Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 5

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**/**mínimo** 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**/**mínimo**.

Isso significa que uma fila priorizada manipula itens comparáveis (Comparable).

API PQ-máximo

```
public class MaxPQ<Item extends
                        Comparable<Item>>
```

`public class`	`MaxPQ`	
	`MaxPQ(int cap)`	cria uma PQ
`void`	`insert(Item v)`	insere item v nesta P
`Item`	`max()`	devolve um máximo
`Item`	`delMax()`	remove e devolve
`boolean`	`isEmpty()`	PQ está vazia?
`int`	`size()`	número de itens

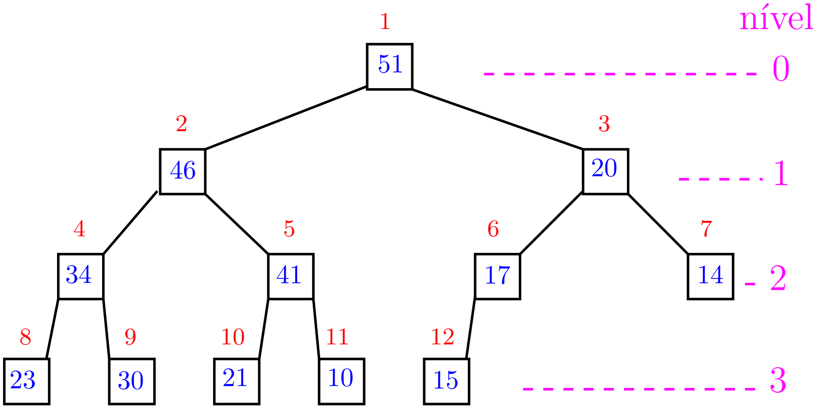
Implementações elementares

Podemos implementar a classe `MaxPQ` ou `MinPQ` com

- vetor de itens não-ordenados;
- vetor de itens ordenados;
- lista ligadas de itens ordenados ou não ordenados.

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

Implementação com *binary heaps*



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

Implementação com *binary heaps*

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

O consumo de tempo das operações envolvendo uma fila priorizada implementada como um binary heap é $O(\lg \mathtt{n})$, 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 $O(\mathtt{n} \lg \mathtt{n})$ (construção se apoia em `swim()`).

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

Hmm

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

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

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

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 esta 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.

API PQ-máximo mutável

```
public class IndexMinPQ<Item extends
                        Comparable<Item>>
```

public class	IndexMinPQ	
	IndexMinPQ(int maxN)	
void	insert(int k, Item item)	insere
void	change(int k, Item item)	muda item
boolean	contains(int k)	k está associado?
void	delete(int k)	remove k e o item
Item	min()	menor item
int	minIndex()	índice do maior ite
int	delMin()	remove maior item retorna seu índice
boolean	isEmpty()	está vazia?
int	size()	número de itens

PQ mutável com binary heaps

Com binary heaps: mantemos tabelas de maneira que seja possível:

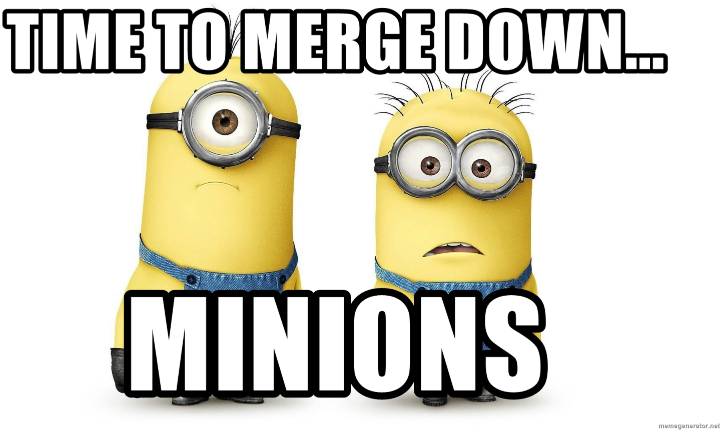
- encontrar um dado `item` e
- trocar itens de posição

em tempo constante.

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

AULA 6

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).

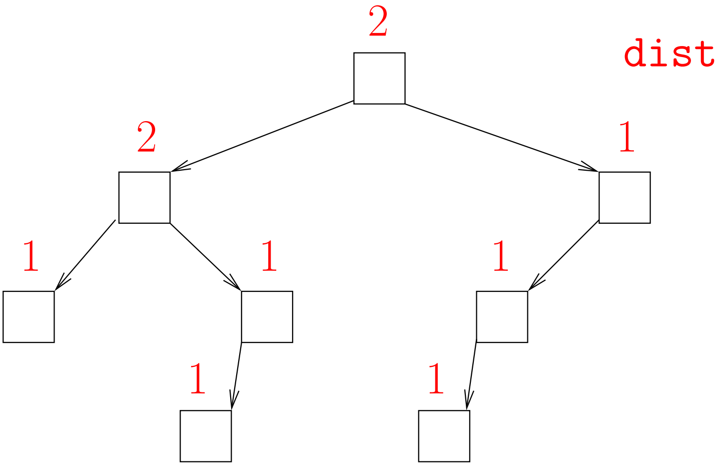
Árvores esquerdistas

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

Exemplo



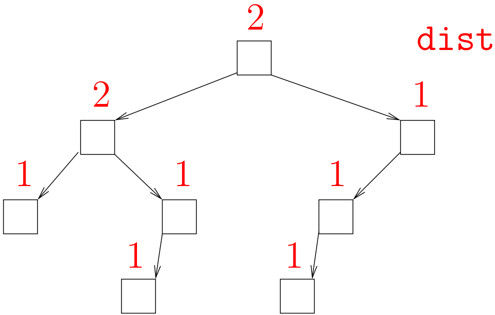
Árvores esquerdistas

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:



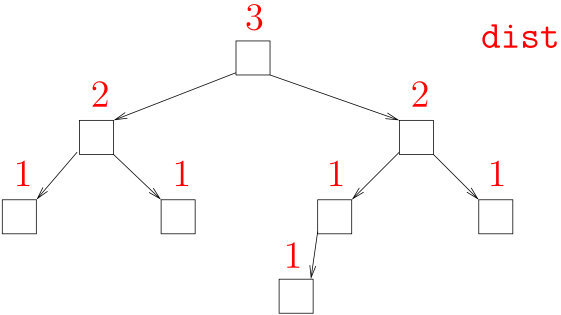
Árvores esquerdistas

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:



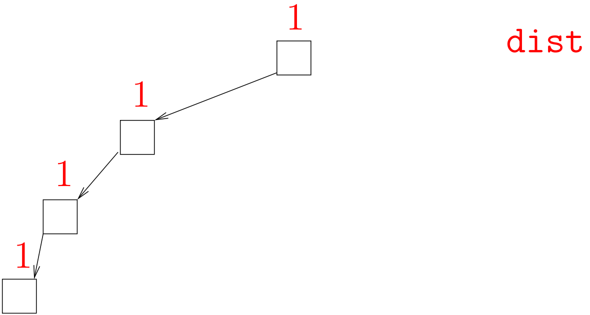
Árvores esquerdistas

Uma árvore é esquerdista se

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

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

Exemplo 3:



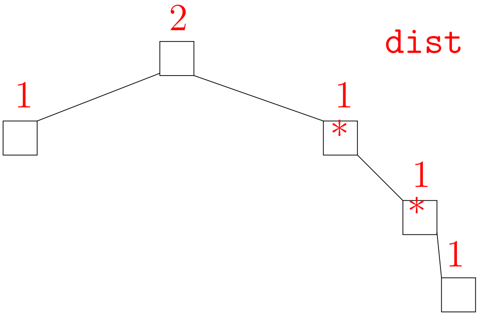
Árvores esquerdistas

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 4: árvore não-esquerdista



Caminho direitista

O **caminho direitista** de um nó $\mathtt{x}$ é a seqüência

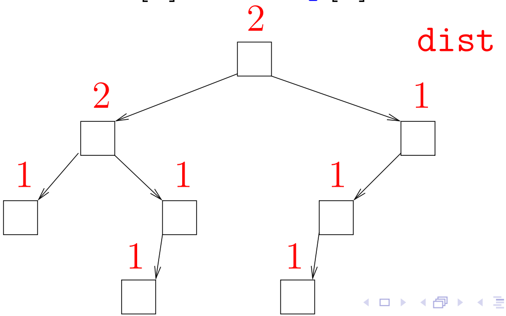
$$\langle \mathtt{x}, \mathtt{dir}[\mathtt{x}], \mathtt{dir}[\mathtt{dir}[\mathtt{x}]], \ldots, \mathtt{null}\rangle.$$

$\mathtt{dcomp}[\mathtt{x}]$:= núm. de nós no caminho direitista de $\mathtt{x}$

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

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

$$\mathtt{dist}[\mathtt{x}] = \mathtt{dcomp}[\mathtt{x}].$$



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

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

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

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

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

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

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

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

Fato estrutural

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

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

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

Prova: (continuação)
Logo,

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

Consequência

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

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

Em particular:

Se $\mathtt{x}$ é raiz de uma árvore esquerdista com $\mathtt{m}$ nós,

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

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

Heap esquerdista

$\mathtt{H}$:= árvore

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

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

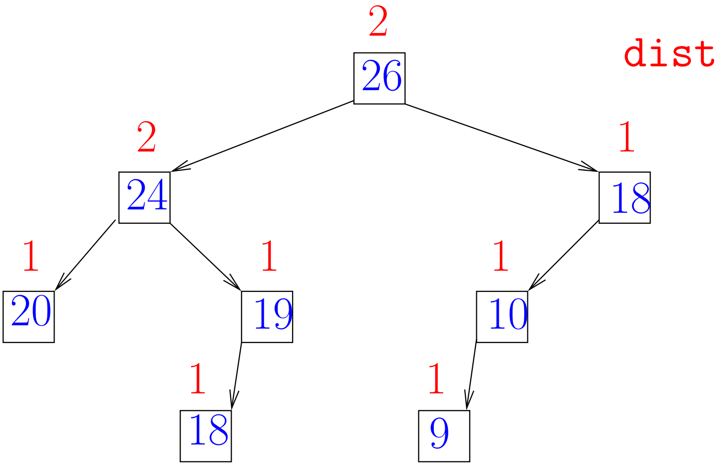
$\mathtt{pai}[\mathtt{x}]$:= pai do nó $\mathtt{x}$

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

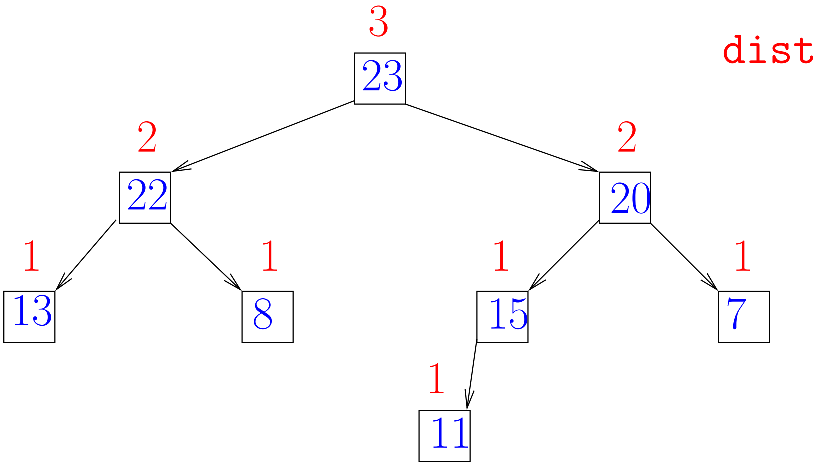
$$\mathtt{prior}[\mathtt{pai}[\mathtt{x}]] \geq \mathtt{prior}[\mathtt{x}]$$

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

Heap esquerdista

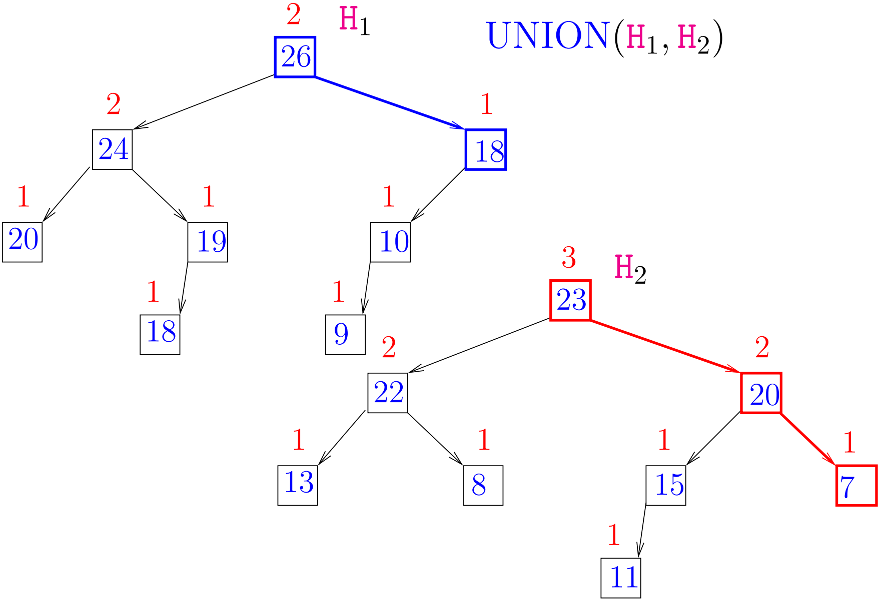


Heap esquerdista

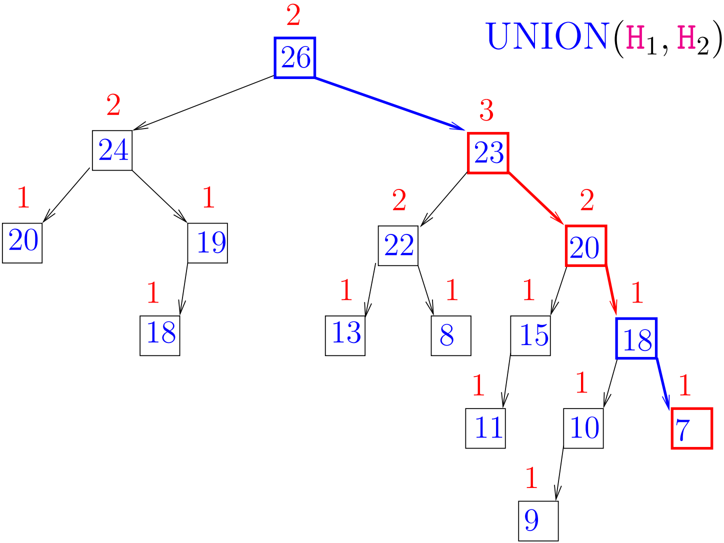


Rotina básica de manipulação

$\mathrm{UNION}(\mathtt{H}_1, \mathtt{H}_2)$

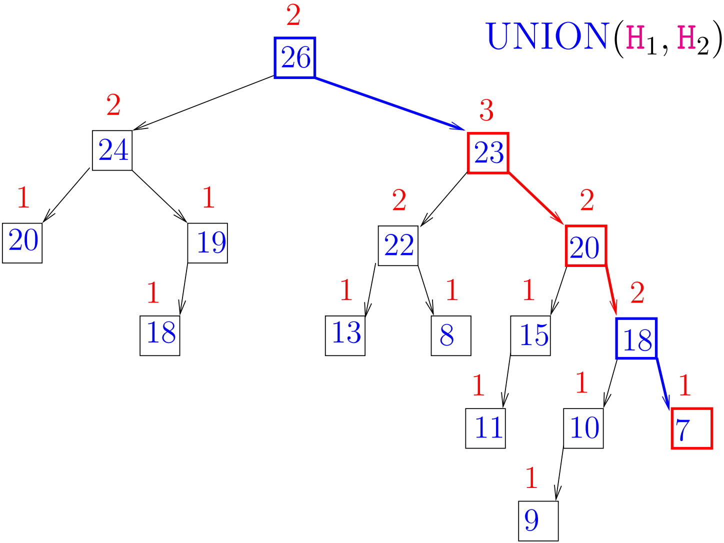


Rotina básica de manipulação



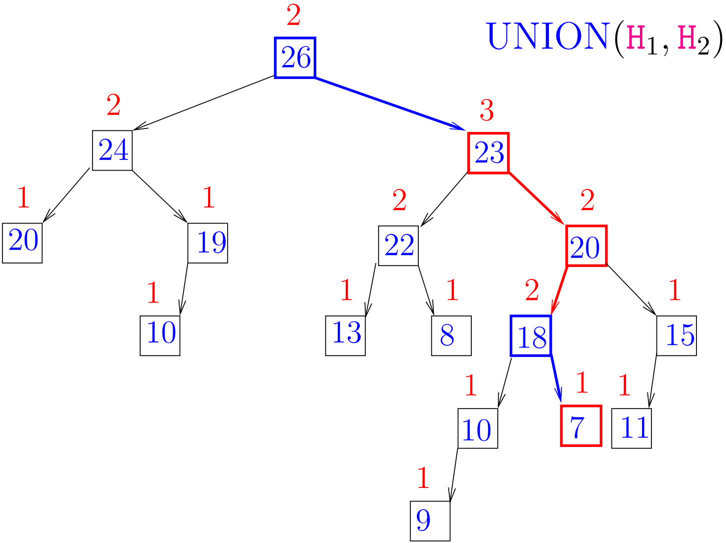
UNION($\texttt{H}_1, \texttt{H}_2$)

Rotina básica de manipulação

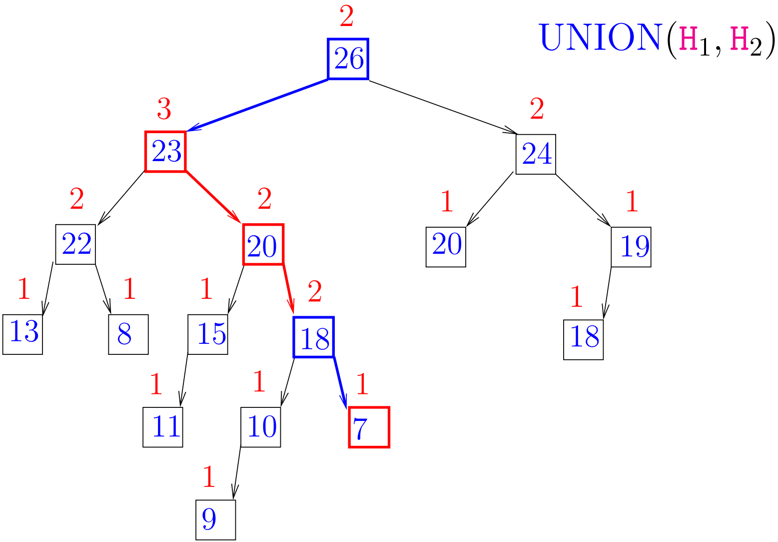


$\mathrm{UNION}(\mathtt{H}_1, \mathtt{H}_2)$

Rotina básica de manipulação



Rotina básica de manipulação



Rotina básica de manipulação

```
LEFTIST-HEAP-UNION (H1, H2)
    se raiz[H1] = null então devolva H2
    se raiz[H2] = null então devolva H1
    se prior[raiz[H1]] < prior[raiz[H2]]
        então H1 ↔ H2
    x1 ← raiz[H1]   x2 ← raiz[H2]
    se esq[x1] = null então esq[x1] ← x2
    senão H′ ← MakeLeftistHeap()
          raiz[H′] ← dir[x1]
          H′ ← LeftistHeapMerge(H′, H2)
          dir[x1] ← raiz[H′]
          se dist[esq[x1]] < dist[dir[x1]]
              então esq[x1] ↔ dir[x2]
          dist[x1] ← dist[dir[x1]] + 1
    devolva H1
```

Consumo de tempo

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

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

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

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

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)
    H' ← MakeLeftistHeap()
    esq[x] ← null
    dir[x] ← null
    dist[x] ← 1
    raiz[H'] ← x
    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 H.

```
LEFTIST-HEAP-EXTRACT (H)
    x ← raiz[H]
    H′ ← MakeLeftistHeap()
    raiz[H′] ← esq[x]
    raiz[H] ← dir[x]
    H ← LeftistHeapMerge(H, H′)
    devolva x
```

Consome tempo $O(\lg m)$.

Cliente `LeftistMaxPQ`: `class BottomM`

No código a seguir,

- `T` é uma abreviatura para `Transaction` e
- `MaxL` é uma abreviatura para `LeftistMaxPQ`.

`Transaction` é uma das classes do algs4.

O programa retorna as `M` transações de menor valor.

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

Cliente LeftistMaxPQ: class BottomM

```
public static void main(String[] args) {
   int M = Integer.parseInt(args[0]);
   MaxL<T> pq =new MaxL<T>();
   while (StdIn.hasNextLine()) {
      pq.insert(new T(StdIn.readLine()));
      if (pq.size() > M)
         pq.delMax();
   }
   Stack<T> stack =new Stack<T>();
   while (!pq.isEmpty())
      stack.push(pq.delMax());
   for (T t :  stack)
      StdOut.println(t);
}
```

subclasse Node

Cada nó de uma árvore esquerdista terá quatro campos:

```
private class Node{
   private Item item;
   private Node left, right;
   private int dist;
   public Node(Item item, Node left,
       Node right, int dist) {
       this.item = item;
       this.left = left;
       this.right = right;
       this.dist = dist;
   }
}
```

Class LeftistMaxPQ: esqueleto

```
public class LeftistMaxPQ<Item extends
                Comparable<Item>> {
   private Node root;
   private int n;
   private class Node{...}
   public LeftistMaxPQ() {...}
   public boolean isEmpty() {...}
   public int size() {...}
   public void insert(Item item) {...}
   public Item delMax() {...}
   public void union(LeftistMaxPQ<Item> t)
   private Node merge(Node r1, Node r2)
   private boolean less(Node r1, Noder2)
}
```

LeftistMaxPQ: construtor, isEmpty() e size()

```
// fila vazia
public LeftistMaxPQ() {
}

public boolean isEmpty() {
   return n == 0;
}

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

LeftistMaxPQ: insert() e delMax()

```
public void insert(Item item) {
   Node s = new Node(item, null, null, 1);
   root = merge(root, s);
   n++;
}
public Item delMax() {
   Item max = root.item;
   root = merge(root.left, root.right);
   n--;
   return max;
}
```

LeftistMaxPQ: union() e less()

```
// atenção:  destrói a MaxPQ that
public
void union(LeftistMaxPQ<Item> that) {
   if (that == null) return;
   this.root = merge(this.root,that.root);
   this.n += that.n;
}

private boolean less(Node r, Node s) {
   return r.item.compareTo(s.item) < 0;
}
```

`LeftistMaxPQ`: `merge()`

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

```
private Node merge(Node r1, Node r2) {
   if (r1 == null) return r2;
   if (r2 == null) return r1;
   if(r1.item.compareTo(r2.item) < 0) {
      Node 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.

LeftistMaxPQ: merge()

```
private Node merge(Node r1, Node r2) {
   if (r1 == null) return r2;
   if (r2 == null) return r1;
   // r1 != null e r2 != null
   if (less(r1, r2)) {
      Node tmp = r1; r1 = r2; r2 = tmp;
   }
   // r1 aponta para o maior item
   if (r1.left == null) r1.left = r2;
   else {
```

LeftistMaxPQ: merge()

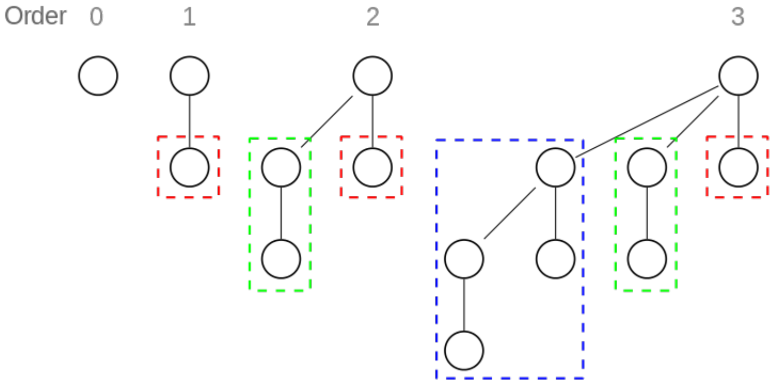
```
  else {
     r1.right = merge(r1.right, r2);
     if (r1.left.dist < r1.right.dist) {
        Node t = r1.left;
        r1.left = r1.right;
        r1.right = t;
     }
     r1.dist = r1.right.dist + 1;
  }
  return r1;
}
```

Conclusão

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

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

Binomial heaps



Fontes: Wikipedia,

Foundations of Data Science

CLRS 19

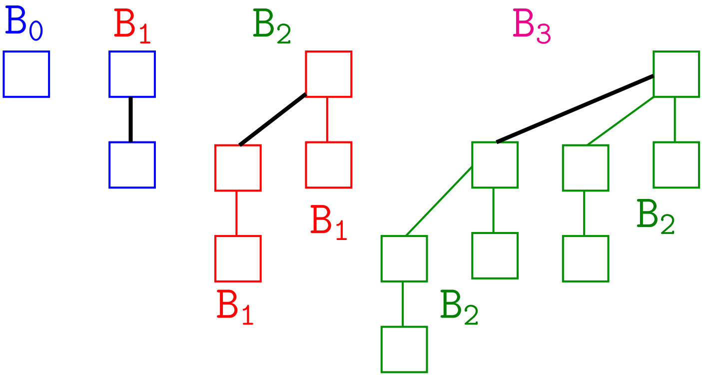
Binomial trees

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

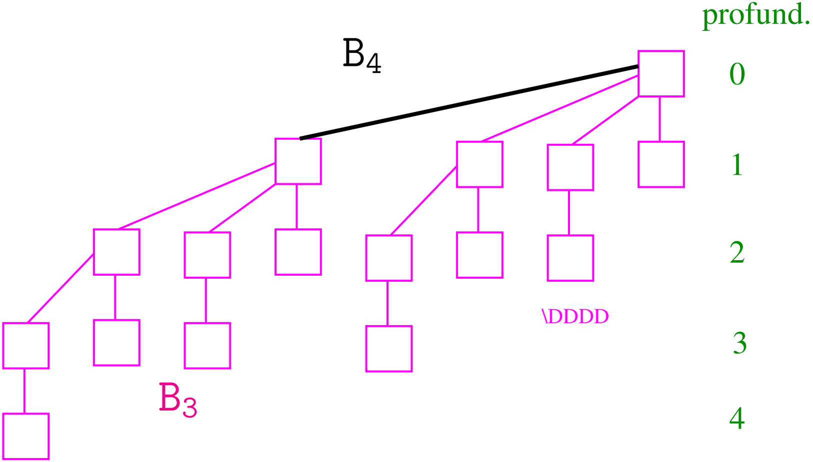
Árvores binomiais são definadas recursivamente:

- um nó é a árvore binomial $\mathtt{B}_0$ de ordem 0.
- para $\mathtt{k} = 1, 2, \ldots$, a árvore binomial $\mathtt{B}_\mathtt{k}$ de ordem $\mathtt{k}$ consiste de duas árvores $\mathtt{B}_{\mathtt{k}-1}$ 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** $\mathbf{k}$:

- 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 $\mathtt{n}$ nós é $\lg \mathtt{n}$.

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

Binomial heap

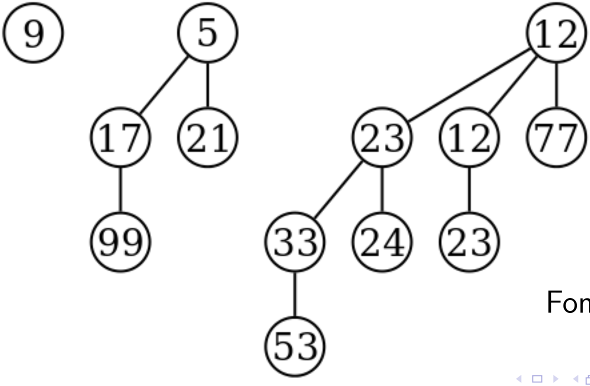
Uma **binomial heap** `H` é uma coleção de binomial trees que satisfaz as seguintes propriedades:

- 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`:

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

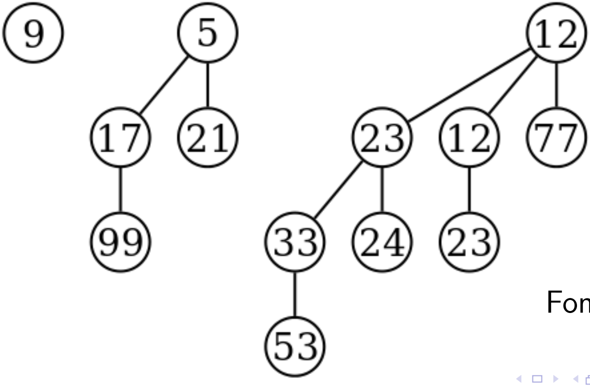


Fonte: Wikipedia

Binomial heap

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

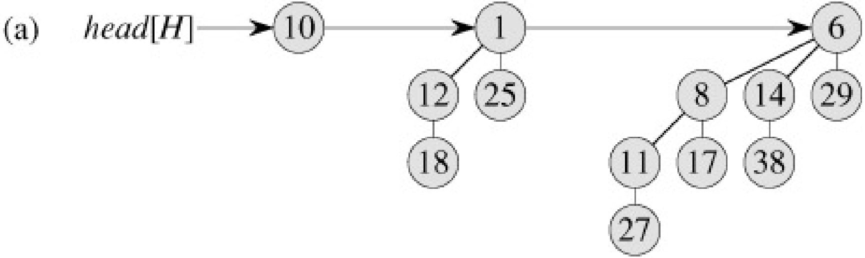
$$\mathtt{n} = 13 = 1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3.$$



Fonte: Wikipedia

Binomial heap: estrutura de dados

Fonte: CLRS

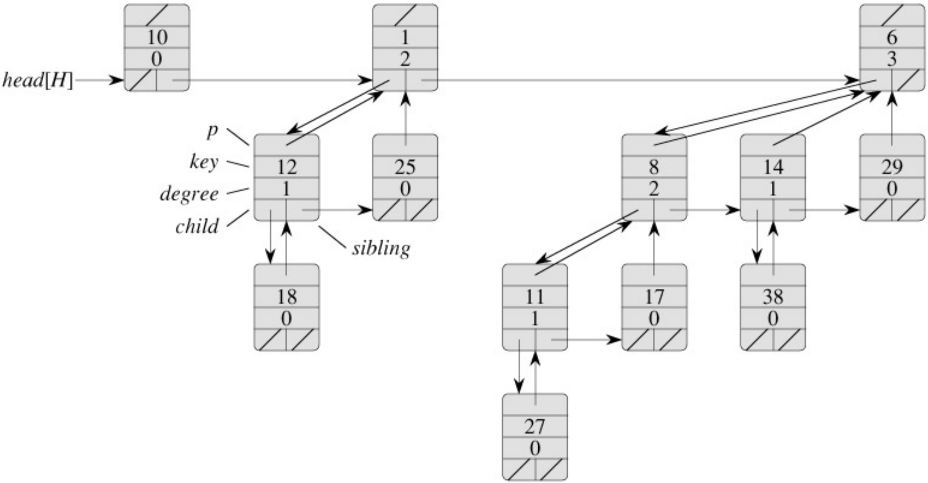


BinomialMinPQ: Node

Representação de um Node de uma binomial tree.

```
private class Node {
   Item item; // ou key
   int order; // ou grau, degree
   Node child; // filho mais a esquerda
   Node sibling; // lista de irmãos
   Node parent; // pai
}
```

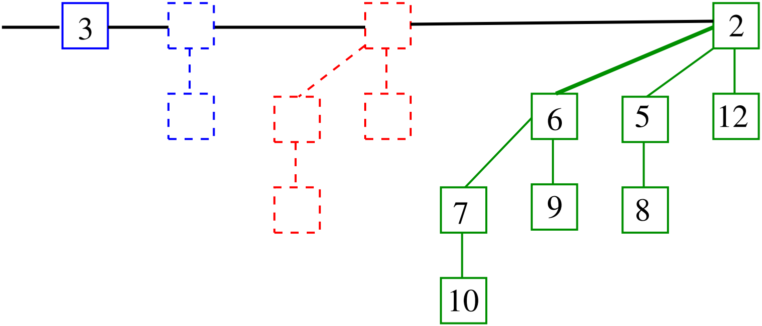
A lista de irmão está em ordenada de acordo com o grau dos nós (ordem das árvores).



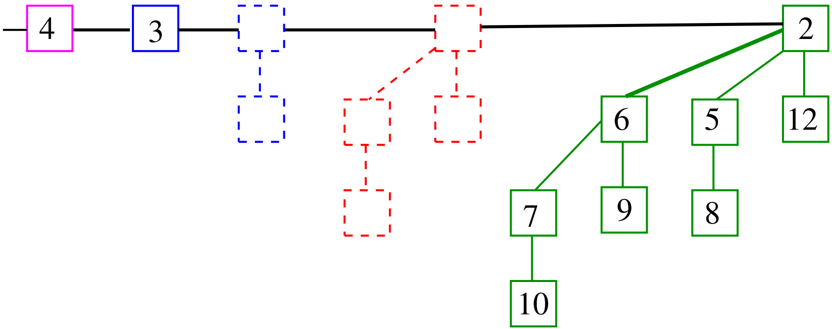
head[H]
10
0
1
2
6
3
p
key
degree
child
sibling
12
1
25
0
8
2
14
1
29
0
18
0
11
1
17
0
38
0
27
0

Fonte: CLRS

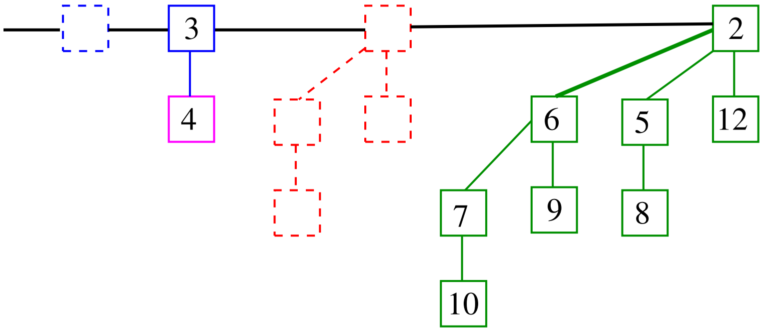
BinomialMinPQ: insert()



BinomialMinPQ: insert()



BinomialMinPQ: `insert()`

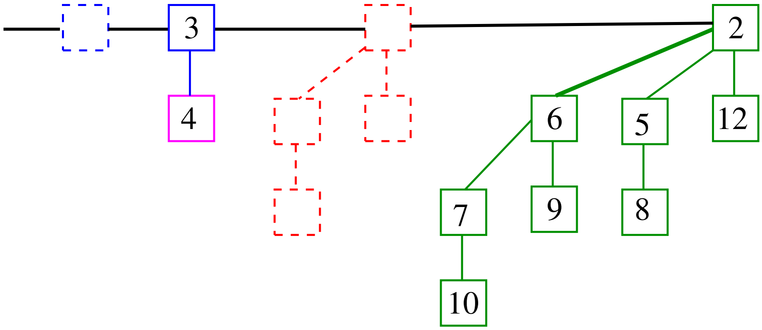


BinomialMinPQ: insert()

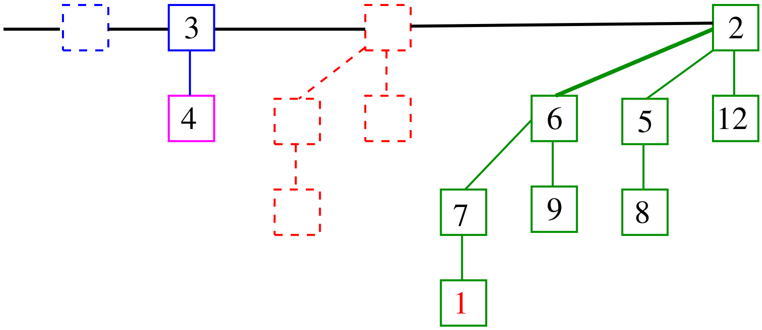
Como existem $1 + \lfloor \lg n \rfloor$ binomial trees que podem ser unidas concluímos o seguinte.

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

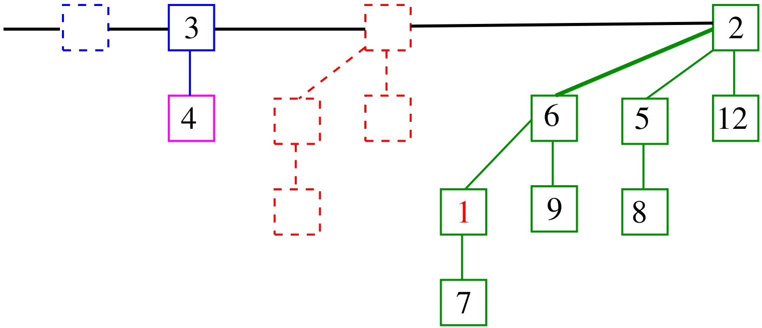
BinomialMinPQ: change()



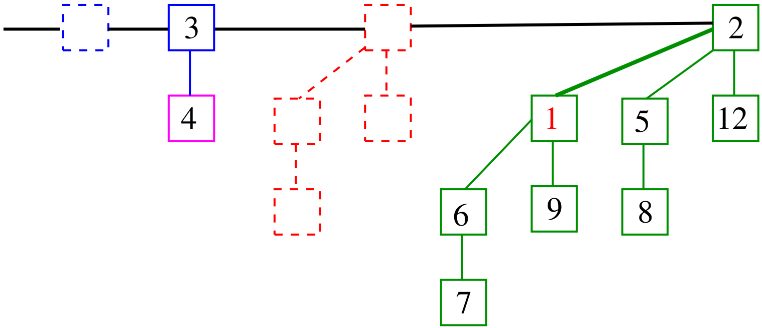
BinomialMinPQ: change()



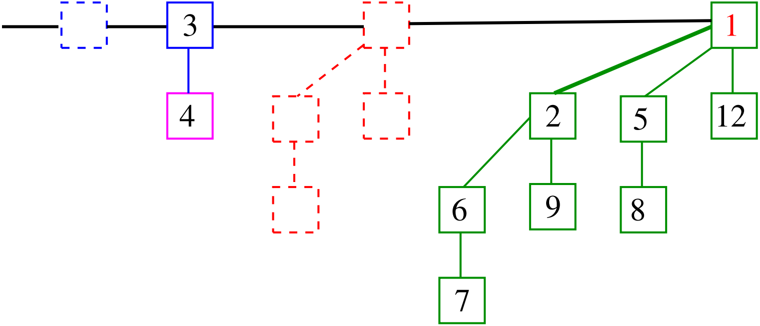
BinomialMinPQ: change()



BinomialMinPQ: change()



BinomialMinPQ: change()



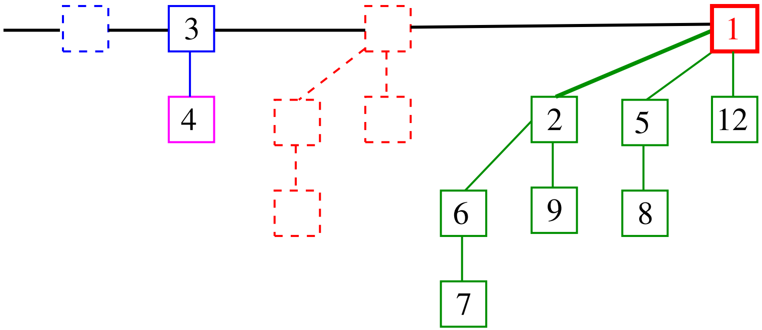
`BinomialMinPQ`: `change()`

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

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

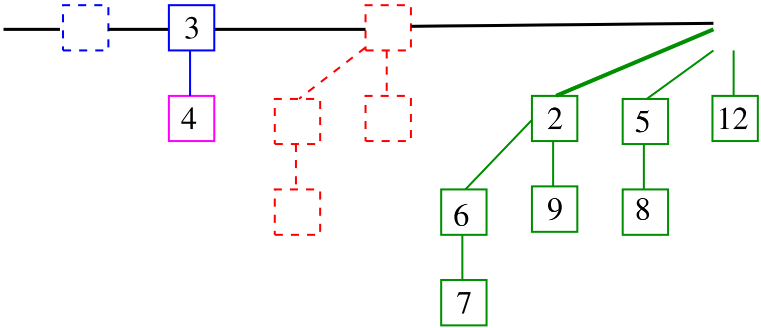
BinomialMinPQ: delMin()

Percorra a lista das raizes e encontre o menor item.



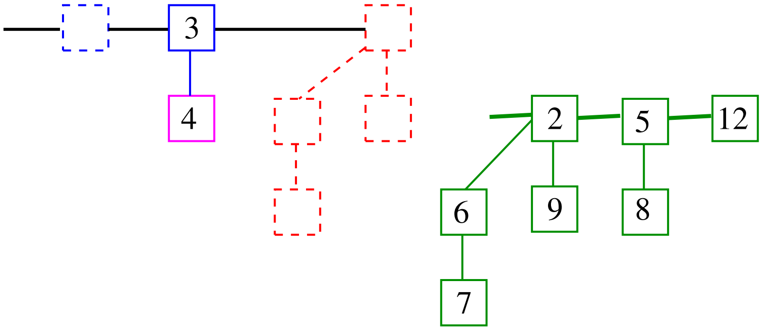
BinomialMinPQ: delMin()

Remova o nó.



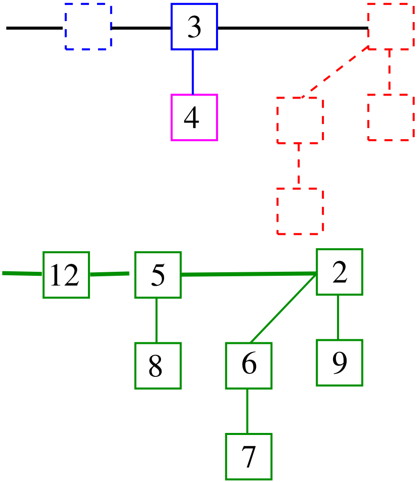
BinomialMinPQ: delMin()

Inverta a lista dos seus filhos.



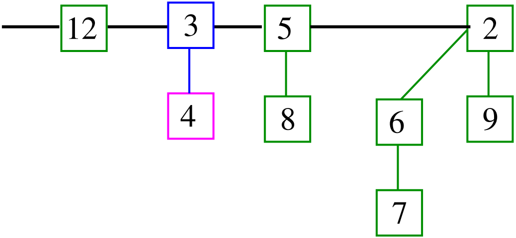
BinomialMinPQ: delMin()

Inverta a lista dos seus filhos.



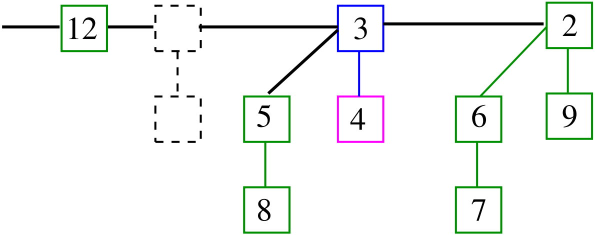
BinomialMinPQ: delMin()

Faça merge() das duas binomial heaps.



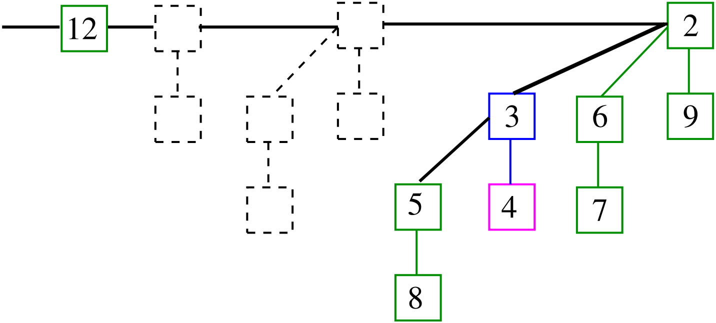
`BinomialMinPQ`: `delMin()`

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 $O(\lg n)$.

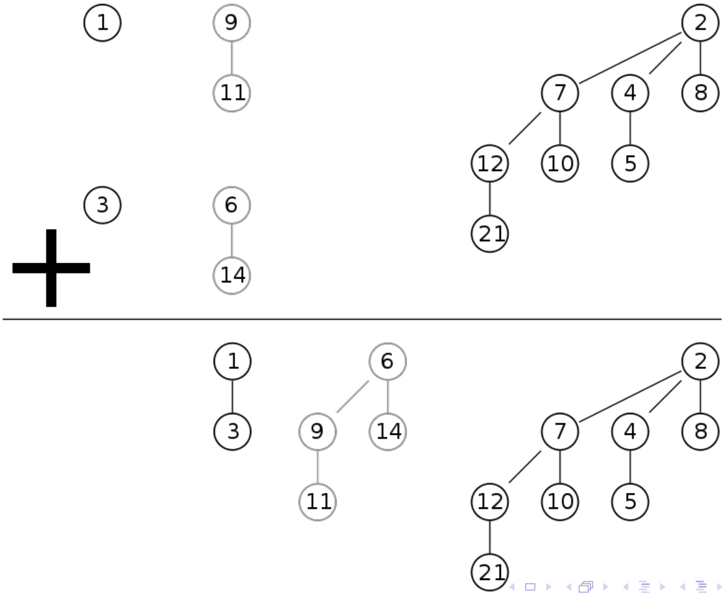
O consumo de tempo para inverter e intercalar listas com até $1 + \lfloor \lg n \rfloor$ itens é $O(\lg n)$.

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

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

BinomialMinPQ: merge()

Fonte: Wikipedia



subclasse Node

Cada nó de uma árvore binomial terá quatro campos (deixaremos parent de fora):

```
private class Node{
   private Item item;
   private Node child, sibling;
   private int order;
   public Node(Item item, Node child,
       Node sibling, int order) {
       this.item = item;
       this.child = child;
       this.sibling = sibling;
       this.order = order;
   }
}
```

Class BinomialMinPQ: esqueleto

```
public class BinomialMinPQ<Item extends
                Comparable<Item>> {
   private Node head;
   private class Node{...}
   public BinomialMinPQ() {...}
   public boolean isEmpty() {...}
   public int size() {...}
   public void insert(Item item) {...}
   public Item delMin() {...}
   public void union(BinomialMinPQ<Item>t)
   private Node merge(Node r1, Node r2)
   private boolean greater(Node r1,
                           Node r2)
}
```

BinomialMinPQ: construtor, isEmpty() e size()

```
// fila vazia
public BinomialMinPQ() {
}

public boolean isEmpty() {
   return head == null;
}

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

BinomialMinPQ: insert()

```
public void insert(Item item) {
   Node x = new Node(item, null, null, 0);
   head = merge(head, x);
   n++;
}
```

BinomialMinPQ: delMin()

```
public Item delMin() {
   // remova minimo do heap
   Node min = eraseMin();
   n--;
   if (min.child == null)
       return min.item;
   Node x= min.child;
   // inverta a lista de filhos
   min.child = null;
   Node 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)
  return min.item;
}
```

BinomialMinPQ: union()

```
// atenção:  destrói a MinPQ that
public
void union(BinomialMinPQ<Item> that) {
   if (that == null) return;
   this.head = merge(this.head,that.head);
   this.n += that.n;
}
```

BinomialMinPQ: greater() e link()

Supõe greater(r1, r2), r2 é transformada em raiz.

```
private boolean greater(Node r, Node s) {
   return r.item.compareTo(s.item) > 0;
}

private void link(Noder1, Noder2) {
   r1.sibling = r2.child;
   r2.child = r1;
   r2.order++;
}
```

BinomialMinPQ: eraseMin()

```
private Node eraseMin() {
   Node min = head, prev = null;
   Node current = head;
   while (current.sibling != null) {
       if (greater(min.item,
                   current.sibling.item)) {
          prev = current;
          min = current.sibling;
       }
       current = current.sibling;
   }
   if (min == head) head = min.sibling;
   else prev.sibling = min.sibling;
   return min; }
```

BinomialMinPQ: mergeR()

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

```
private Node mergeR(Node r1, Node r2) {
   if (r1 == null) return r2;
   if (r2 == null) return r1;
   if(r1.order < r2.order) < 0) {
      Node t = r1; r1 = r2; r2 = t;
   }
   r1.sibling = mergeR(r1.sibling, r2);
   return r1;
}
```

O *essencialmente* é devido ao fato de ser necessário juntarmos as árvores repetidas.

BinomialMinPQ: merge()

```
private Node merge(Node r1, Node r2) {
   if (r1 == null) return r2;
   if (r2 == null) return r1;
   Node r = mergeR(r1, r2);
   Node x = r;
   Node prevx = null, nextx = x.sibling;
   while (nextx!= null) {
       if (x.order < nextx.order
       || (nextx.sibling!= null
       && nextx.sibling.order==x.order)) {
           prevx = x; x = nextx;
       } else
```

BinomialMinPQ: merge()

```
    } else
    if (greater(nextx.item, x.item)) {
        x.sibling = nextx.sibling;
        link(nextx, x);
    } else {
        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 $\mathrm{O}(1)$.

Consumo de tempo MinPQ

	heap	d-heap	fibonacci heap
insert()	$O(\lg n)$	$O(\log_d n)$	$O(1)$
delMin()	$O(\lg n)$	$O(\log_d n)$	$O(\lg n)$
change()	$O(\lg n)$	$O(\log_d n)$	$O(1)$

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