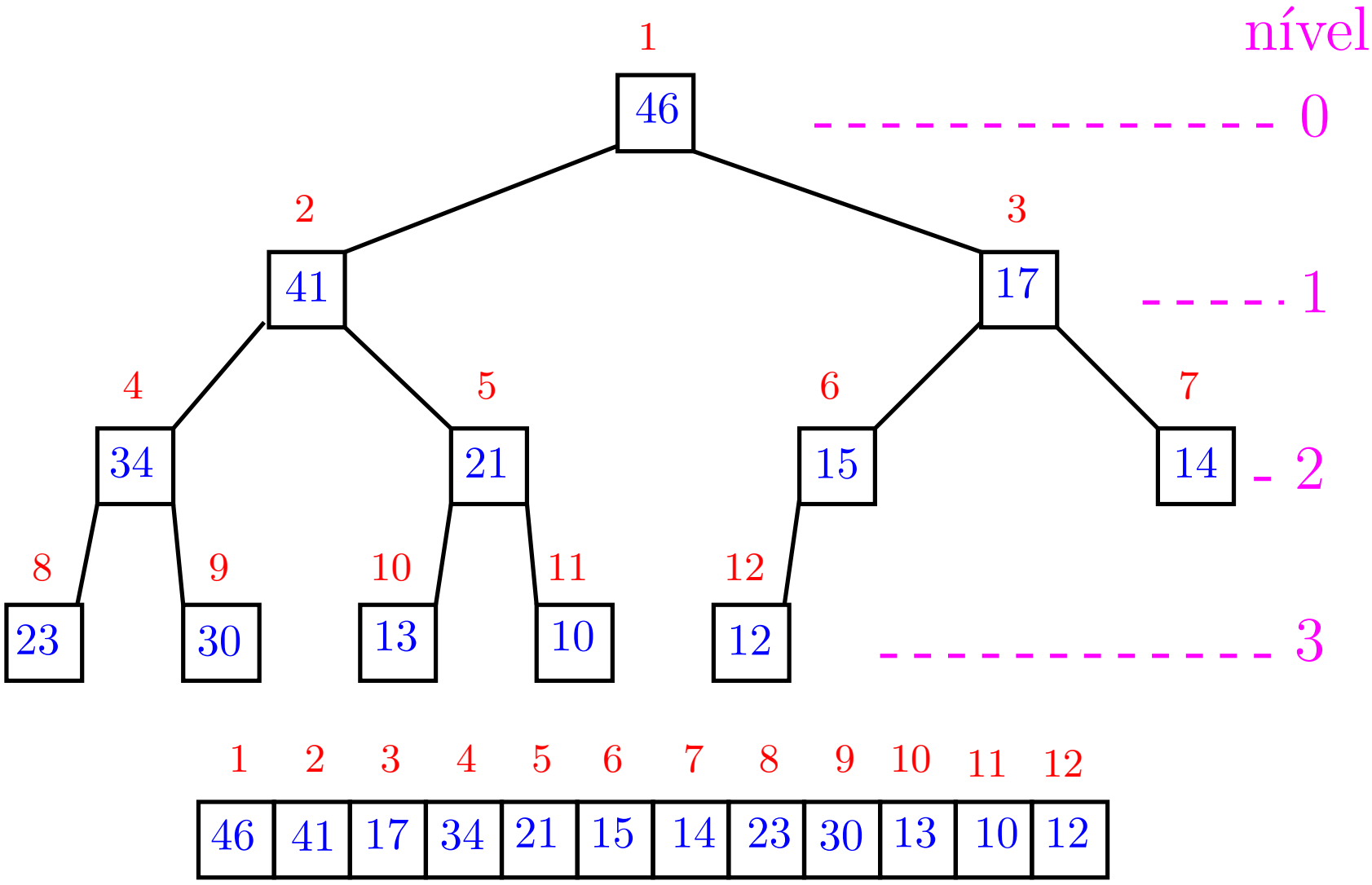
Função básica: `sink()`



Função básica: `sink()`



Função básica: `sink()`



Função sink

Implementação faz apenas deslocamentos (linha 5).

```
static void sink (int p, int m, int *h) {
1    int f = 2*p; int x = h[p];
2    while (f <= m) {
3       if (f < m && h[f] < h[f+1]) f++;
4       if (x >= h[f]) break;
5       h[p] = h[f];
6       p = f; f = 2*p; /* sink */
     }
7    h[p] = x;
 }
```

Consumo de tempo

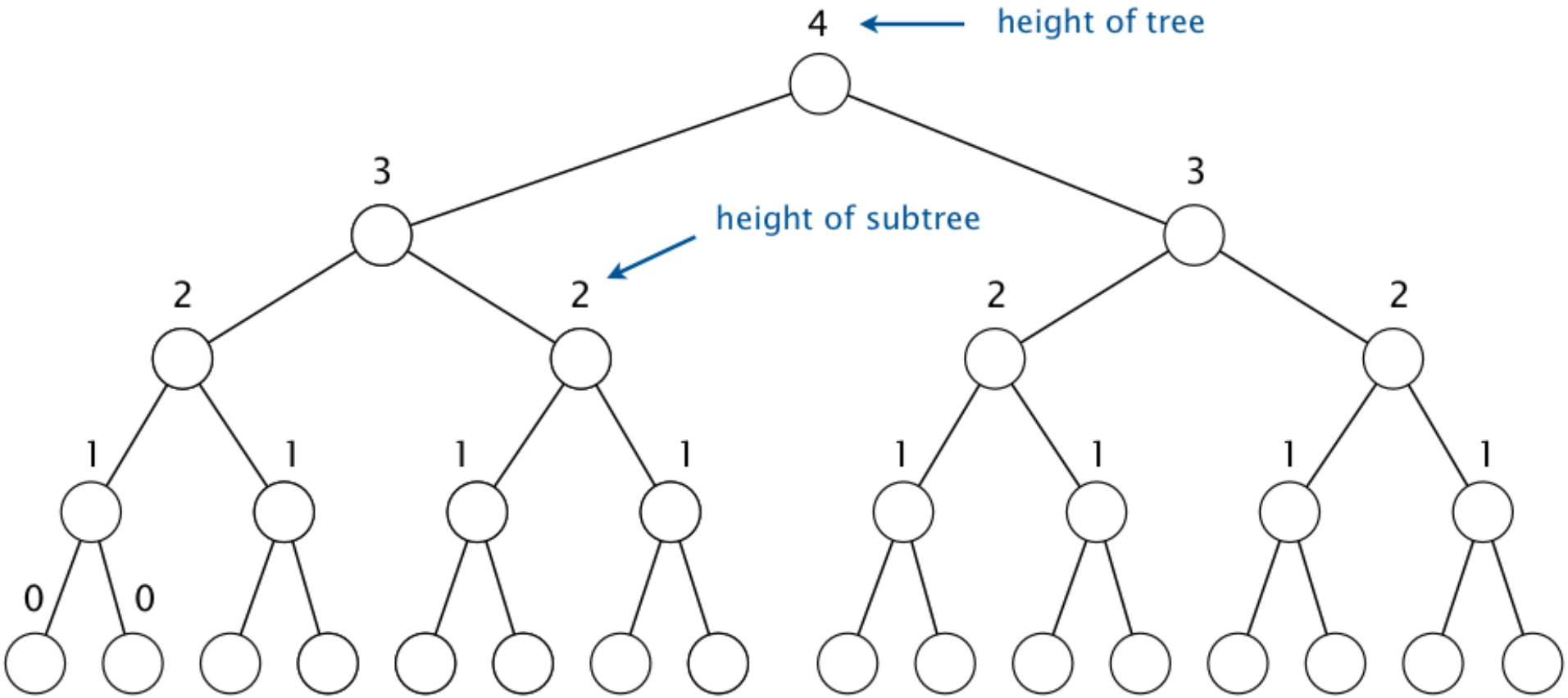
O consumo de tempo da função `sink()` é proporcional a $\lg m$.

O consumo de tempo da função `sink()` é $O(\lg m)$.

Verdade seja dita . . .

O consumo de tempo da função `sink()` é proporcional a $O(\lg(m/p))$.

Altura das subárvores



Fonte: algs4

Consumo de tempo

Suponha que a altura do `max-heap` seja `h`.

Existem no máximo 2^{h-k} nós de altura `k`.

Número de deslocamentos feitos durante
a construção do heap é não superior a

$$h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(1)$$

Consumo de tempo

Suponha que a altura do max-heap seja h.

Existem no máximo 2^{h-k} nós de altura k.

Número de deslocamentos feitos durante a construção do heap é não superior a

$$\begin{aligned} & h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(1) \\ = \; & 2^h \left(\frac{h}{2^h} + \frac{h-1}{2^{h-1}} + \frac{h-2}{2^{h-2}} + \cdots + \frac{1}{2} \right) \\ = \; & 2^h \sum_{i=1}^{h} \frac{i}{2^i} \;=\; 2^{h+1} - 2 - h. \end{aligned}$$

A última igualdade vem de uma conta semelhante às da aula passada, mas com séries finitas.

Consumo de tempo

Suponha que a altura do max-heap seja h.

Existem no máximo 2^{h-k} nós de altura k.

Número de deslocamentos feitos durante
a construção do heap é não superior a

$$\begin{aligned} & h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}(1) \\ = \; & 2^h \sum_{i=1}^{h} \frac{i}{2^i} \;=\; 2^{h+1} - 2 - h \\ \leq \; & 2(2^h - 1) \;<\; 2m. \end{aligned}$$

A última desigualdade vale pois uma árvore binária completa de altura $h-1$ tem 2^h-1 nós.

Conclusão

O consumo de tempo para
construir um `max-heap` é $O(n)$.

AULA 6

Filas priorizadas

Fonte: `We love it`

Filas priorizadas, PF, Priority queues, S&W

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

Item deve ser um tipo com uma ordem total.

Cliente MinPQ: TopM

Transaction é uma das classes do algs4.

Em C, essa classe está definida
nos arquivos transaction.h e transaction.c.

Cliente MinPQ: TopM

Transaction é uma das classes do algs4.

Em C, essa classe está definida
nos arquivos transaction.h e transaction.c.

O programa retorna as M transações de maior valor.

As transações são lidas da entrada padrão
e estão no arquivo transactions.txt.

Cliente MinPQ: TopM

Arquivo transaction.h:

```
struct transaction {
   char nome[12];
   int dia, mes, ano;
   float valor;
};
typedef struct transaction Transaction;

Transaction *readT();
void printT(Transaction *);
void freeT(Transaction *);
bool lessT(Transaction *, Transaction *);
```

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);
        }
    }
```

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();
}
```

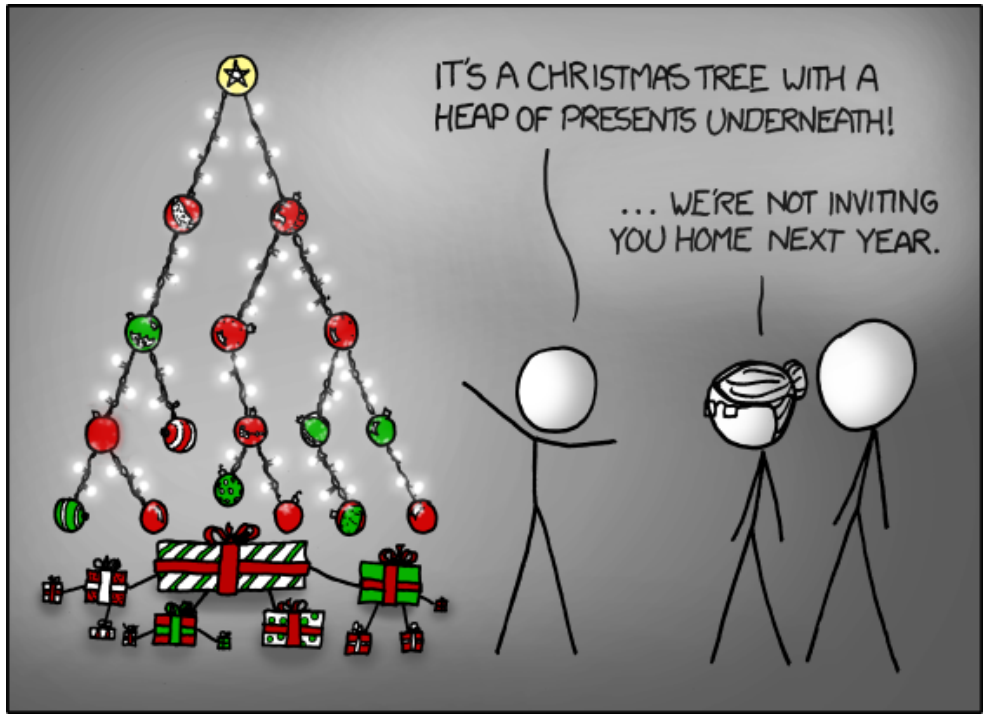
Implementações elementares

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.

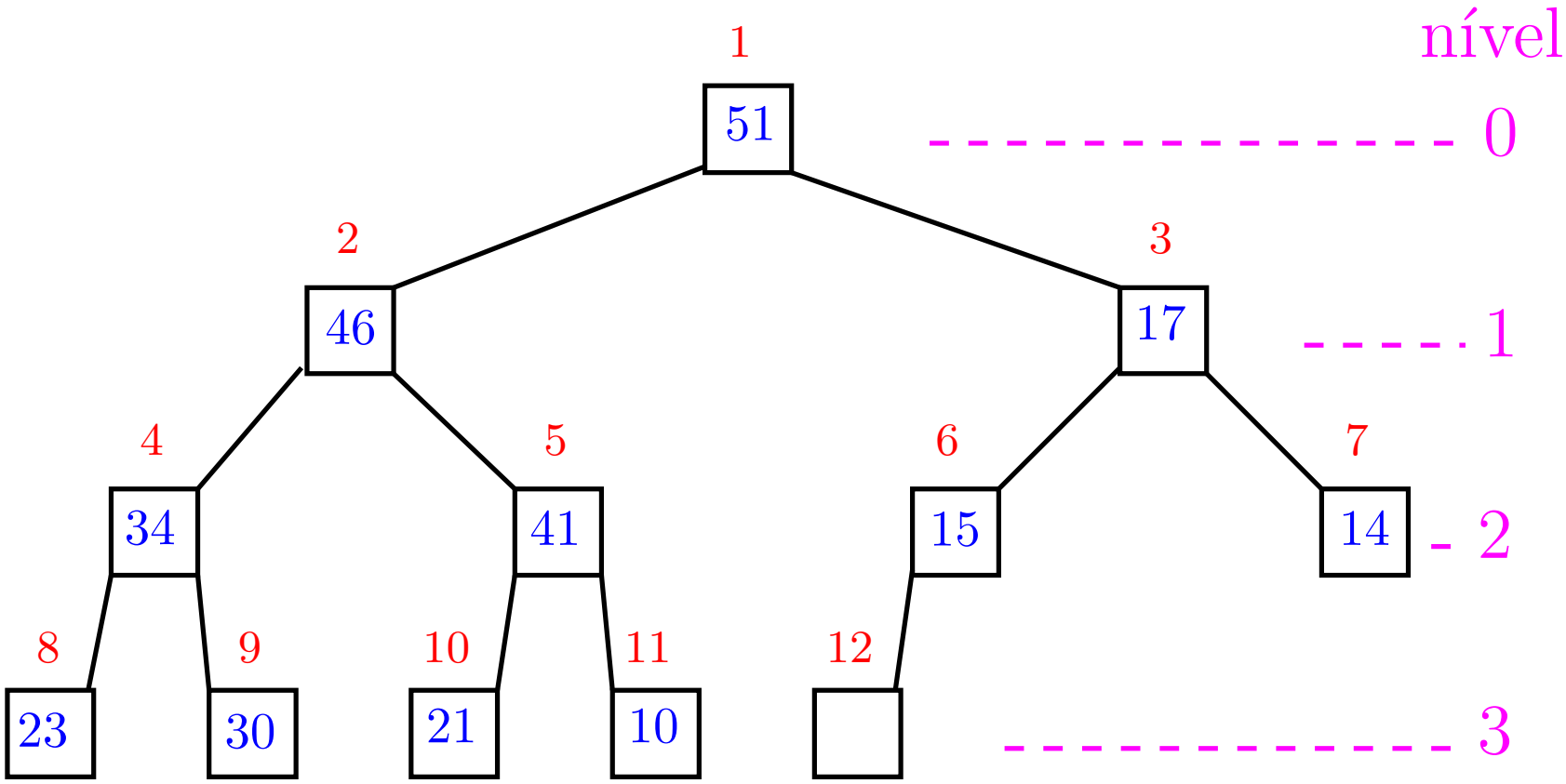
PQ de máximo



Fonte: http://xkcd.com/835/

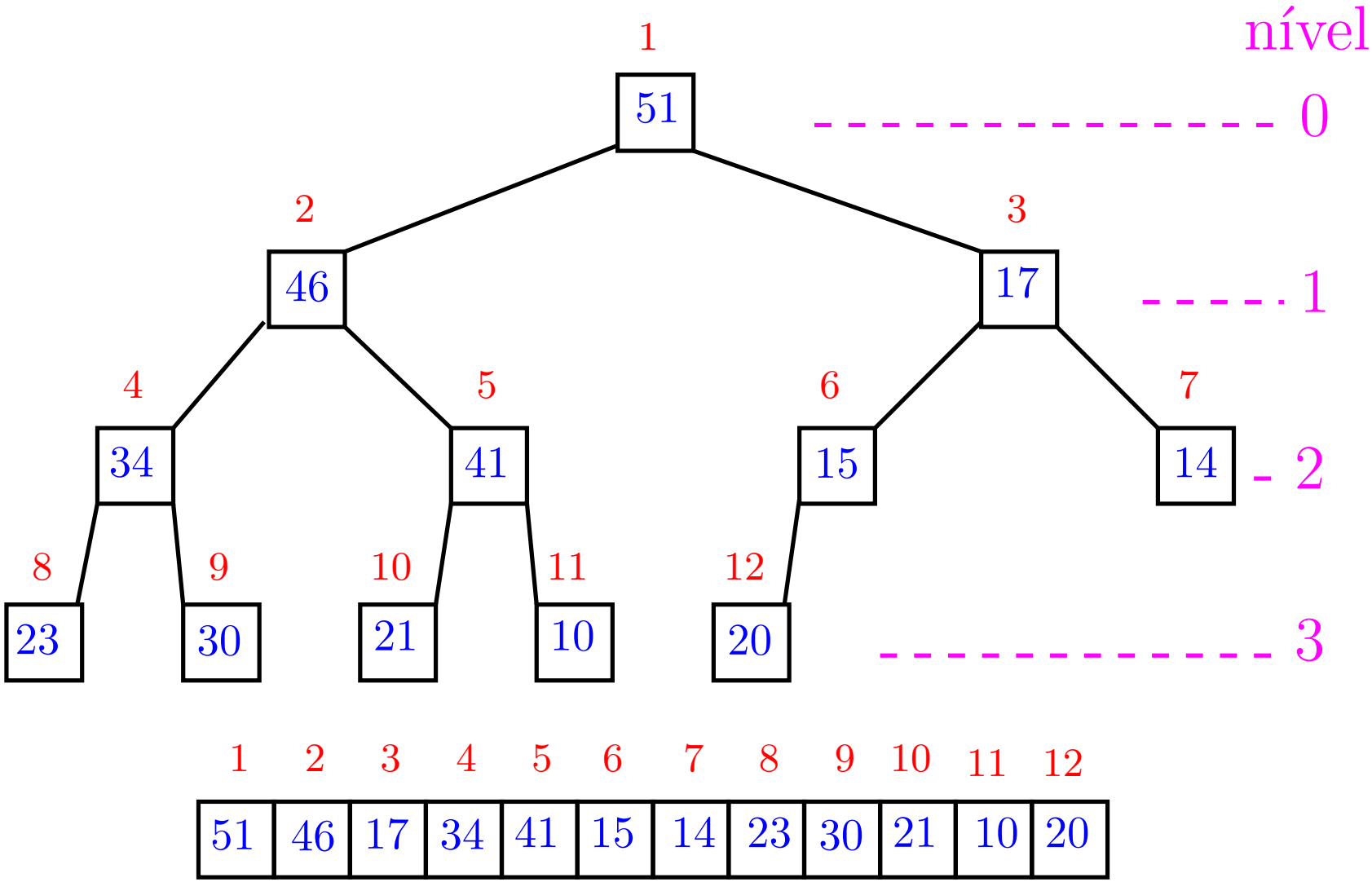
Filas priorizadas, PF, Priority queues, S&W

max-heap: insert()

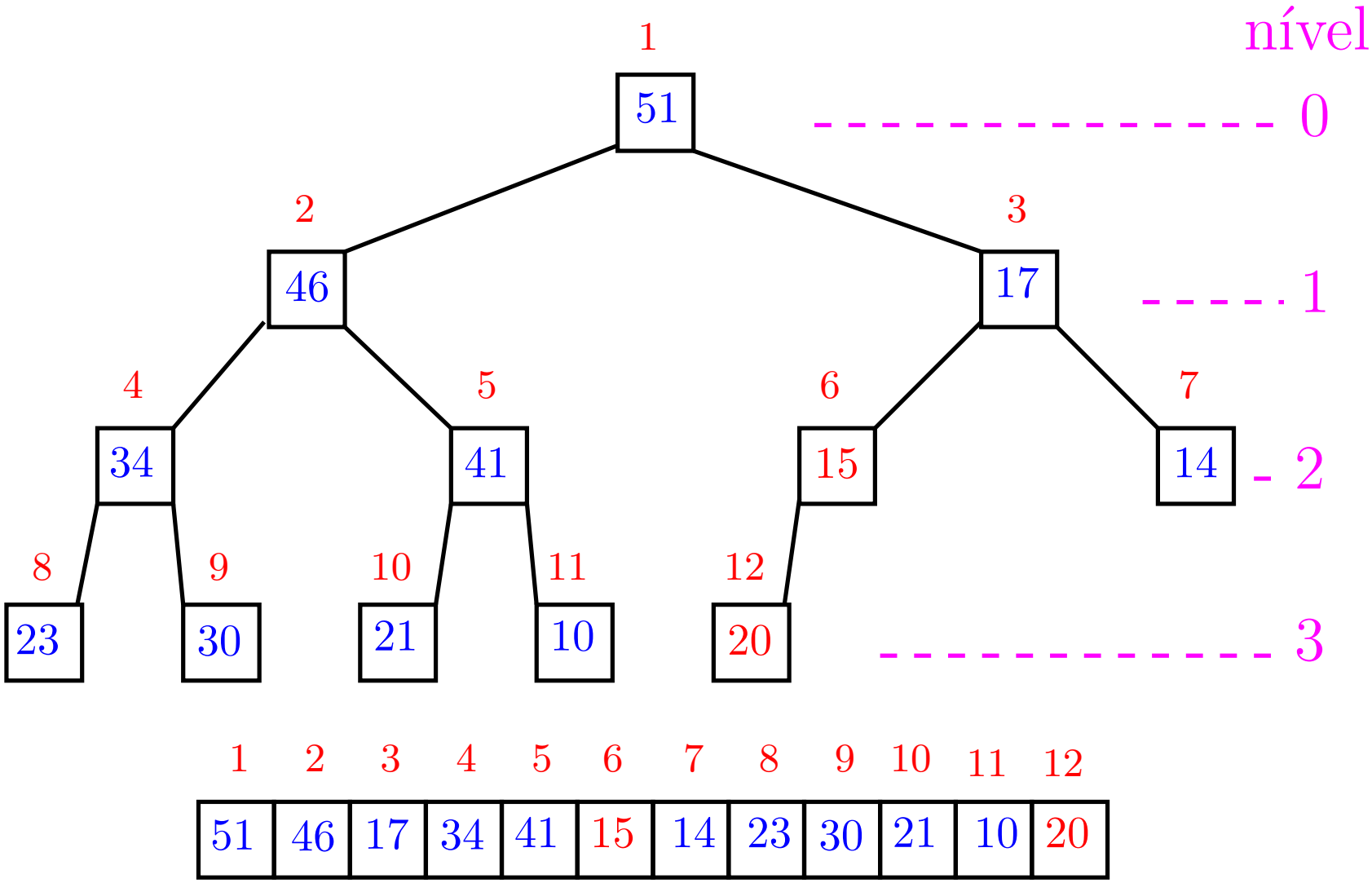


1	2	3	4	5	6	7	8	9	10	11	12
51	46	17	34	41	15	14	23	30	21	10	

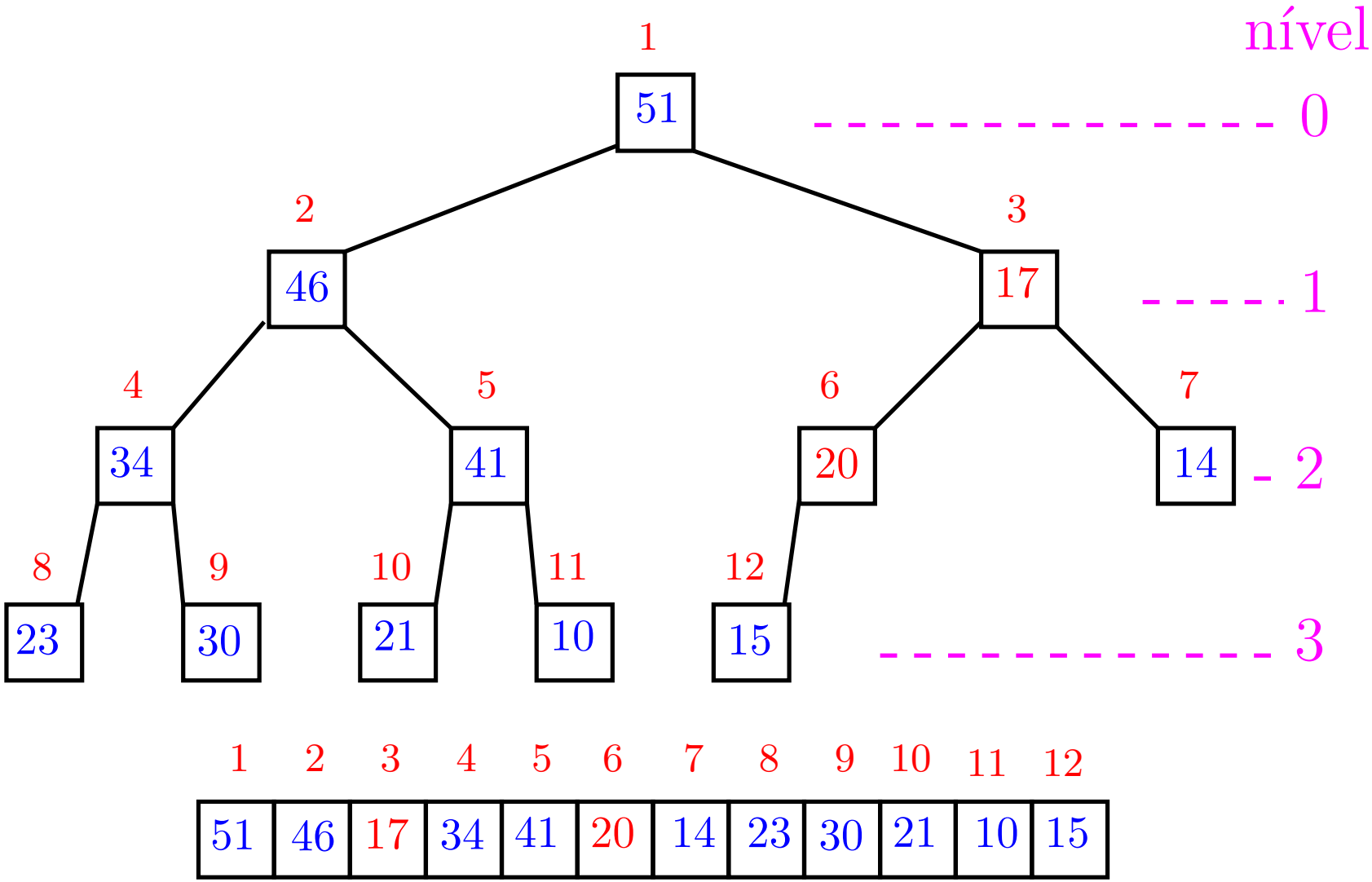
swim()


nível
1
51
0
2
46
3
17
1
4
34
5
41
6
15
7
14
2
8
23
9
30
10
21
11
10
12
20
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 20

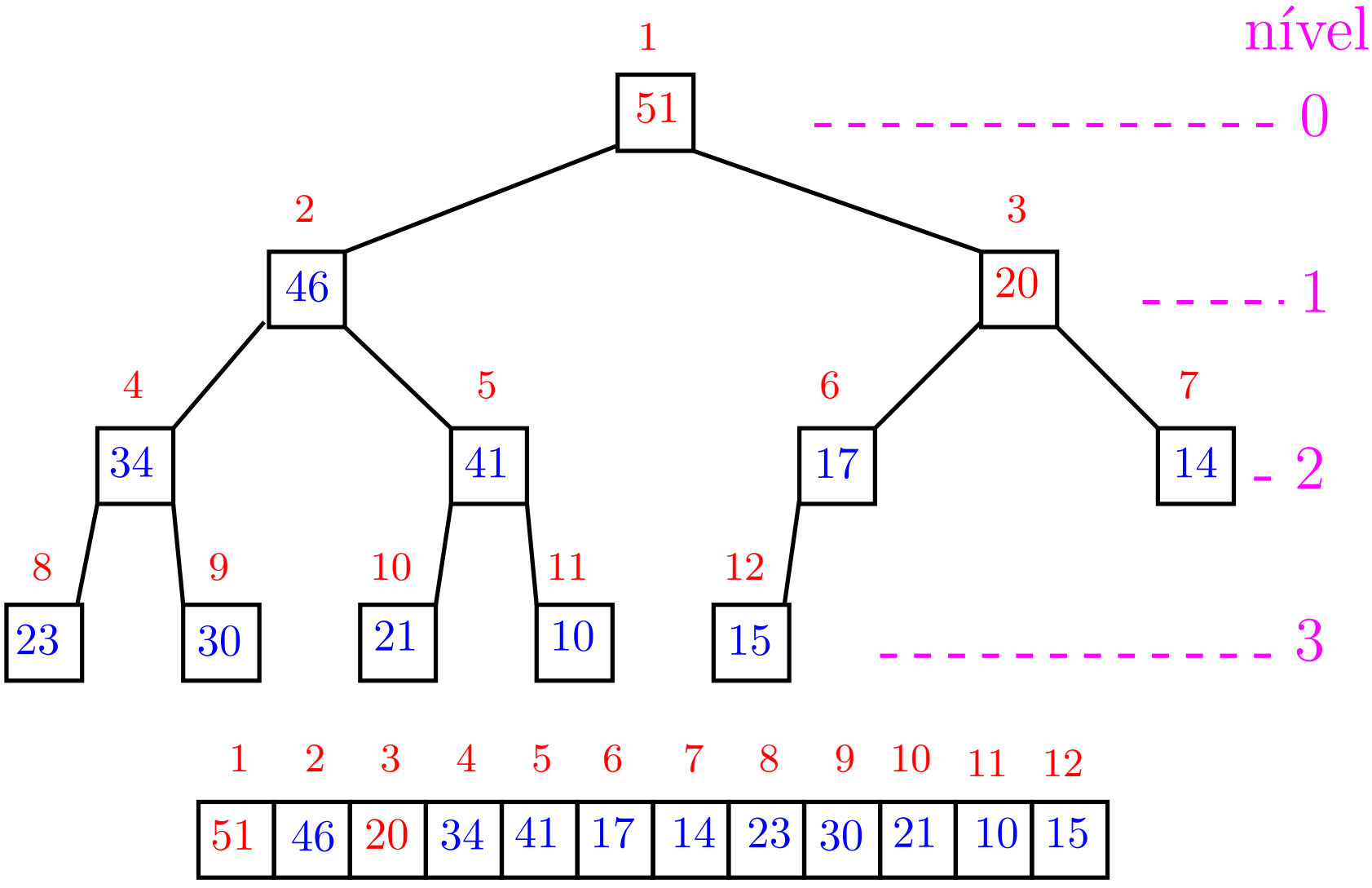
swim()


nível
1
51
0
2
46
3
17
1
4
34
5
41
6
15
7
14
2
8
23
9
30
10
21
11
10
12
20
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 15 14 23 30 21 10 20

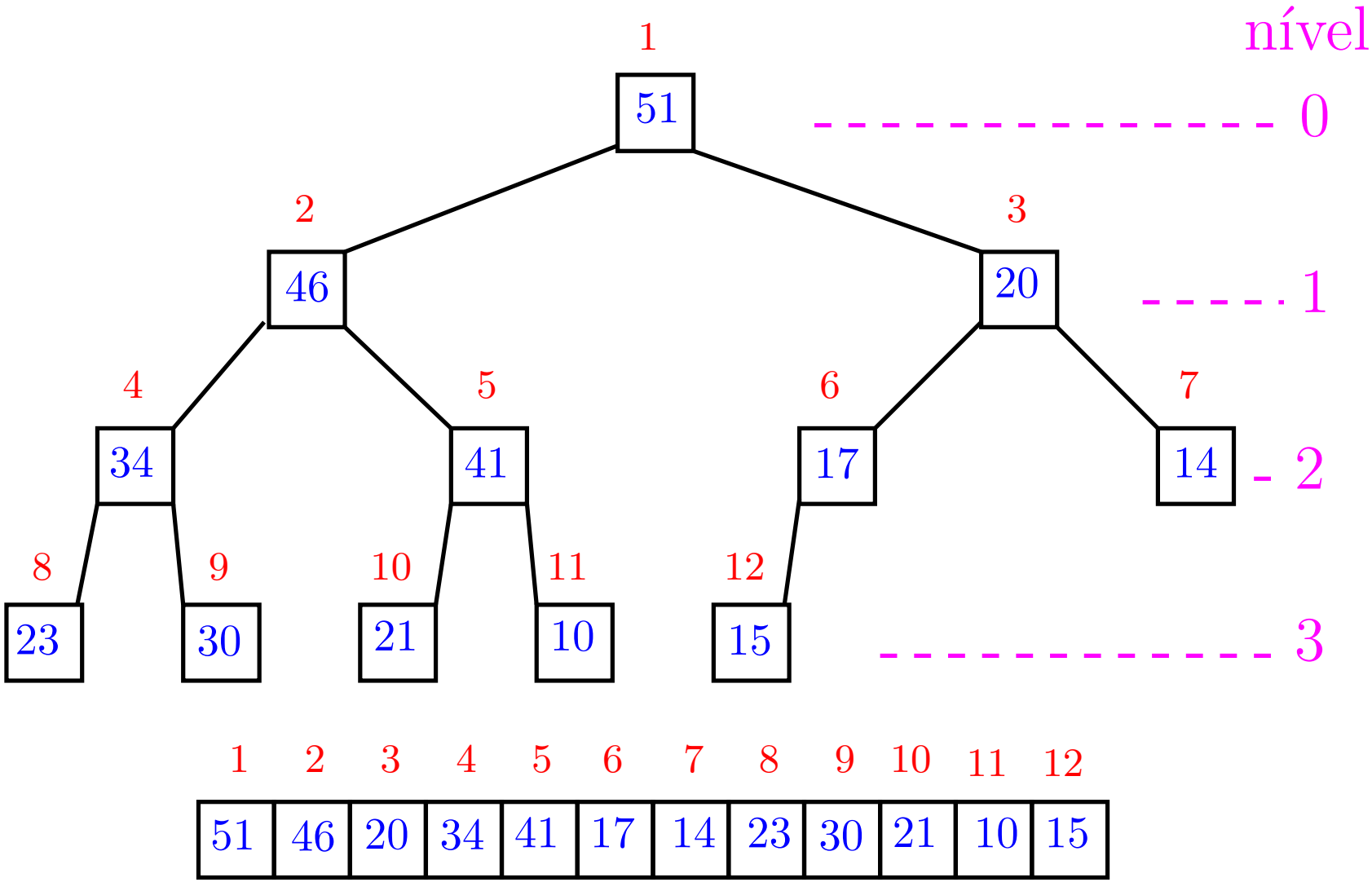
swim()


nível
1
51
0
2
46
3
17
1
4
34
5
41
6
20
7
14
2
8
23
9
30
10
21
11
10
12
15
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 17 34 41 20 14 23 30 21 10 15

swim()


nível
1
51
0
2
46
3
20
1
4
34
5
41
6
17
7
14
2
8
23
9
30
10
21
11
10
12
15
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 20 34 41 17 14 23 30 21 10 15

swim()


nível
1
51
0
2
46
3
20
1
4
34
5
41
6
17
7
14
2
8
23
9
30
10
21
11
10
12
15
3
1 2 3 4 5 6 7 8 9 10 11 12
51 46 20 34 41 17 14 23 30 21 10 15

Função swim()

Função que recebe um max-heap pq[1..m-1] de inteiros e rearranja o vetor pq[1..m] de modo que seja um max-heap.

```
static void swim (int f, int pq[]) {
1    int p = f/2; int x = pq[f];
2    while (p > 1 && pq[p] < x) {
3       pq[f] = pq[p];
4       f = p; p = f/2; /* sobe */
     }
5    pq[f] = x;
 }
```

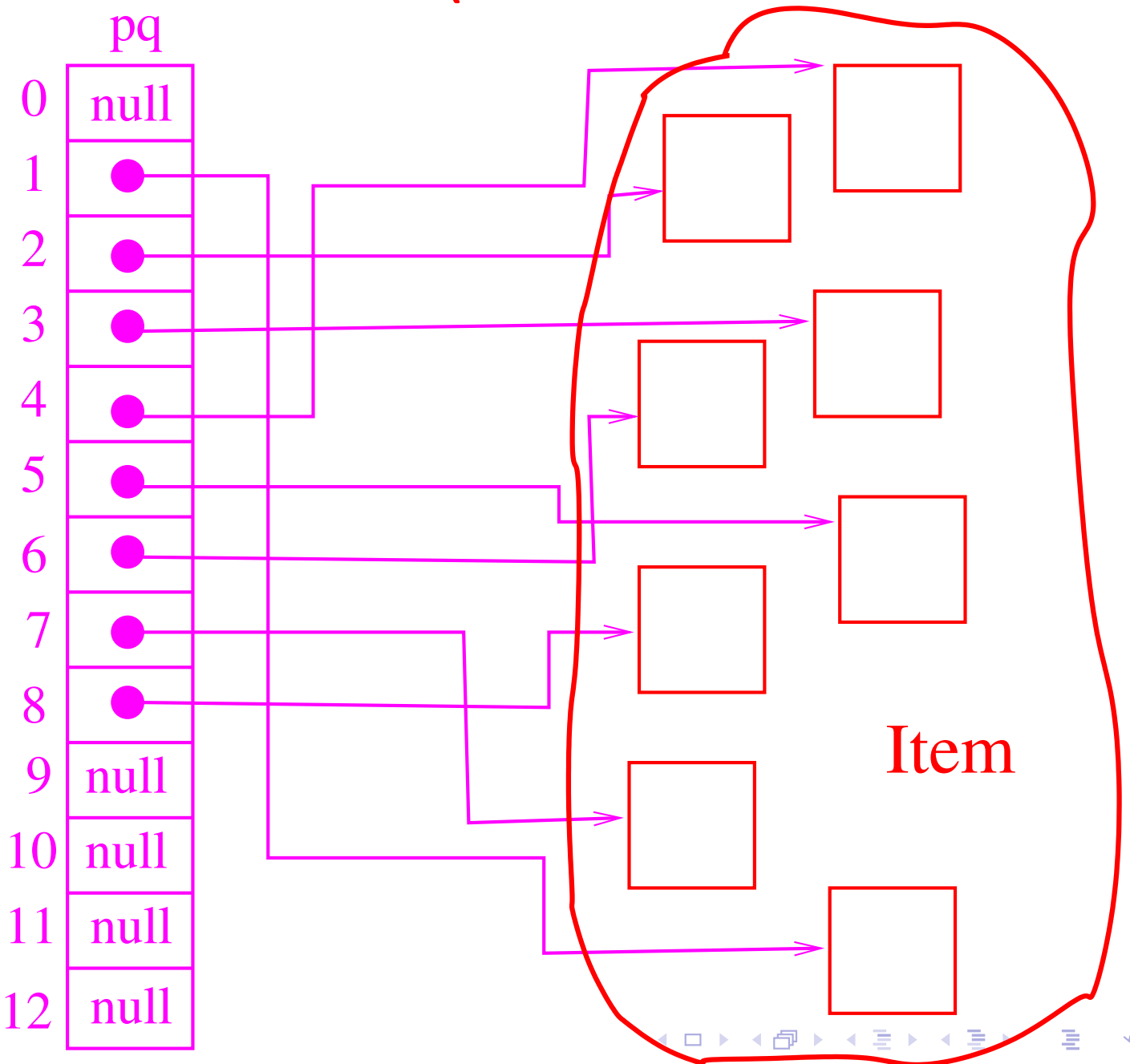
Função swim() genérica

Função que recebe um max-heap pq[1..m-1] e rearranja o vetor pq[1..m] de modo que seja um max-heap.

```
static void swim (int f, Item *pq[]) {
1    int p = f/2; Item *x;
2    x = pq[f];
2    while (p > 1 && less(pq[p], x)) {
3       pq[f] = pq[p];
4       f = p; p = f/2; /* sobe */
     }
5    pq[f] = x;
 }
```

Classe MaxPQ: estrutura

n = 8



Exemplo: `less()` do `transaction.c`

```
bool less(Transaction *a, Transaction *b)
{
   return a->valor < b->valor;
}
```

Arquivo MaxPQ.c: esqueleto

```
#include "MaxPQ.h"

static Item pq; /* heap em pq[1..n] */
static int n;

void MaxPQInit(int max) {...}
void MaxPQInsert(Item item) {...}
Item MaxPQMax() {...}
Item MaxPQDelMax() {...}
bool MaxPQEmpty() {...}
int  MaxPQSize() {...}
void MaxPQFree() {...}

static void swim(int k) {...}
static void sink(int k) {...}
```

MaxPQ: Init, Insert e Max

```
void MaxPQInit(int maxN) {
    pq = mallocSafe((maxN+1)*sizeof(Item));
    n = 0;
}

void MaxPQInsert(Item item) {
    pq[++n] = item;
    swim(n);
}

Item MaxPQMax() {
    return pq[1];
}
```

MaxPQ: DelMax, Size e Empty

```
Item MaxPQDelMax() {
   Item item = pq[1];
   pq[1] = pq[n--];
   sink(1);
   return item;
}

int MaxPQSize() {
   return n;
}

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

MaxPQ: swim genérico

```
static void swim(int f) {
   Item x = pq[f];
   while (f > 1 && less(pq[f/2], x)) {
      pq[f] = pq[f/2];
      f = f/2;
   }
   pq[f] = x;
}
```

O less depende do Item.

MaxPQ: sink genérico

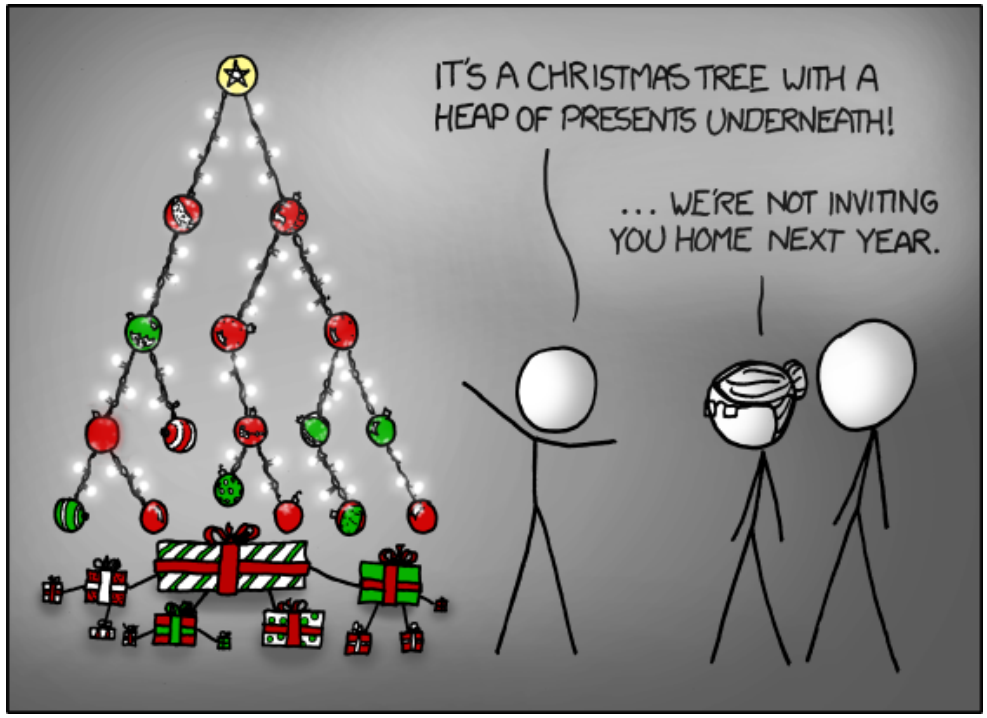
```
static void sink(int p) {
   Item x = pq[p];
   while (2*p <= n) {
      int f = 2*p;
      if (f < n && less(pq[f], pq[f+1]))
         f++;
      if (!less(x, pq[f])) break;
      pq[p] = pq[f];
      p = f;
   }
   pq[p] = x;
}
```

Conclusão

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

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

PQ com itens mutáveis



Fonte: http://xkcd.com/835/

Filas priorizadas, PF, Priority queues, S&W

PQ com itens mutáveis

Não sei se **PQ com itens mutáveis** é
um bom nome para o que S&W chamam
de *index priority queues*.

Em algumas aplicações,
é razoável permitirmos que o cliente
altere a prioridade de um item que já está na fila.

Uma maneira de lidar com isso é
associar um único índice a cada item.

Já comentamos essa estratégia
quando tratamos de union-find.

Interface para PQ-mínimo mutável

Arquivo IndexMinPQ.h

void	IMiPQinit(int maxN)	cria e inicializa
void	IMiPQinsert(int k, Item item)	insere
void	IMiPQchange(int k, Item item)	muda item
bool	IMiPQcontains(int k)	k está associado?
void	IMiPQdelete(int k)	remove k e 0 item associado
Item	IMiPQmin()	menor item
int	IMiPQminIndex()	índice do menor item
int	IMiPQdelMin()	remove menor item retorna seu índice
bool	IMiPQisEmpty()	está vazia?
int	IMiPQsize()	número de itens

IMiPQ é uma abreviatura para IndexMinPQ.

Cliente de IndexMinPQ: merge

Multiway é uma das classes do algs4,
e ela contém o método merge.

O programa intercala (merge) arquivos ordenados.

Os nomes dos arquivos (streams)
são lidos da linha de comando.

A linhas dos streams são lidas da entrada padrão.

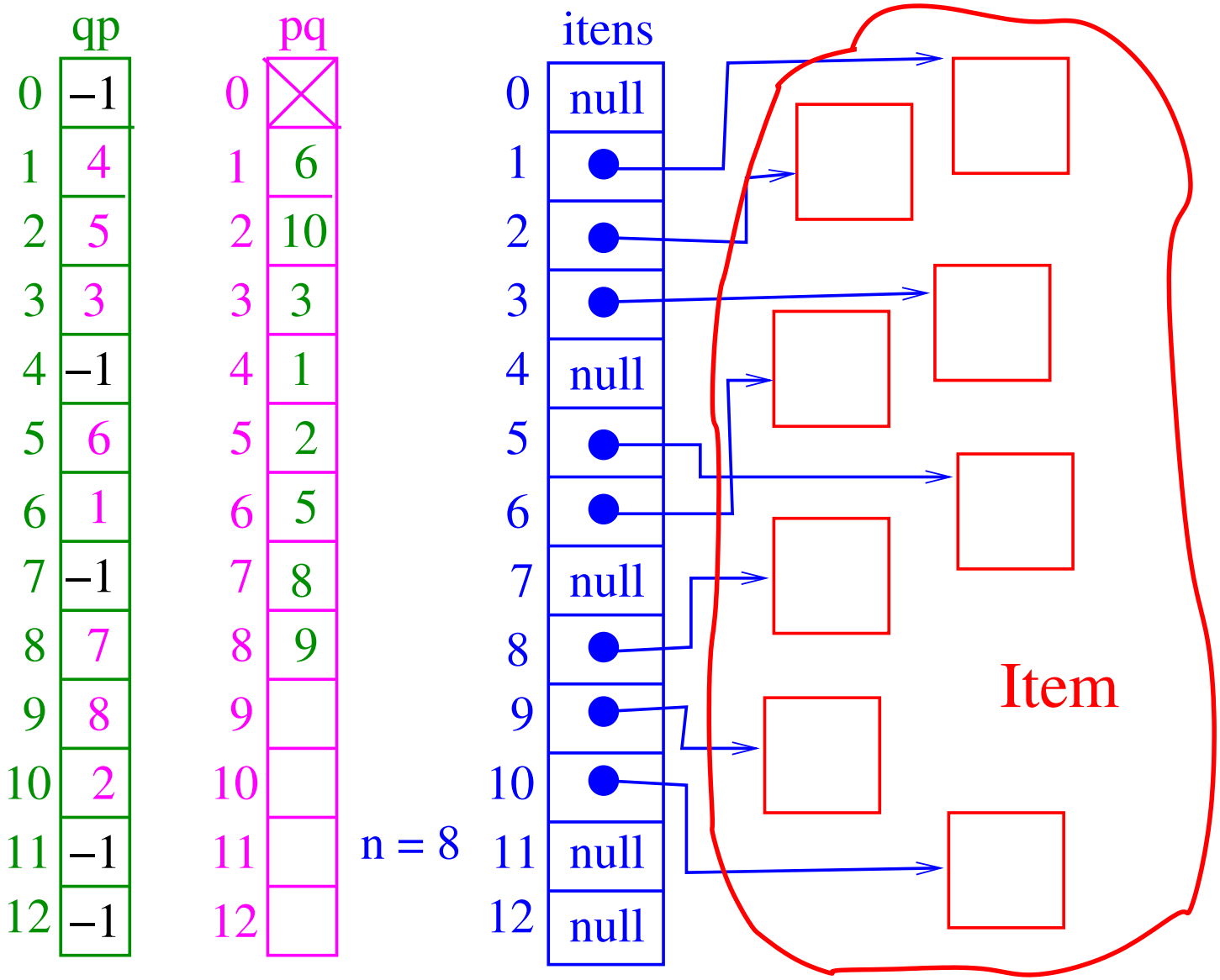
Cliente de IndexMinPQ: merge

```
/* Reads sorted text files;
 * merges them into a sorted output;
 * prints the results.
 */

void merge(int n, FILE *streams[]);

int main(int argc, char *argv) {
    int n = argc-1; FILE **streams;
    streams = mallocSafe(n*sizeof(*streams));
    for (int i = 0; i < n; i++)
        streams[i] = fopen(argv[i+1], "r");
    merge(n, streams);
}
```

Cliente de IndexMinPQ: merge

```
char *readString(FILE *f);
void merge(int n, FILE *streams[]) {
    int i;   char *t;
    IMiPQinit(n);
    for (i = 0; i < n; i++)
        if ((t = readString(streams[i])) != NULL)
            IMiPQinsert(i, t);
    while (!IMiPQisEmpty()) {
        printf("%s ", IMiPQmin());
        i = IMiPQdelMin();
        if ((t = readString(streams[i])) != NULL)
            IMiPQinsert(i, t);
    }
    printf("\n");
    IMiPQfree();
}
```

Class IndexMinPQ: estrutura



IndexMinPQ: greater() e exch()

```
typedef char *Item;

static bool greater(int i, int j) {
    Item itemI = itens[pq[i]];
    Item itemJ = itens[pq[j]];
    return greaterItem(itemI, itemJ) > 0;
}

static void exch(int i, int j) {
    Item t = pq[i];
    pq[i] = pq[j];
    pq[j] = t;
    /* para consistência */
    qp[pq[i]] = i;
    qp[pq[j]] = j;
}
```

Arquivo IndexMinPQ.c: esqueleto

```
#include "IndexMinPQ.h"

static int *pq;    /* heap em pq[1..n] */
static int *qp;    /* qp[pq[i]] = pq[qp[i]] = i*/

static Item *itens;
static int n;

void init(int max) {...}
bool isEmpty() {...}
int  size() {...}
Item min() {...}
int  delMin() {...}
int  minIndex() {...}
```

Arquivo IndexMinPQ.c: esqueleto

```
void insert(int k, Item a) {...}
void delete(int k) {...}
void change(int k, Item item) {}
bool contains(int k) {...}

/* métodos administrativos */
static void swim(int k) {...}
static void sink(int k) {...}
static bool less(int i, int j){...}
static void exch(int i, int j){...}
```

IndexMinPQ: estruturas e init

```
static int *pq;
static int *qp;

static Item *itens;
static int n;

void init(int maxN) {
    pq = mallocSafe((maxN+1)*sizeof(int));
    qp = mallocSafe((maxN+1)*sizeof(int));
    itens = mallocSafe((maxN+1)*sizeof(Item));
    for (int i = 0; i <= maxN; i++) {
        qp[i] = -1;
        itens[i] = NULL;
    }
    n = 0;
}
```

IndexMinPQ: isEmpty() e size()

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

int size() {
   return n;
}
```

IndexMinPQ: insert() e contains()

```
void insert(int k, Item item) {
    n++;
    itens[k] = item;
    pq[n] = k;
    qp[k] = n;
    swim(n);
}

bool contains(int k) {
    return qp[k] != -1;
}
```

IndexMinPQ: delMin() e min()

```
int delMin() {
   int indexOfMin = pq[1];
   exch(1, --n);
   sink(1);
   itens[pq[n+1]] = NULL; /* loitering */
   qp[pq[n+1]] = -1;
   return indexOfMin;
}

Item min() {
   return itens[pq[1]];
}
```

IndexMinPQ: minIndex() e change()

```
int minIndex() {
   return pq[1];
}

void change(int k, Item item) {
   itens[k] = item;
   swim(qp[k]);
   sink(qp[k]);
}
```

IndexMinPQ: delete()

```
void delete(int k) {
   int j = pq[n];
   exch(qp[k],n--);
   /* arruma heap */
   sink(qp[j]);
   swim(qp[j]);
   /* destroi o rastros de k */
   itens[k] = null;
   qp[k] = -1;
}
```

IndexMinPQ: swim() e sink()

```
static void swim(int f) {
    while (f > 1 && greater(f/2, f)) {
        exch(f/2, f);
        f = f/2;
    }
}

static void sink(int p) {
    while (2*p <= n) {
        int f = 2*p;
        if (f < n && greater(f, f+1)) f++;
        if (!greater(p, f)) break;
        exch(p, f);
        p = f;
    }
}
```

Conclusão

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

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