MAC0323 Algoritmos e Estruturas de Dados II

Edição 2020 – 2

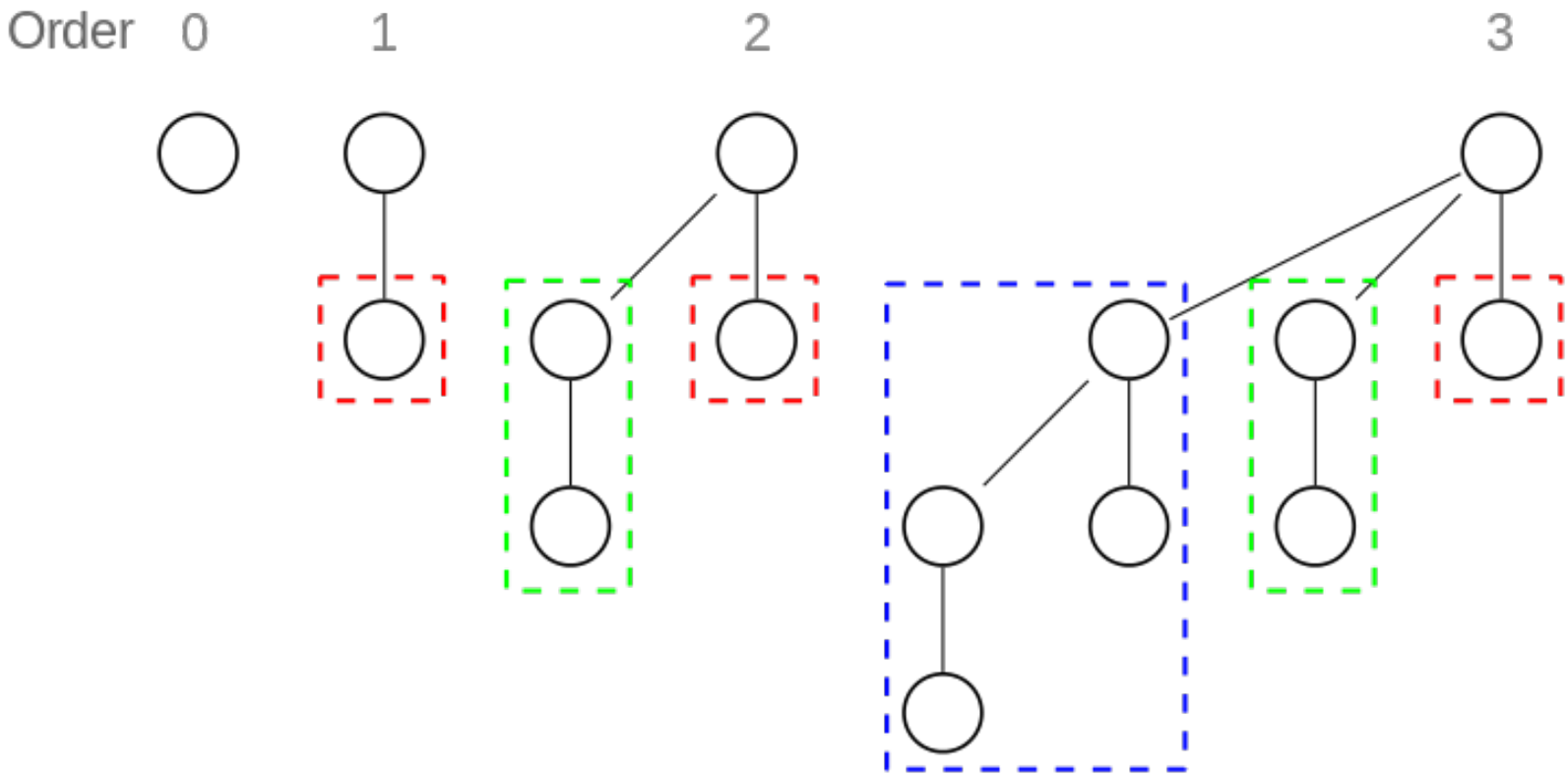
Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 7

AULA 7

Binomial heaps



Fontes: Wikipedia,

Foundations of Data Science

CLRS 19

Binomial trees: estrutura

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

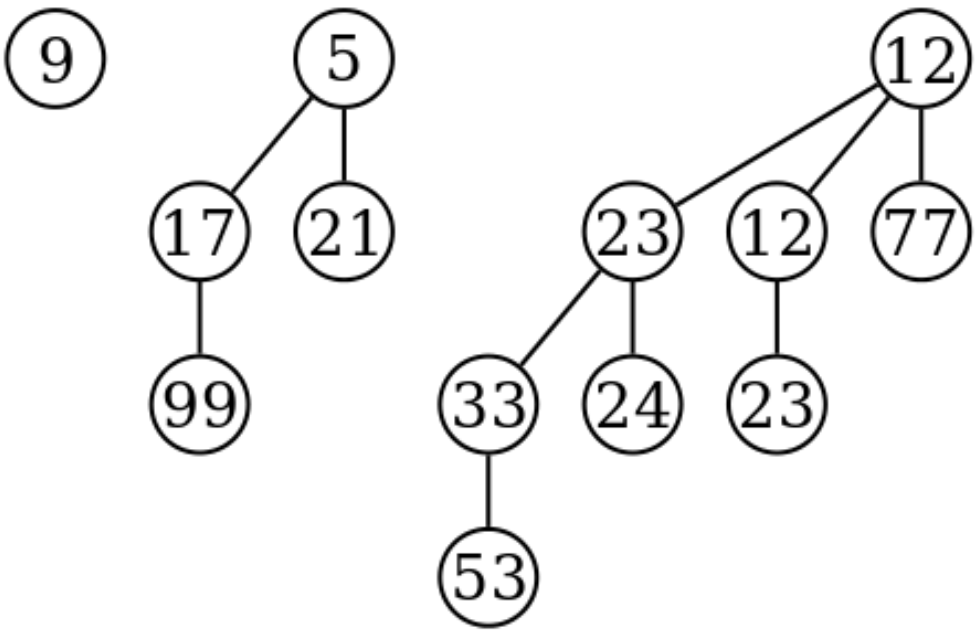
- a raiz tem k filhos;
- a árvore tem 2^k nós;
- a árvore tem altura k;
- há exatamente $\binom{k}{i}$ com profundidade i;
- os filhos da raiz são as árvores binomiais B_{k-1}, B_{k-2}, B_{k-3},...

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`;
- `H` tem no máximo uma binomial tree de cada ordem.

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

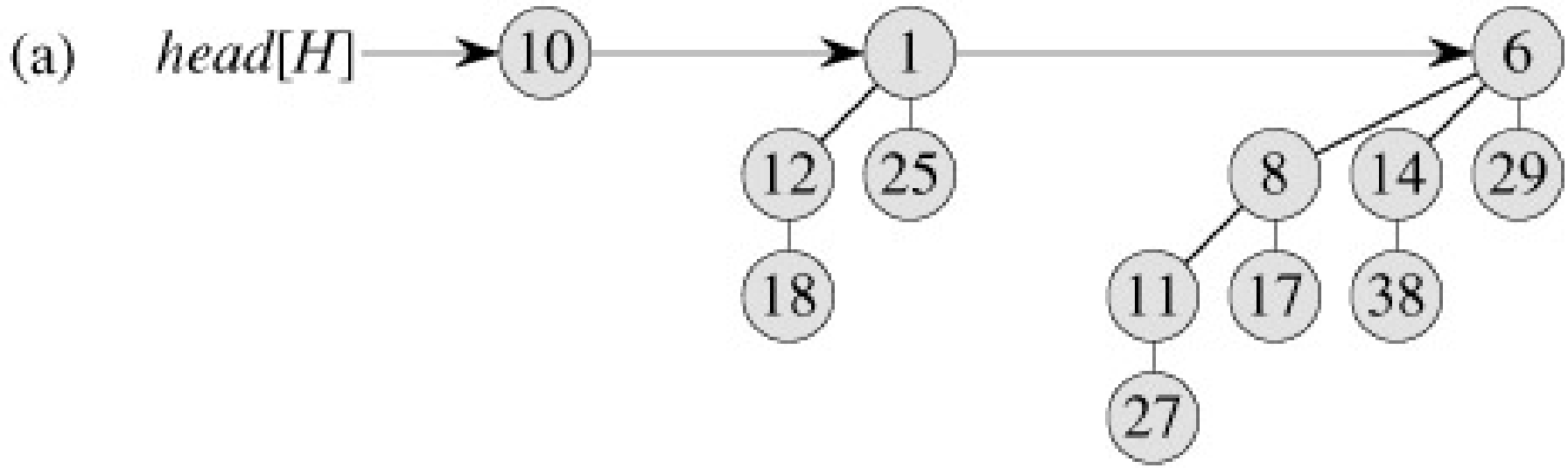


Fonte: Wikipedia

AULA 8

Binomial heap: estrutura de dados

Fonte: CLRS

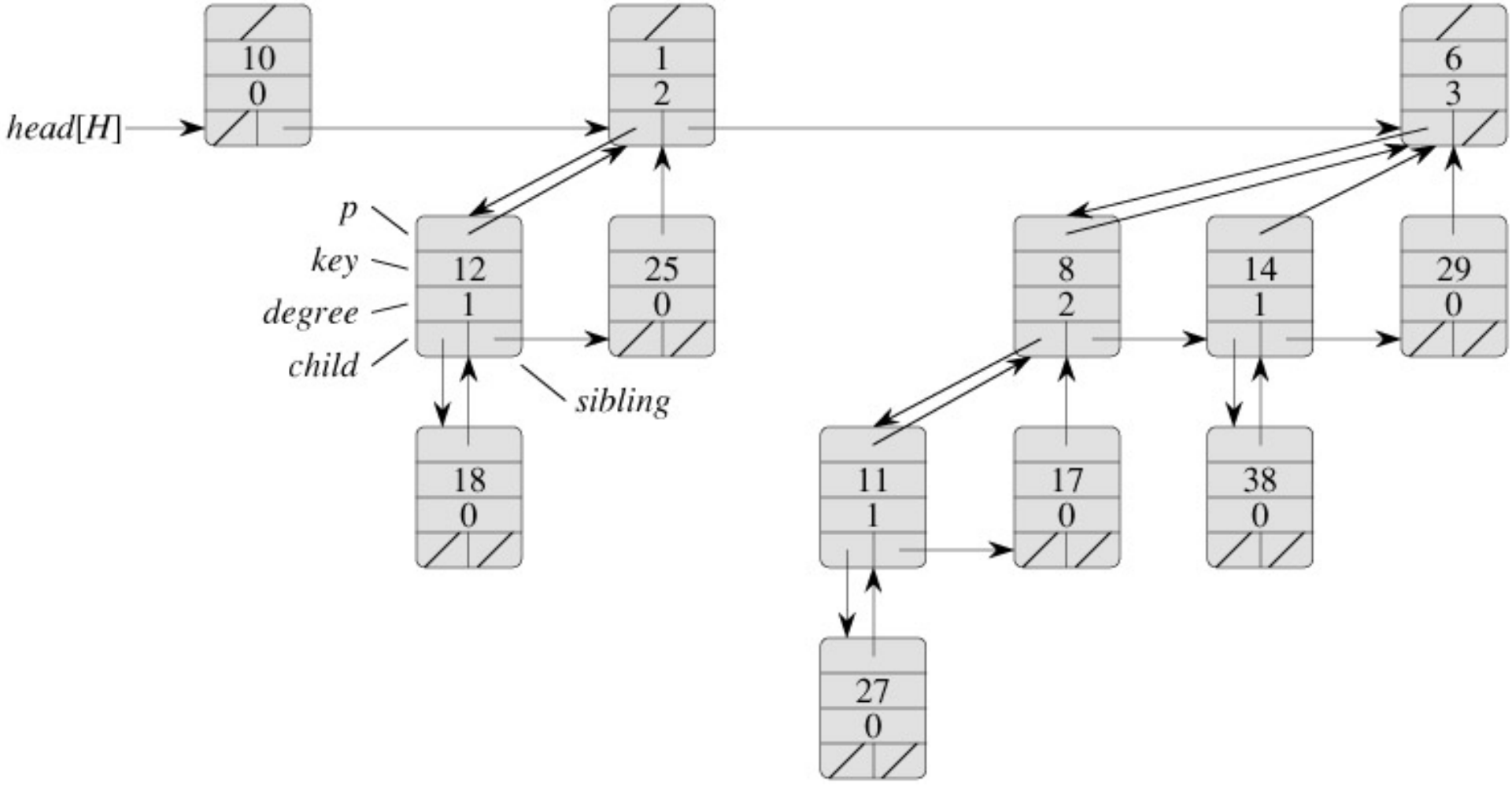


BinomialMinPQ: struct node e Link

Representação de um nó de uma binomial tree.

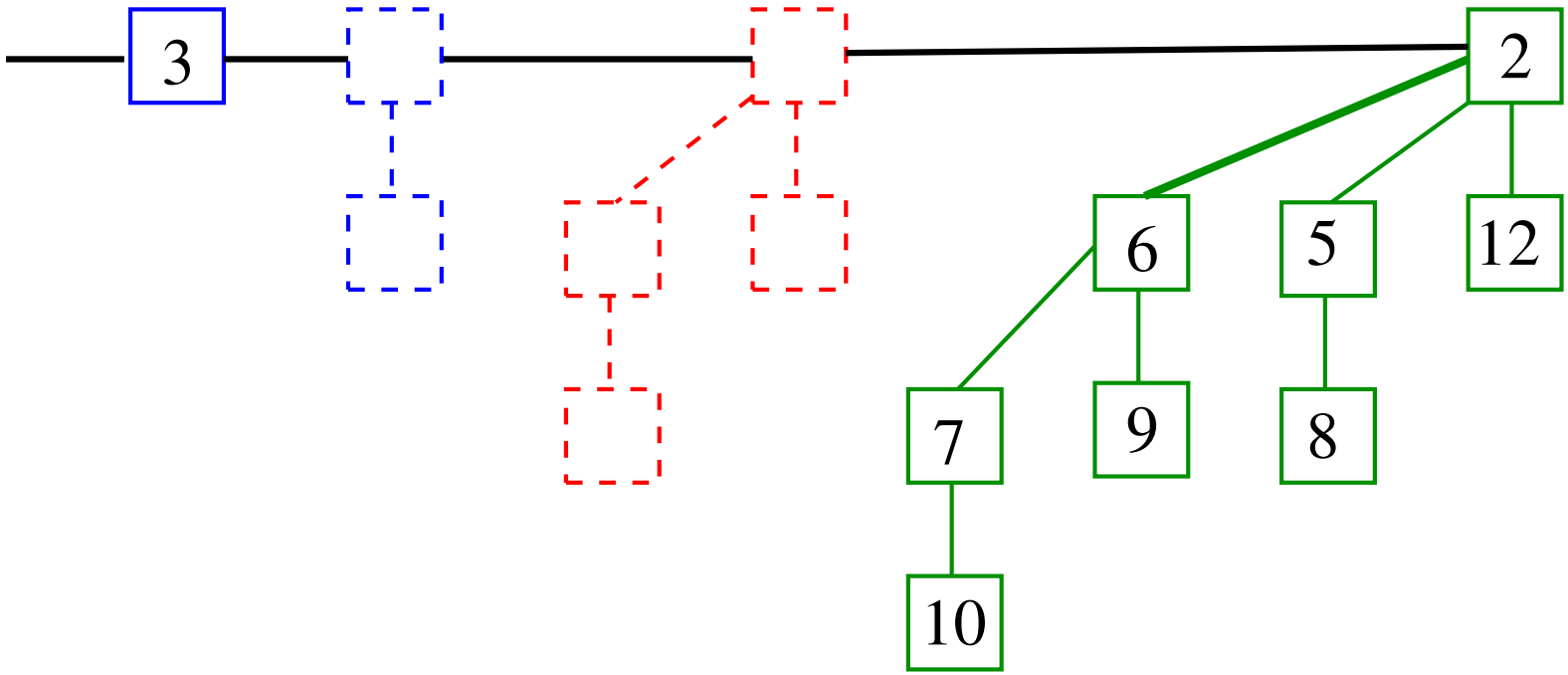
```
typedef struct node *Link;
struct node {
   Item item;
   Link child,    /* filho mais a esquerda */
   Link sibling, /* lista de irmãos */
   Link parent;  /* pai */
   int  order;   /* ou grau, degree */
};
```

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

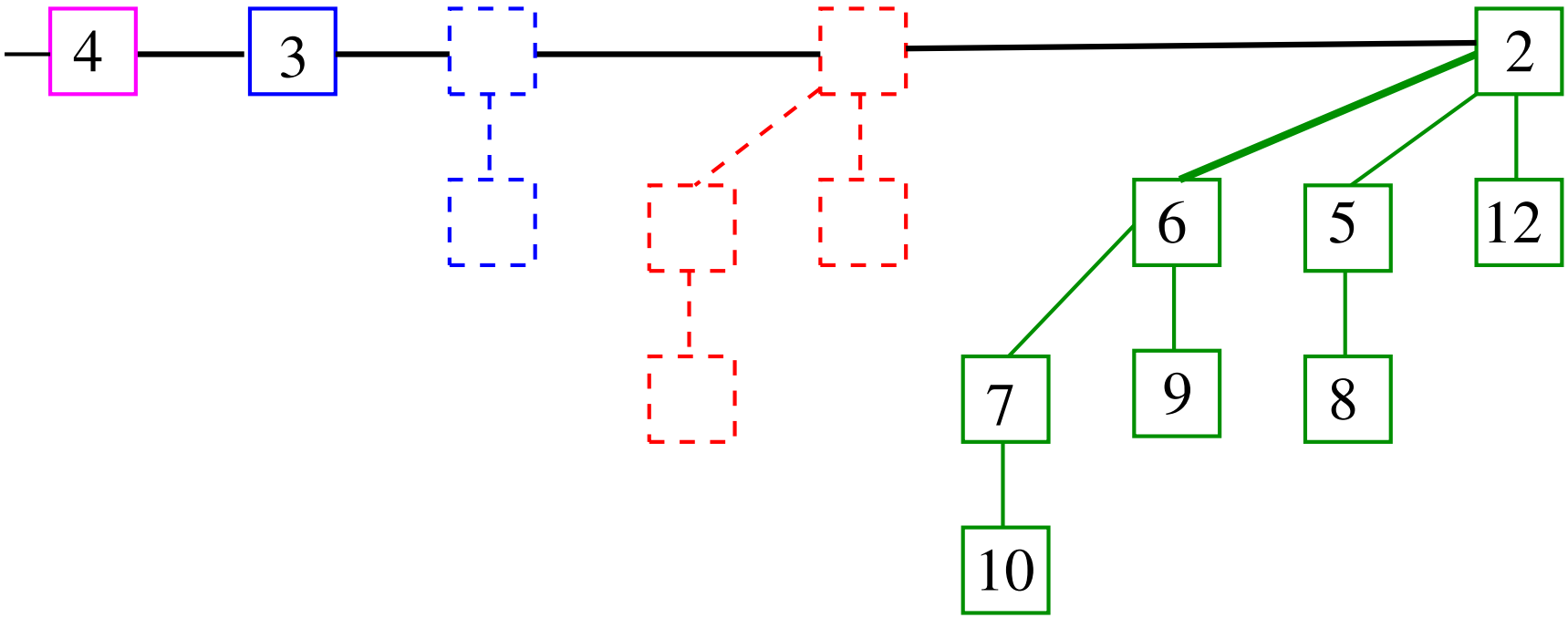

head[H]
10
0
1
2
6
3
p
key
degree
child
sibling
12
1
25
0
8
2
14
1
29
0
18
0
11
1
17
0
38
0
27
0

Fonte: CLRS

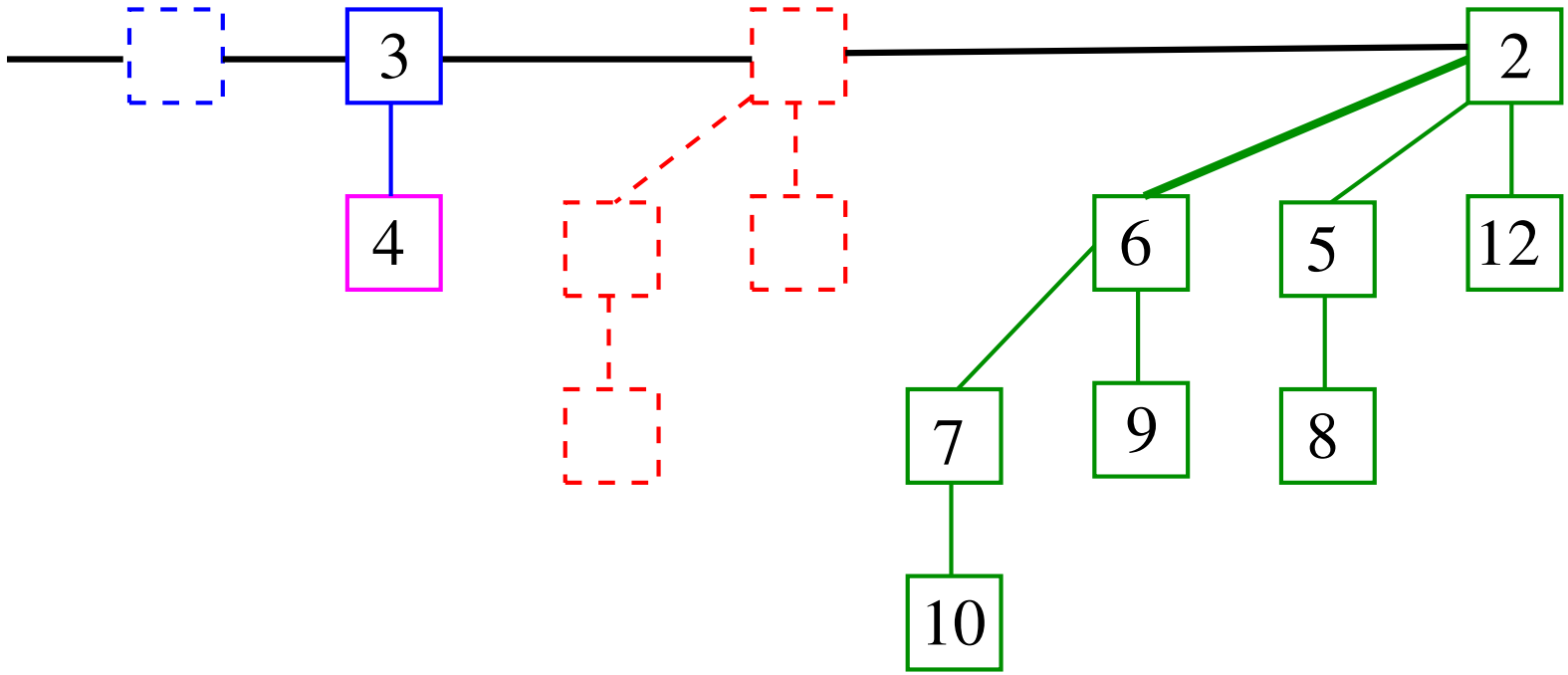
BinomialMinPQ: insert()



BinomialMinPQ: insert()



BinomialMinPQ: insert()

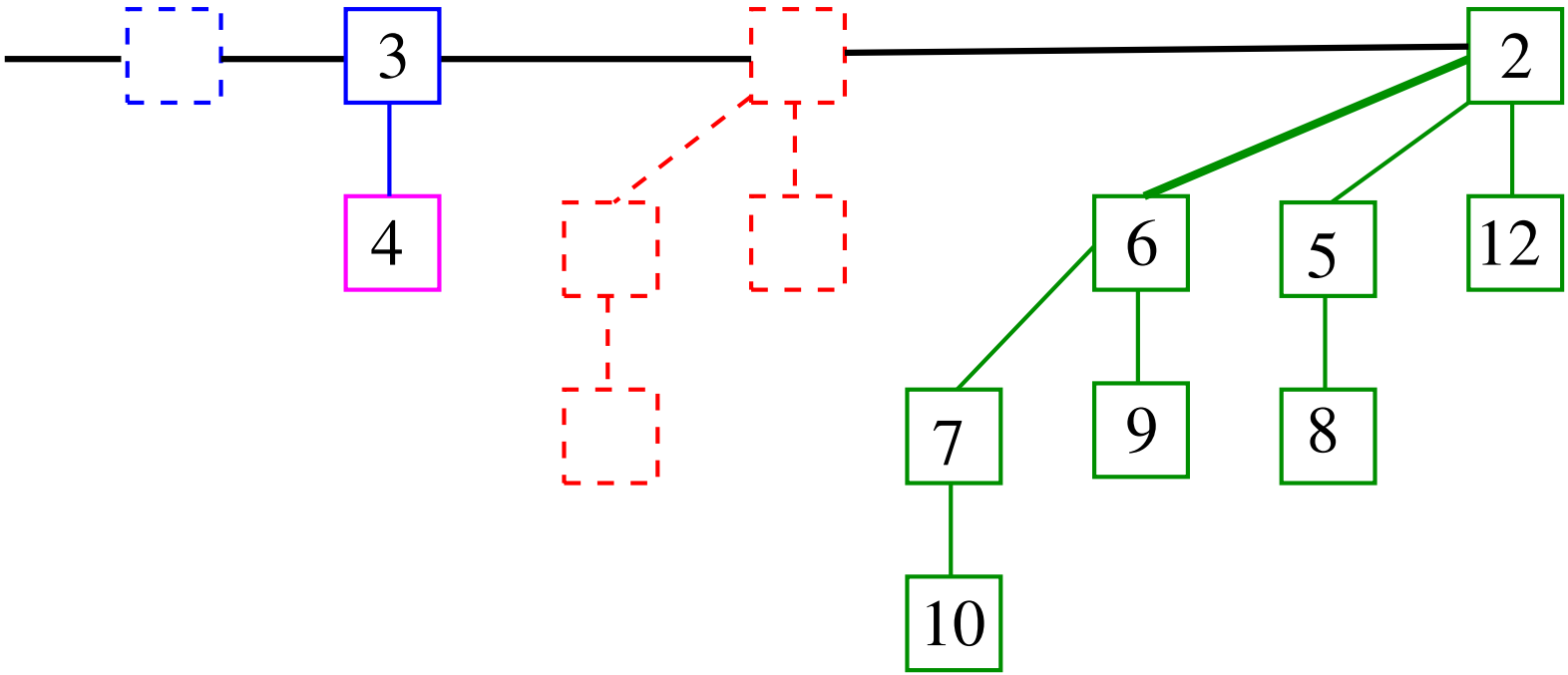


BinomialMinPQ: insert()

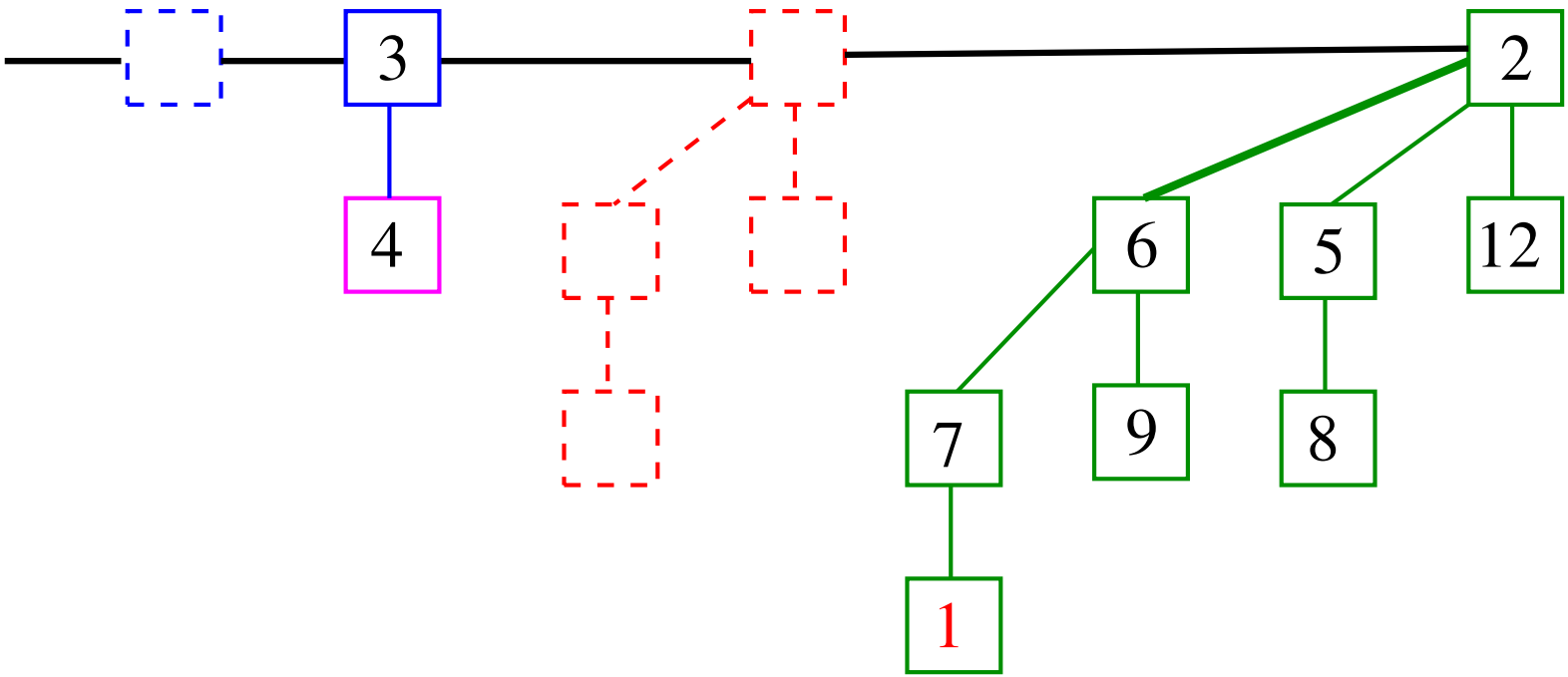
Como existem $1 + \lfloor \lg n \rfloor$ binomial trees
que podem ser unidas concluímos o seguinte.

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

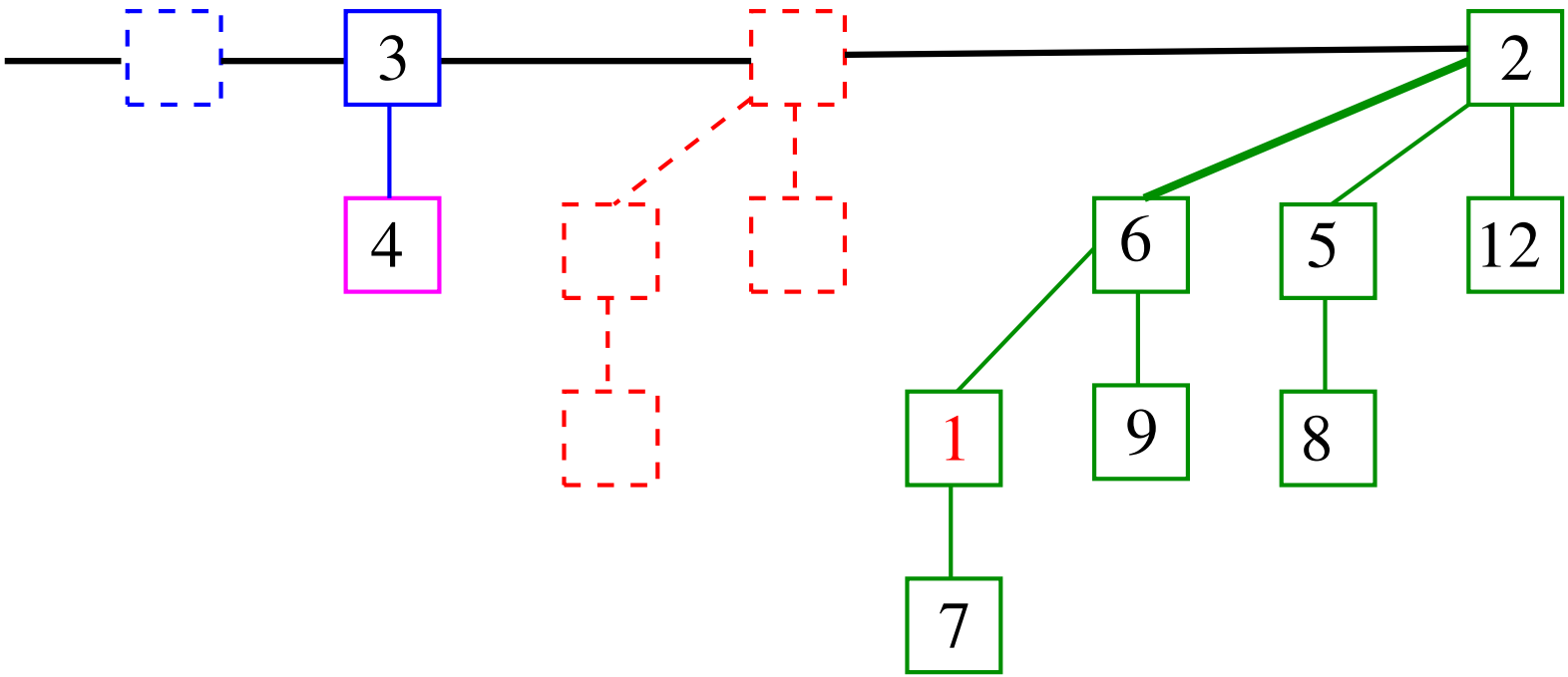
BinomialMinPQ: change()



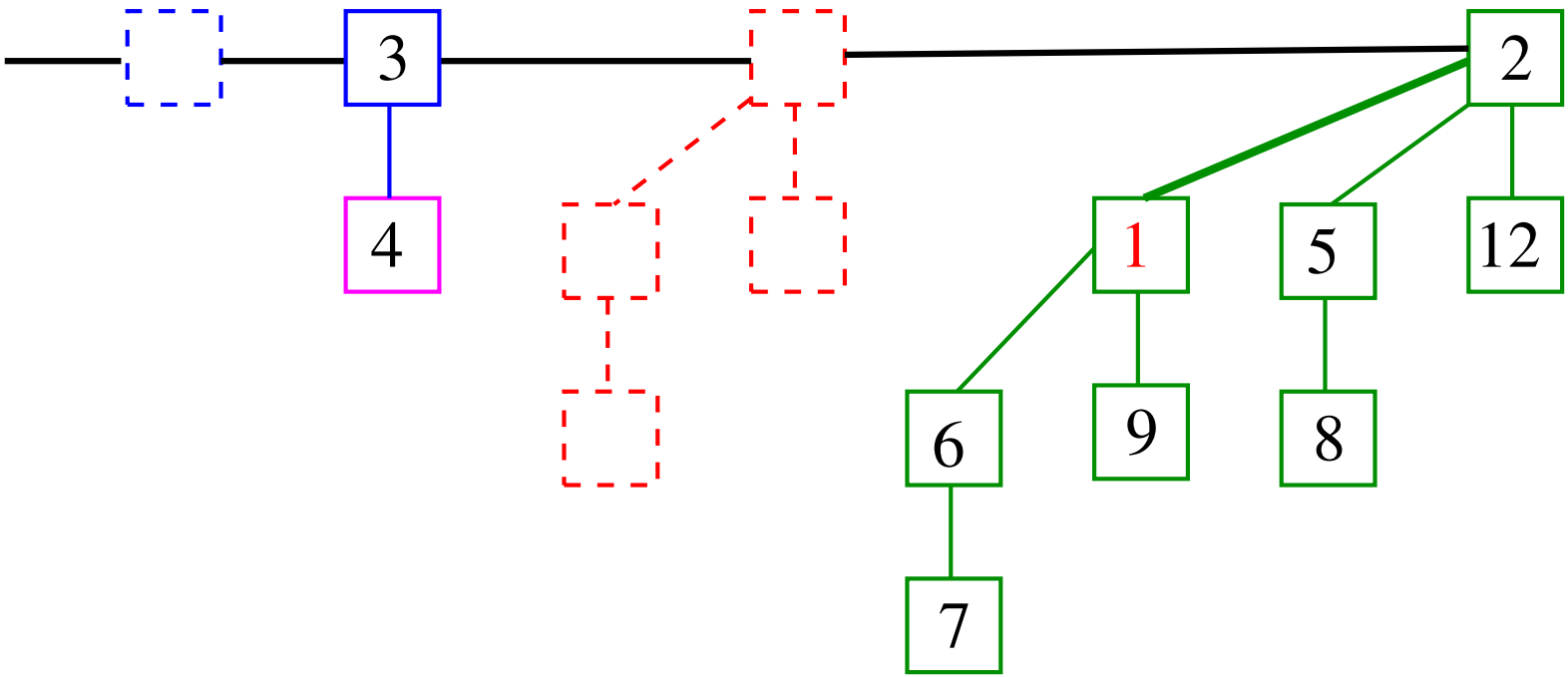
BinomialMinPQ: change()



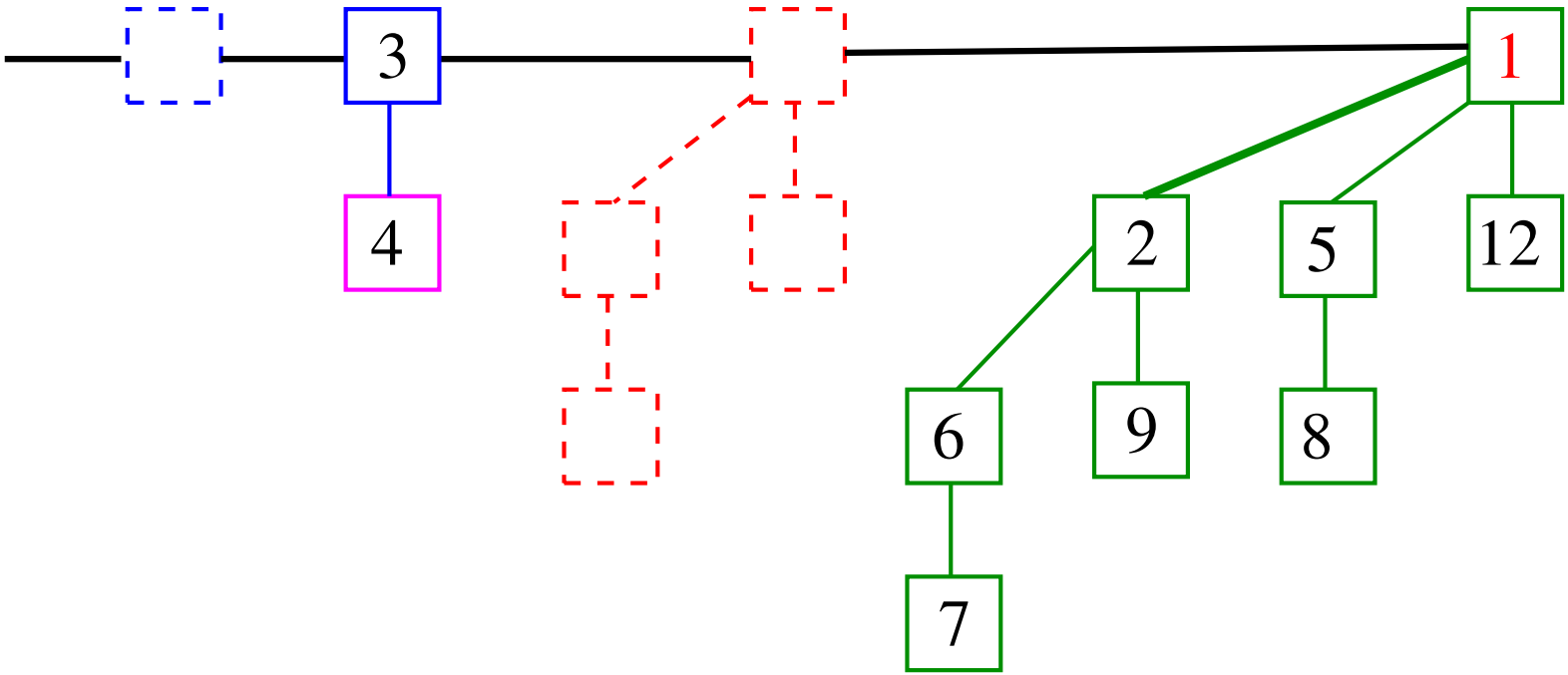
BinomialMinPQ: change()



BinomialMinPQ: change()



BinomialMinPQ: change()



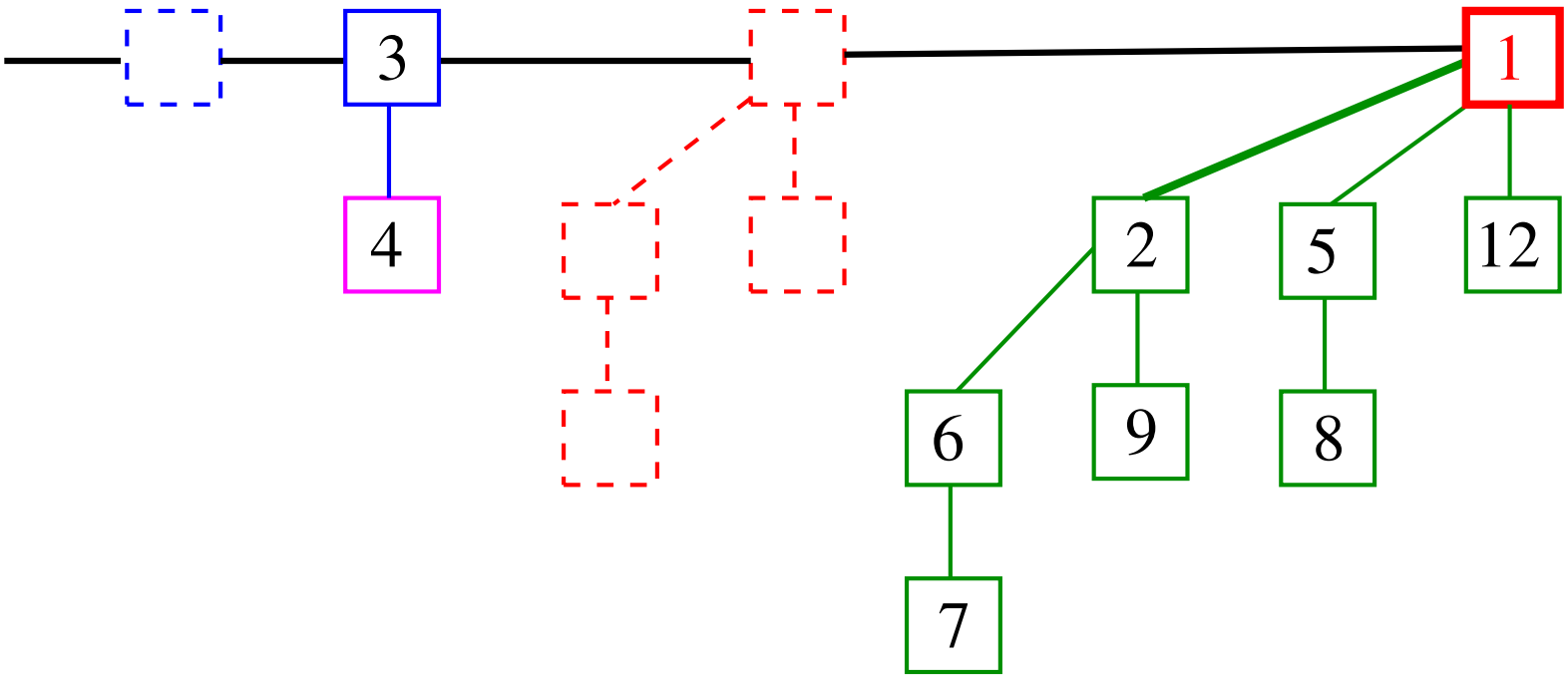
BinomialMinPQ: change()

Como a maior altura de uma árvore na binomial heap é $\lfloor \lg n \rfloor$ concluímos o seguinte.

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

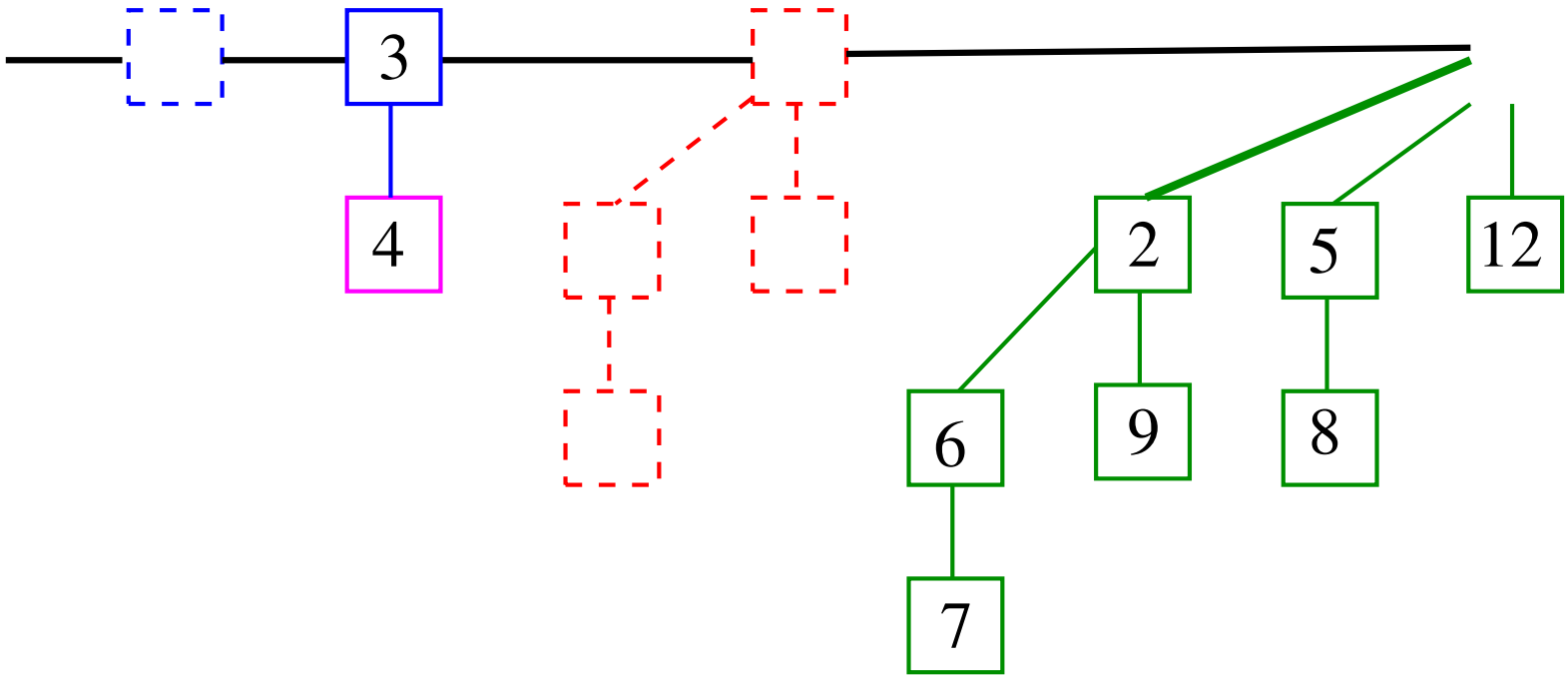
`BinomialMinPQ`: `delMin()`

Percorra a lista das raízes e encontre o menor `item`.



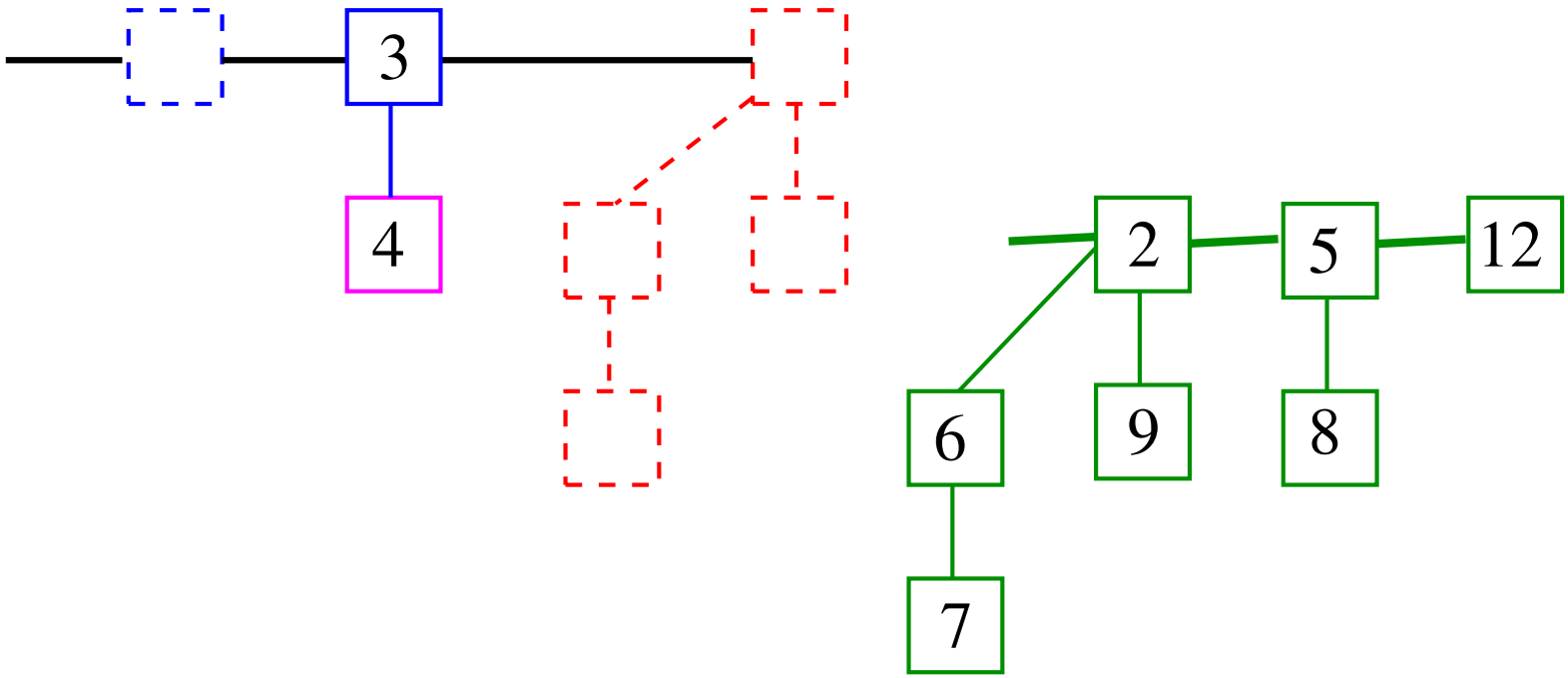
BinomialMinPQ: delMin()

Remova o nó.



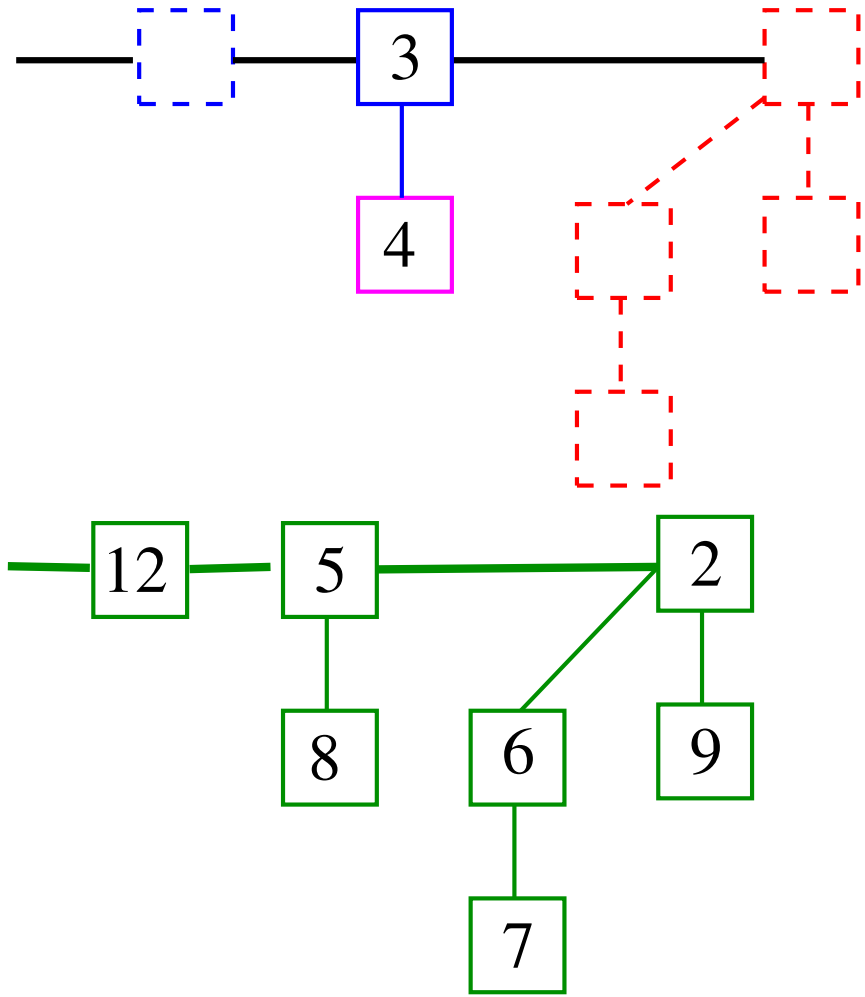
BinomialMinPQ: delMin()

Inverta a lista dos seus filhos.



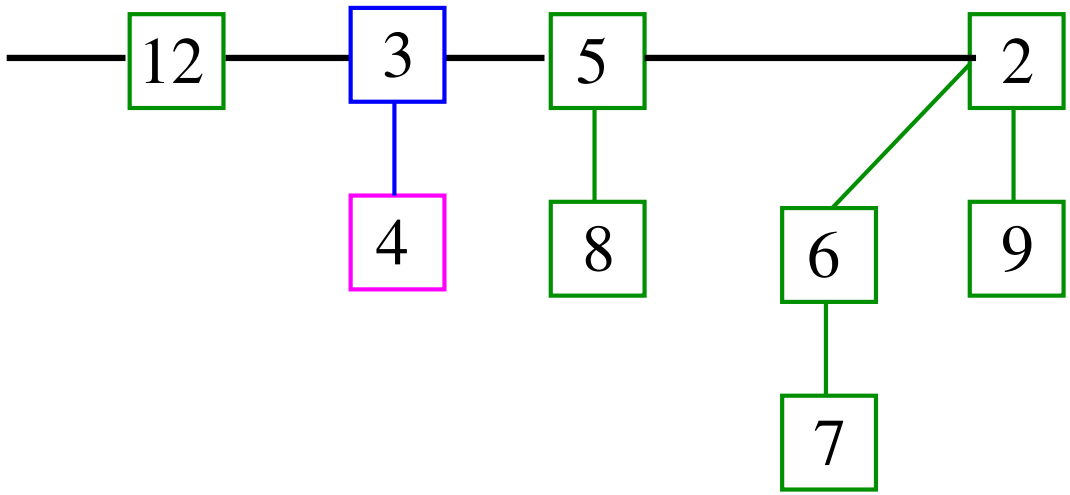
BinomialMinPQ: delMin()

Inverta a lista dos seus filhos.



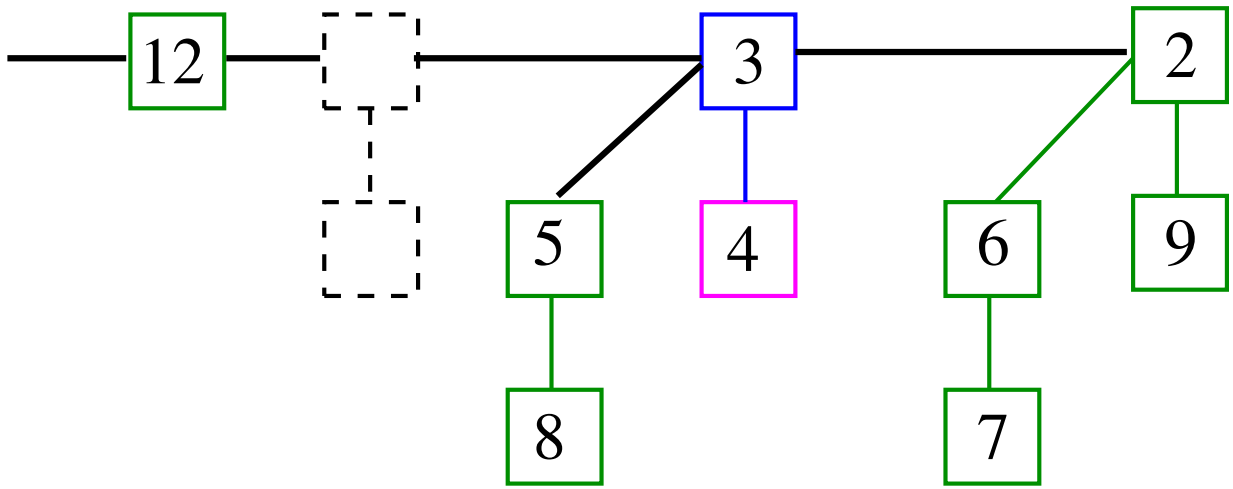
BinomialMinPQ: delMin()

Faça merge() das duas binomial heaps.



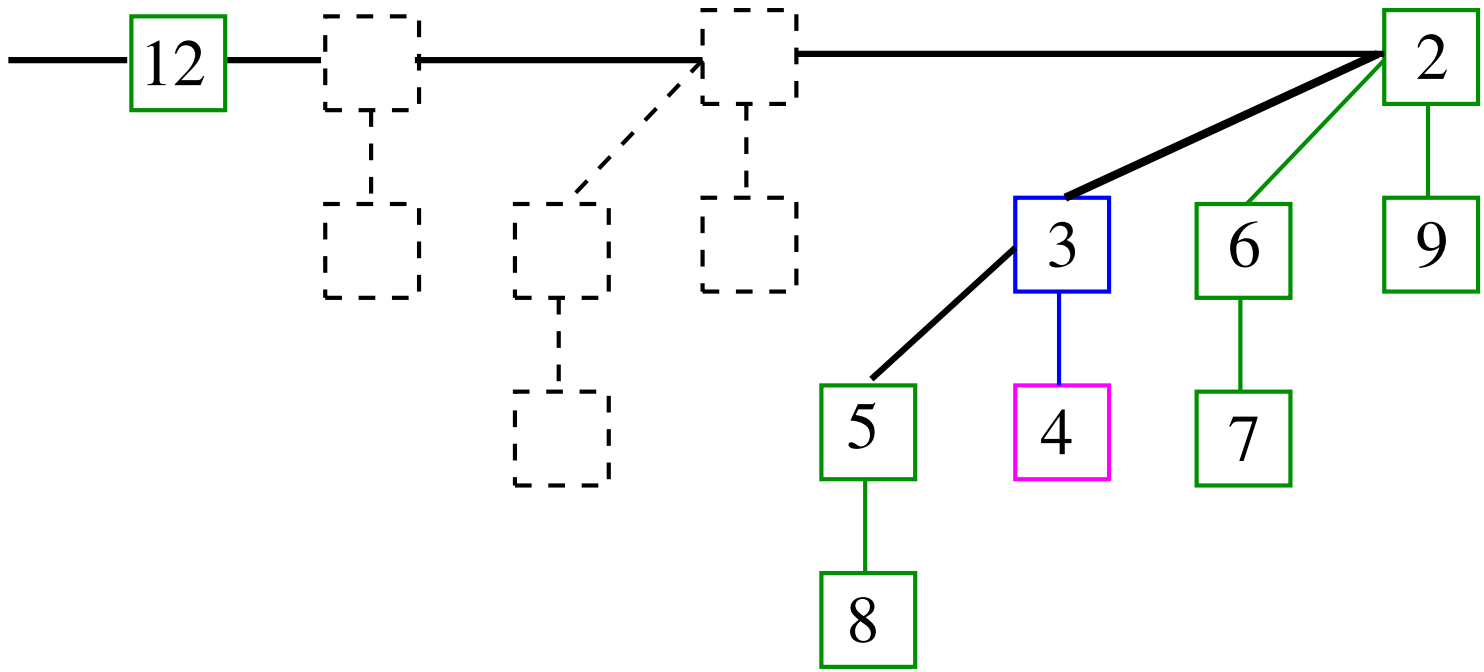
BinomialMinPQ: delMin()

Faça merge() das árvores de mesma ordem.



BinomialMinPQ: delMin()

Faça merge() das árvores de mesma ordem.



BinomialMinPQ: delMin()

O consumo de tempo para
encontrar o menor item é $\mathrm{O}(\lg \mathtt{n})$.

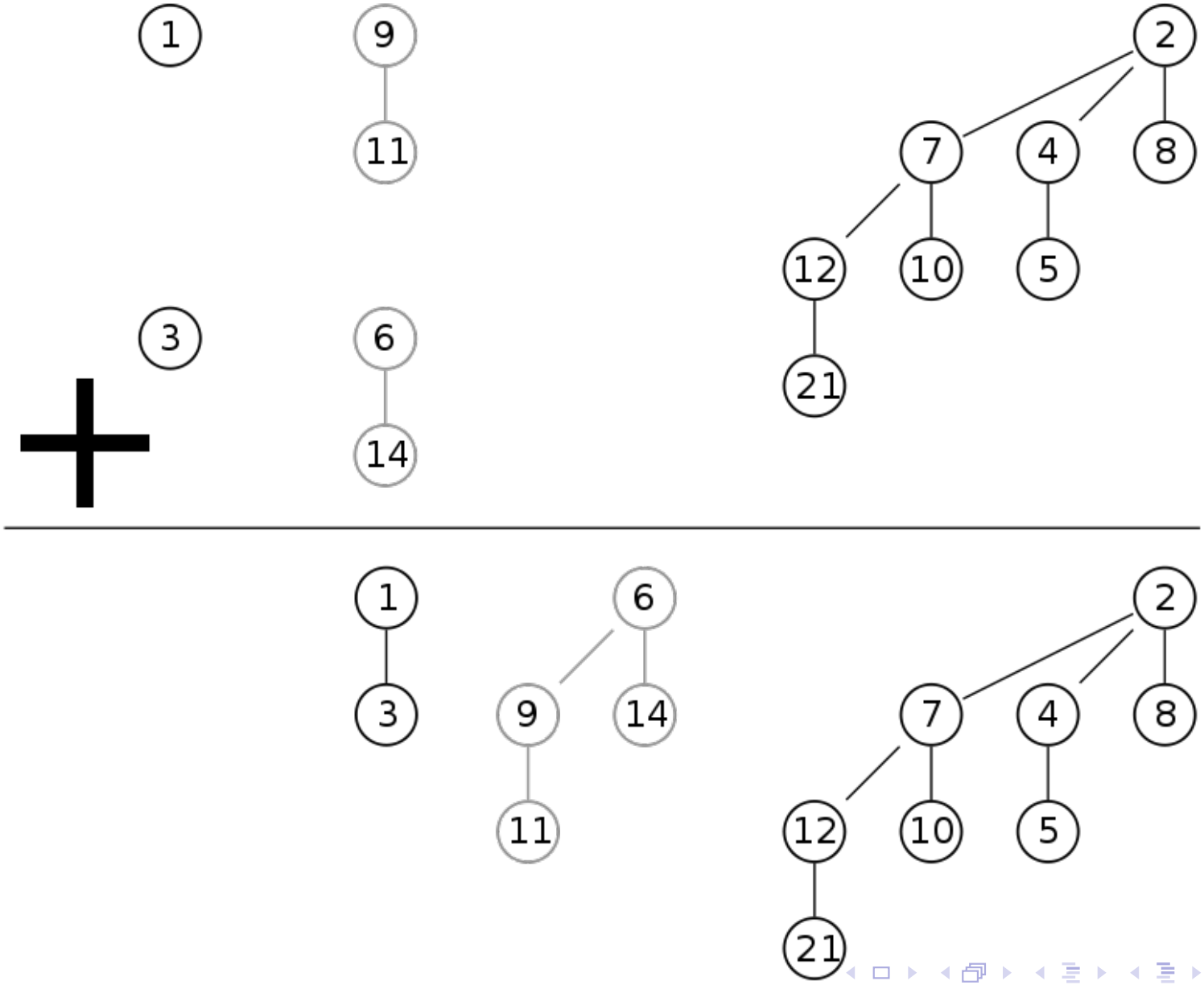
O consumo de tempo para inverter e
intercalar listas com até $1 + \lfloor \lg \mathtt{n} \rfloor$ itens é $\mathrm{O}(\lg \mathtt{n})$.

Finalmente, o consumo de tempo para
unir árvores de mesma ordem é $\mathrm{O}(\lg \mathtt{n})$.

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

BinomialMinPQ: merge()

Fonte: Wikipedia



BinomialMinPQ: struct node e Link

Cada nó da árvore binomial terá quatro campos.

```
typedef struct node *Link;
struct node {
    Transaction *item;
    Link child, sibling;
    int order;
};

Link newNode(Transaction *item, Link child,
             Link sibling, int order){
    Link p = mallocSafe(sizeof(*p));
    p->item = item;    p->order = order;
    p->sibling = sibling; p->child = child;
    return p;
}
```

Arquivo BinomialMinPQ.c: esqueleto

Igual ao do LMinPQ.

```
#include "MinPQ.h"

static Link head;
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) {...}
```

BinomialMinPQ: init, isEmpty e size

Igual ao do LMinPQ.

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

int MinPQSize() {
   return n;
}

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

BinomialMinPQ: insert()

```
void MinPQInsert(Item item) {
   Link s = newNode(item, NULL, NULL, 0);
   head = merge(head, s);
   n++;
}
```

BinomialMinPQ: delMin()

```
Item MinPQDelMin() {
   Link x, prevx, nextx, min;
   Item item;

   /* remova minimo do heap */
   min = eraseMin();
   item = min->item;
   n--;

   if (min->child != NULL) {
      /* inverta lista de filhos do min */
      x = min->child;
      prevx = NULL;  nextx = x->sibling;
```

BinomialMinPQ: delMin()

```
        while (nextx != NULL) {
            x->sibling = prevx;
            prevx = x;
            x = nextx;
            nextx = nextx->sibling;
        }
        x->sibling = prevx;
        head = merge(head, x)
    }
    freeNode(min);
    return item;
}
```

BinomialMinPQ: eraseMin()

Remove um mínimo da lista da raiz e seus siblings.

```
static Link eraseMin() {
    Link min = head, prev = NULL, current = head;
    while (current->sibling != NULL) {
        if (less(current->sibling->item, min->item)){
            prev = current;
            min = current->sibling;
        }
        current = current->sibling;
    }
    if (min == head) head = min->sibling;
    else prev->sibling = min->sibling;
    return min;
}
```

BinomialMinPQ: link()

As duas próximas rotinas serão usadas no merge.

Supõe que r2->item < r1->item
e r2->order == r1->order.

link(r1, r2) inclui r1 como primeiro filho de r2.

```
static void link(Link r1, Link r2) {
   r1->sibling = r2->child;
   r2->child = r1;
   (r2->order)++;
}
```

BinomialMinPQ: mergeR()

mergeR(r1, r2) intercala as listas ligadas dos caminhos sibling de r1 e r2:

```
static Link mergeR(Link r1, Link r2) {
   if (r1 == null) return r2;
   if (r2 == null) return r1;
   if(r1->order < r2->order) {
      Link t = r1;  r1 = r2;  r2 = t;
   }
   r1->sibling = mergeR(r1->sibling, r2);
   return r1;
}
```

Para o merge, falta juntarmos as árvores repetidas.

BinomialMinPQ: merge()

```
static Link merge(Link r1, Link r2) {
    Link r, x, prevx, nextx;
    if (r1 == null) return r2;
    if (r2 == null) return r1;
    r = mergeR(r1, r2);
    x = r;
    prevx = NULL;  nextx = x->sibling;
    while (nextx != NULL) {
        if (x->order < nextx->order ||
            (nextx->sibling != NULL &&
             nextx->sibling->order == x->order)) {
            /* não repete ou tem três repetidas */
               prevx = x;  x = nextx;
            }
        else ...
```

BinomialMinPQ: merge()

```
        else if (less(x->item, nextx->item)) {
            /* nextx vira filho de x */
            x->sibling = nextx->sibling;
            link(nextx, x);
        }
        else { /* x vira filho de nextx */
            if (prevx == NULL) r = nextx;
            else prevx->sibling = nextx;
            link(x, nextx);
            x = nextx;
        }
        nextx = x->sibling;
    }
    return r;
}
```

`BinomialMinPQ`: Conclusões

Em uma binomial heap, o consumo de tempo das operações `insert()`, `delMin()`, `change()` e `union()` no pior caso é $\mathrm{O}(\lg \mathtt{n})$, onde `n` é o número de itens na binomial heap.

A operação administrativa básica é a intercalação (`merge()`) de listas ordenadas pelo campo `order`.

O consumo de tempo **amortizado** de `insert()` é $\mathrm{O}(1)$.

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

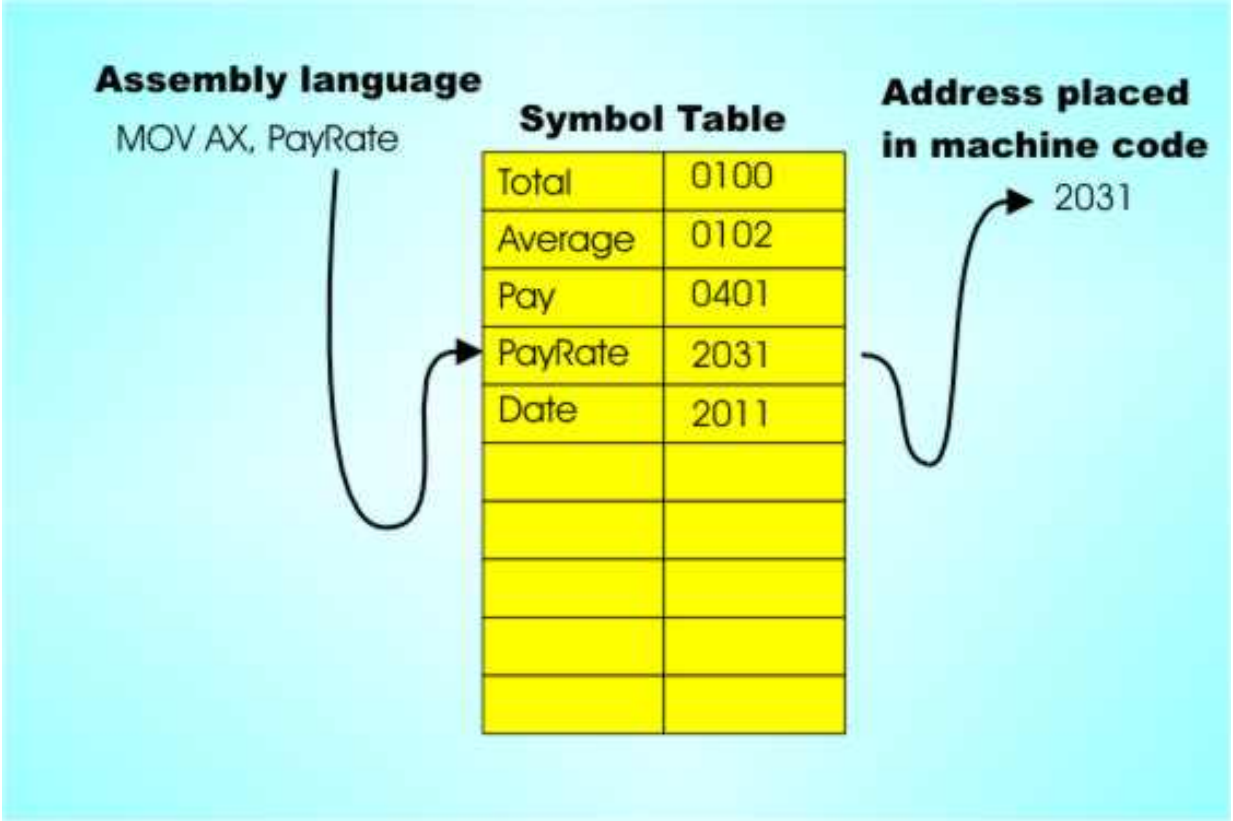
Consumo de tempo MinPQ

	heap	d-heap	Fibonacci heap
`insert()`	$\mathrm{O}(\lg \mathtt{n})$	$\mathrm{O}(\log_d \mathtt{n})$	$\mathrm{O}(1)$
`delMin()`	$\mathrm{O}(\lg \mathtt{n})$	$\mathrm{O}(\log_d \mathtt{n})$	$\mathrm{O}(\lg \mathtt{n})$
`change()`	$\mathrm{O}(\lg \mathtt{n})$	$\mathrm{O}(\log_d \mathtt{n})$	$\mathrm{O}(1)$

Como fazer o `change()`
em `LMinPQ` e `BinomialMinPQ`?

Em Fibonacci heap,
o consumo de `delMin()` é amortizado.

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
dictionaries, *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?

Além destas, usaremos as rotinas de um iterador.

Cliente: `Index()`

`keys` = palavras e
`vals` = lista de posições onde a palavra ocorre

```
int main(int argc, char *argv) {
    int minLength = atoi(argv[1]),
        minOccurrence = atoi(argv[2]), n;
    char **words = readAllStrings(&n), *s;
    /* words[0..n-1] guarda as palavras */
    Queue q;
```

Cliente: Index()

```
stInit();
for (int i = 0; i < n; i++){
    s = words[i];
    if (strlen(s) < minLength) continue;
    if (!stContains(s))
        stPut(s, queueInit());
    q = stGet(s);
    queuePut(q, i);
}
```

Cliente: Index()

```
    stStartIterator();
    while (stHasNext()) {
        s = stNext();
        q = stGet(s);
        if (queueSize(q) >= minOccurrence) {
            printf("%s : ", s);
            queueDump(q);
        }
    }
    stFree();
}
```