Filas de Prioridade

Heap Binomial

Introdução

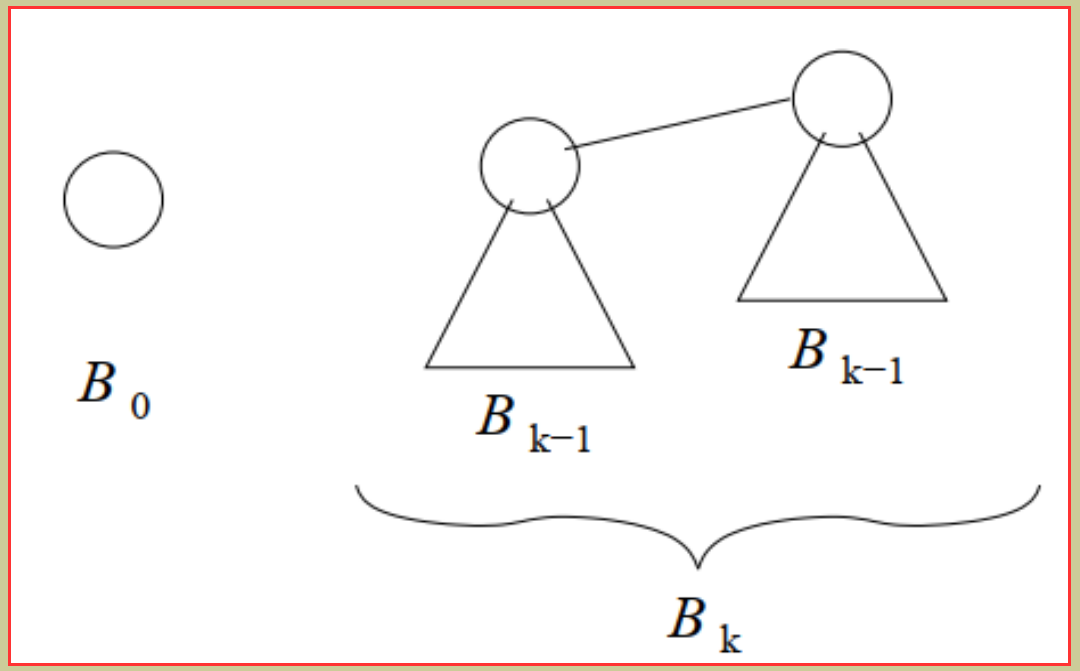
Um Heap Binomial

- É visto como uma coleção de Árvores Binomiais
- Pode ser implementado com uma lista ligada de Árvores Binomiais

Árvore Binomial

Definição (recursiva) de uma Árvore Binomial

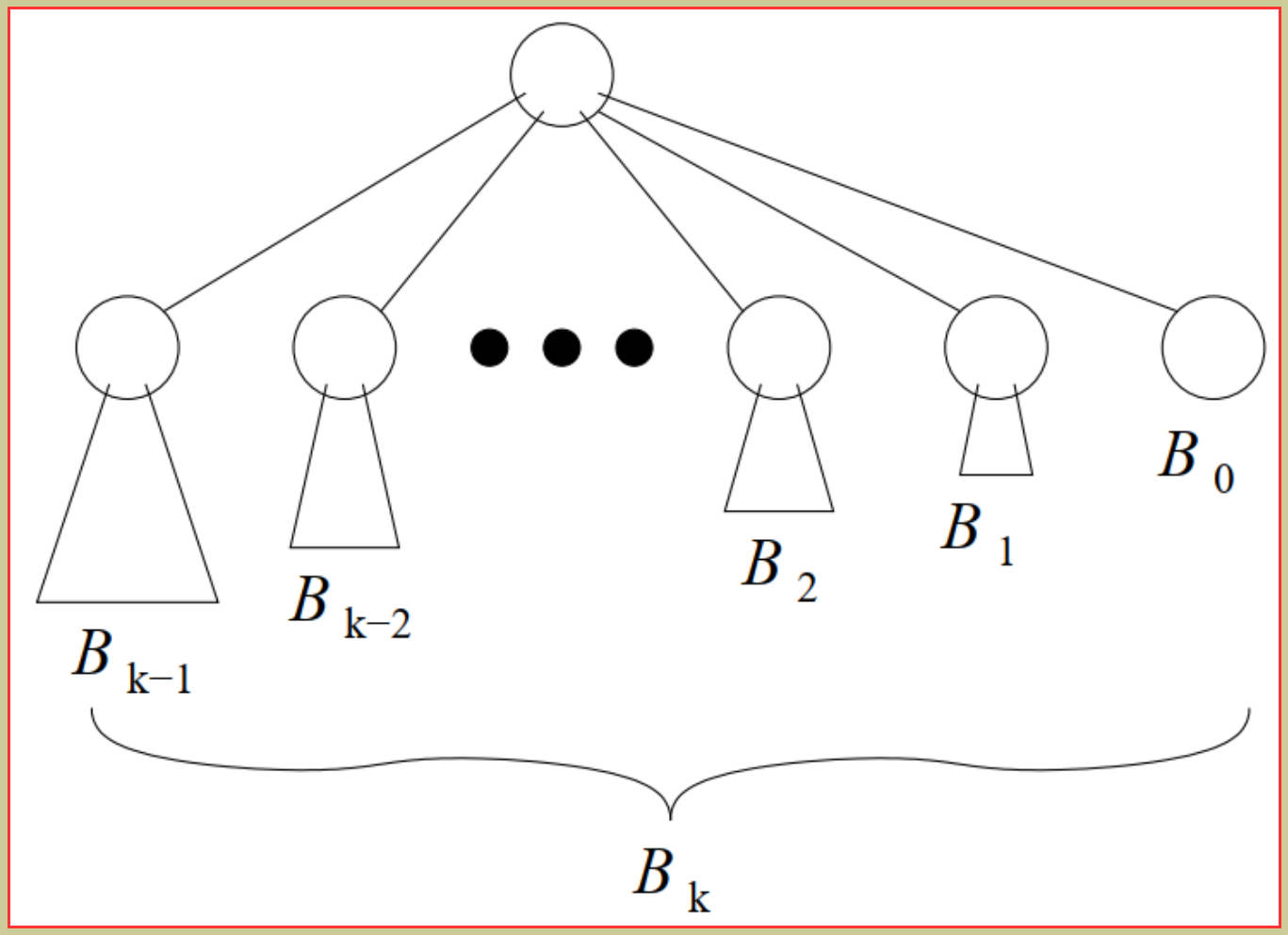
- Uma Árvore Binomial de ordem 0 (B_0) tem 1 nodo
- Uma Árvore Binomial de ordem k (B_k) pode ser construída
  - a partir de 2 árvores binomiais de ordem k-1, e
  - tornando uma como a filha mais à esquerda da outra



Propriedades

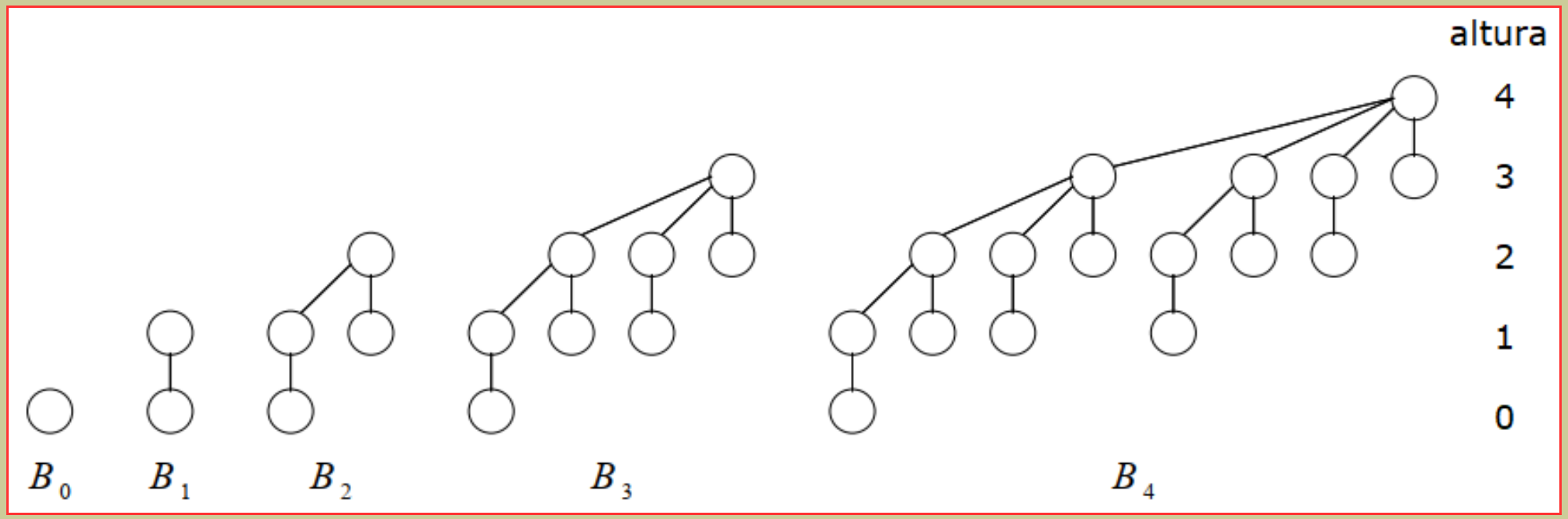
- Uma Árvore Binomial de ordem k, B_k, tem as seguintes propriedades:
  - tem exatamente 2^k nodos
  - tem altura (profundidade) k
  - tem exatamente ${}^kC_i$ nodos em cada nível i (i = 0, 1, …, k)
  - a raiz tem grau k
  - os filhos da raiz são Árvores Binomiais de ordem k-1 (esquerda), k-2, …, 0 (direita)
  - o grau máximo de qualquer nó de uma Árvore Binomial com n nós é $\log_2 n$.

Representação genérica



- a raiz tem grau k
- os filhos da raiz são Árvores Binomiais de ordem k-1 (esquerda), k-2, …, 0 (direita)

Exemplos



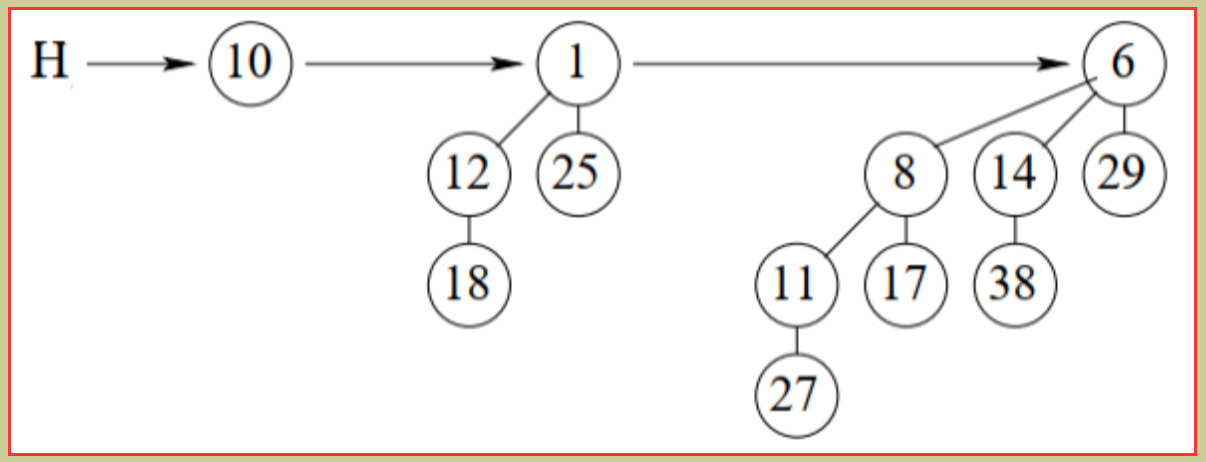
Heap Binomial

Definição

- Um **heap Binomial H** com estrutura **minHeap** é uma coleção de árvores binomiais, que satisfaz as seguintes propriedades:
  - toda a árvore binomial de H tem estrutura de minHeap,
    - a chave de um qualquer elemento é maior ou igual à chave do seu pai,
    - a raiz contém o elemento com a menor chave da árvore
  - para cada inteiro k existe no máximo uma Árvore Binomial de grau k, B_k, em H
    - para um Heap com N nodos existe, no máximo, [log n] + 1 árvores binomiais

Exemplo

- Exemplo de um Heap com as árvores B_0, B_2 e B_3, com 13 nodos ($2^0 + 2^2 + 2^3$)



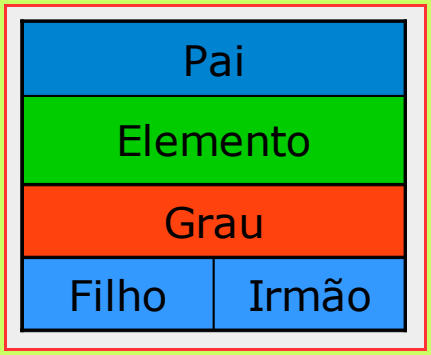
Implementação

- Um heap Binomial é representado computacionalmente com uma lista (ligada) de árvores binomiais
- Cada nodo contém a seguinte informação (dados):
  - um ponteiro para o nodo **pai**
  - a informação do elemento a guardar (**chave**)
  - **grau** do nodo (número de filhos)
  - um ponteiro para o seu **filho mais à esquerda**
  - um ponteiro para o seu **irmão direito** (lista ligada de irmãos)

Implementação

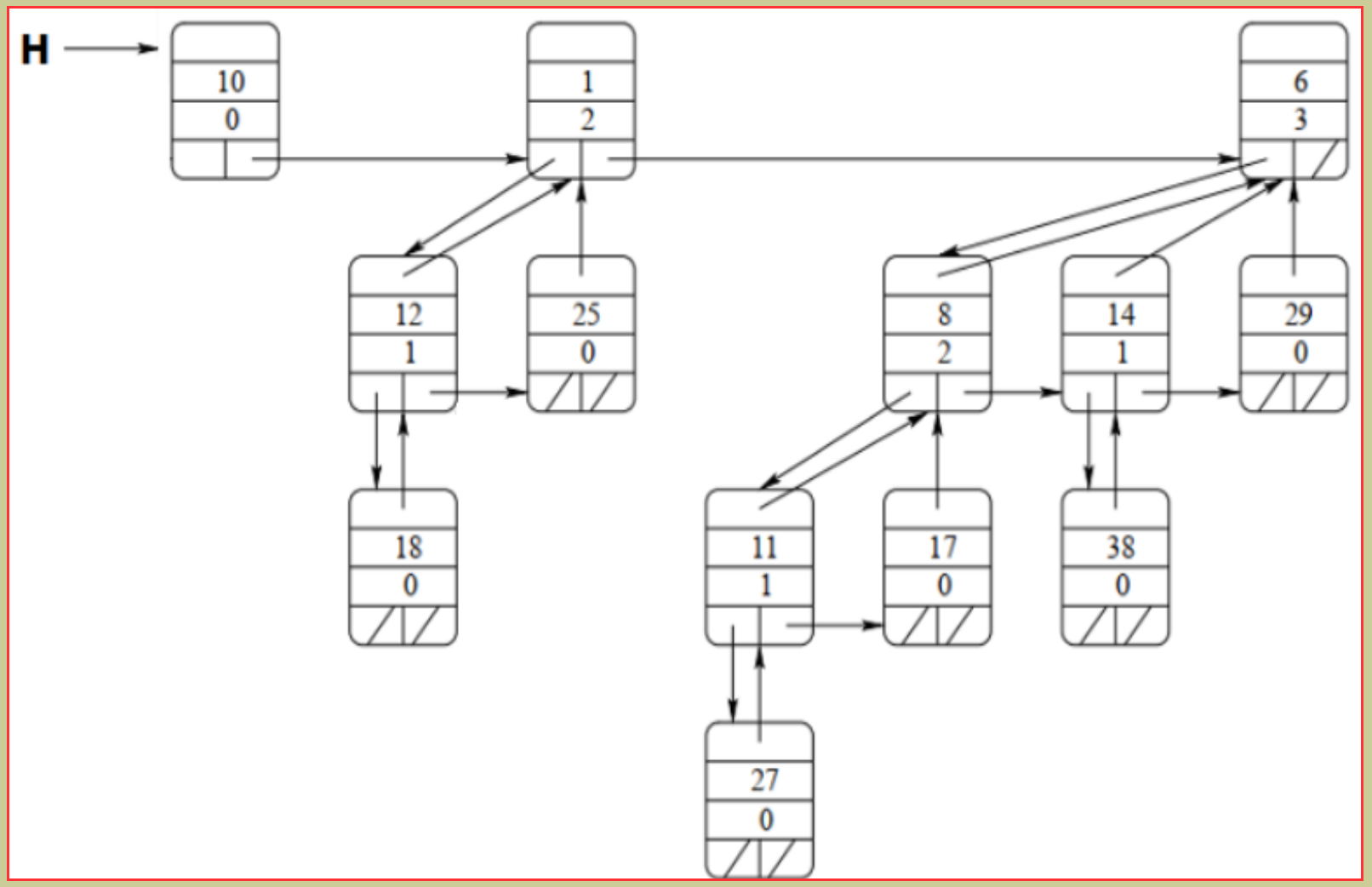
- Exemplo

```
struct NodoHB {
    struct NodoHB *Pai;
    INFOHB Elemento;
    int  Grau;
    struct NodoHB *Filho;      // filho mais à esquerda
    struct NodoBB *Irmao;      // irmão direito
};
typedef struct NodoHB *PNodoHB;
```



Implementação

- Representação gráfica:



Operação auxiliar criar()

- Criar um novo Heap H
  - alocar memória e devolver uma estrutura H tal que H = NULL
  - ordem de complexidade: O(1)

```
algoritmo criar()
  H ← NULL
  devolver H
fim_algoritmo
```

Operação auxiliar vazio(H)

- Verificar se um Heap H está vazio
  - devolver 1 se H é vazio e 0 caso contrário
  - ordem de complexidade: O(1)

```
algoritmo vazio(H)
  se (H = NULL) então
     devolver 1
  senão
     devolver 0
  fim_se
fim_algoritmo
```

Operação principal consultar(H)

- Consultar um Heap: determina o nodo cujo elemento tem a menor chave,
  - basta percorrer as raizes das árvores de H e determinar o menor valor das chaves; devolve um ponteiro para o nodo com aquele elemento
  - ordem de complexidade (pior caso): O(log n)
    - o número máximo de raizes é [log n] + 1, em que [x] = parte inteira do real x

Operação principal consultar(H)

- Consultar um Heap (algoritmo)

```
algoritmo consultar(H)
  P ← NULL
  Q ← H
  min ← ∞
  enquanto (Q ≠ NULL) fazer
     se (chave(Q) < min) então
        min ← chave(Q)
        P ← Q
     fim_se
     Q ← irmão(Q)
  fim_enquanto
  devolver P
fim_algoritmo
```

Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - o processo consiste no seguinte:
    - criar um novo Heap com um único nodo com o elemento x, H1
    - aplicar a operação União de Heaps entre H e H1
  - ordem de complexidade (pior caso): O(log n)
    - ordem de complexidade de criar um Heap: O(1)
    - ordem de complexidade da operação de união de duas Heaps: O(log n)

Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H

```
algoritmo inserir(x, H)
   H1 ← criar()
   pai(x) ← NULL
   filho(x) ← NULL
   irmão(x) ← NULL
   grau(x) ← 0
   H1 ← x
   H ← uniao(H, H1)
fim_algoritmo
```

Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - União de dois Heaps (operação auxiliar)
    - ordem de complexidade (pior caso): O(log n)
    - utilizar uma operação auxiliar para juntar duas Árvores B_{k-1} (T1 e T2)
      - a raiz de T2 será também raiz da nova Árvore B_k
    - utilizar uma operação auxiliar para juntar dois Heaps ordenados por ordem crescente do grau das raizes

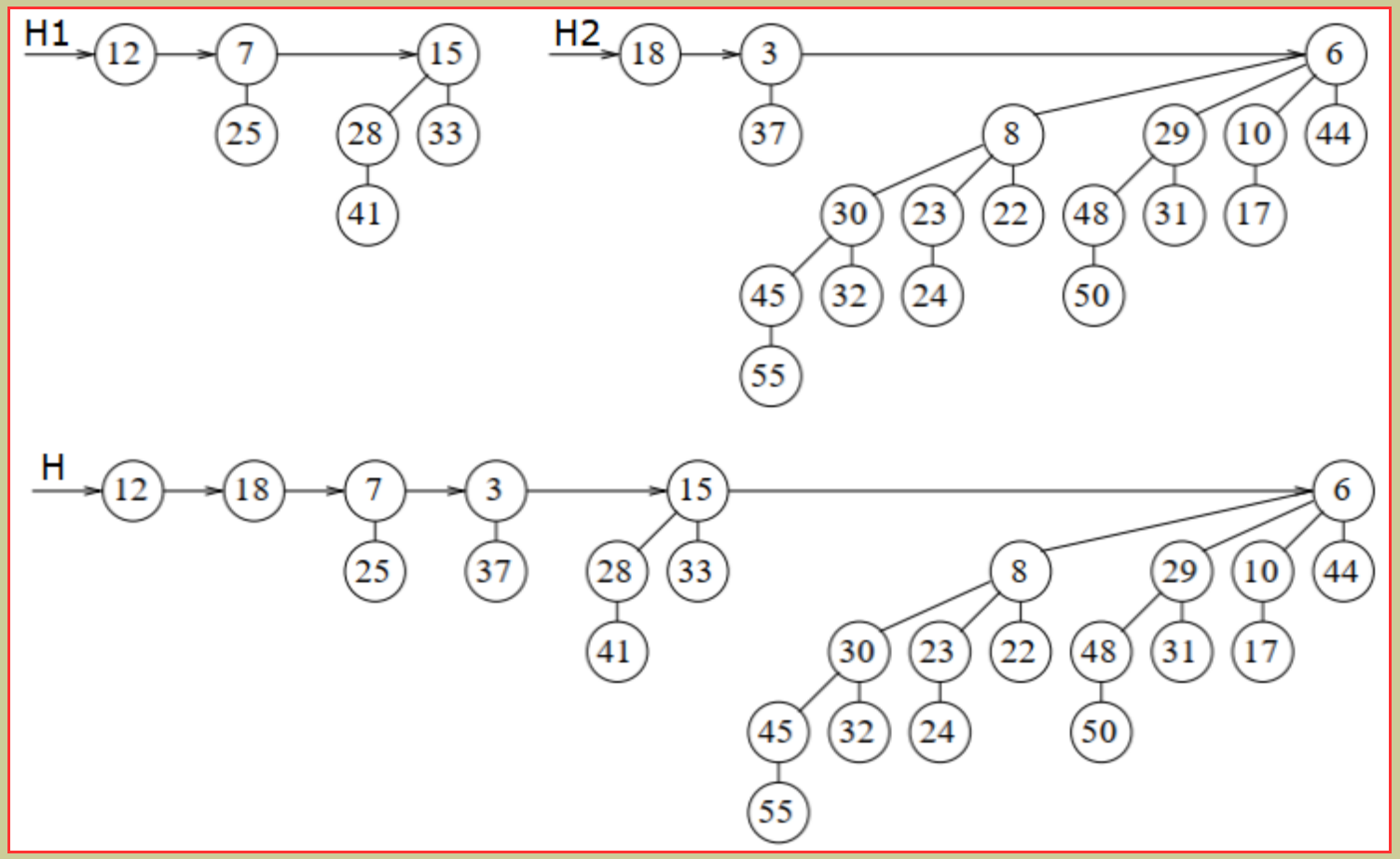
Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - operação para juntar duas Árvores B_{k-1} (T1 e T2) (operação auxiliar)

```
algoritmo juntarAB(T1, T2)
    pai(T1) ← T2
    irmão(T1) ← filho(T2)
    filho(T2) ← T1
    grau(T2) ← grau(T2) + 1
fim_algoritmo
```

Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - operação para juntar dois Heaps ordenados por ordem crescente do grau das raizes



Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - operação para juntar dois Heaps ordenados por ordem crescente do grau das raizes

```
algoritmo juntar(H1, H2)
  H = criar()
  // determinar início da lista
  se (grau(H1) ≤ grau(H2)) então
     H ← H1
     H1 ← irmão(H1)
  senão
     H ← H2
     H2 ← irmão(H2)
  fim_se
  P ← H
  // determinar restantes elementos
...
```

Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H

```
  enquanto (H1 ≠ NULL e H2 ≠ NULL) fazer
     se (grau(H1) ≤ grau(H2)) então
        irmão(P) ← H1
        H1 ← irmão(H1)
     senão
        irmão(P) ← H2
        H2 ← irmão(H2)
     fim_se
     P ← irmão(P)
  fim_enquanto
  se (H1 = NULL) então
     irmão(P) ← H2
  senão
     irmão(P) ← H1
  fim_se
  devolver H
fim_algoritmo
```

Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - União de dois Heaps (operação auxiliar)
    - caso 1:

      não há Árvores do mesmo grau consecutivas
    - caso 2:

      há 3 Árvores do mesmo grau consecutivas, formadas após a união de duas Árvores

      exemplo: cada um dos Heaps originais tinha uma B_1 e uma B_2;

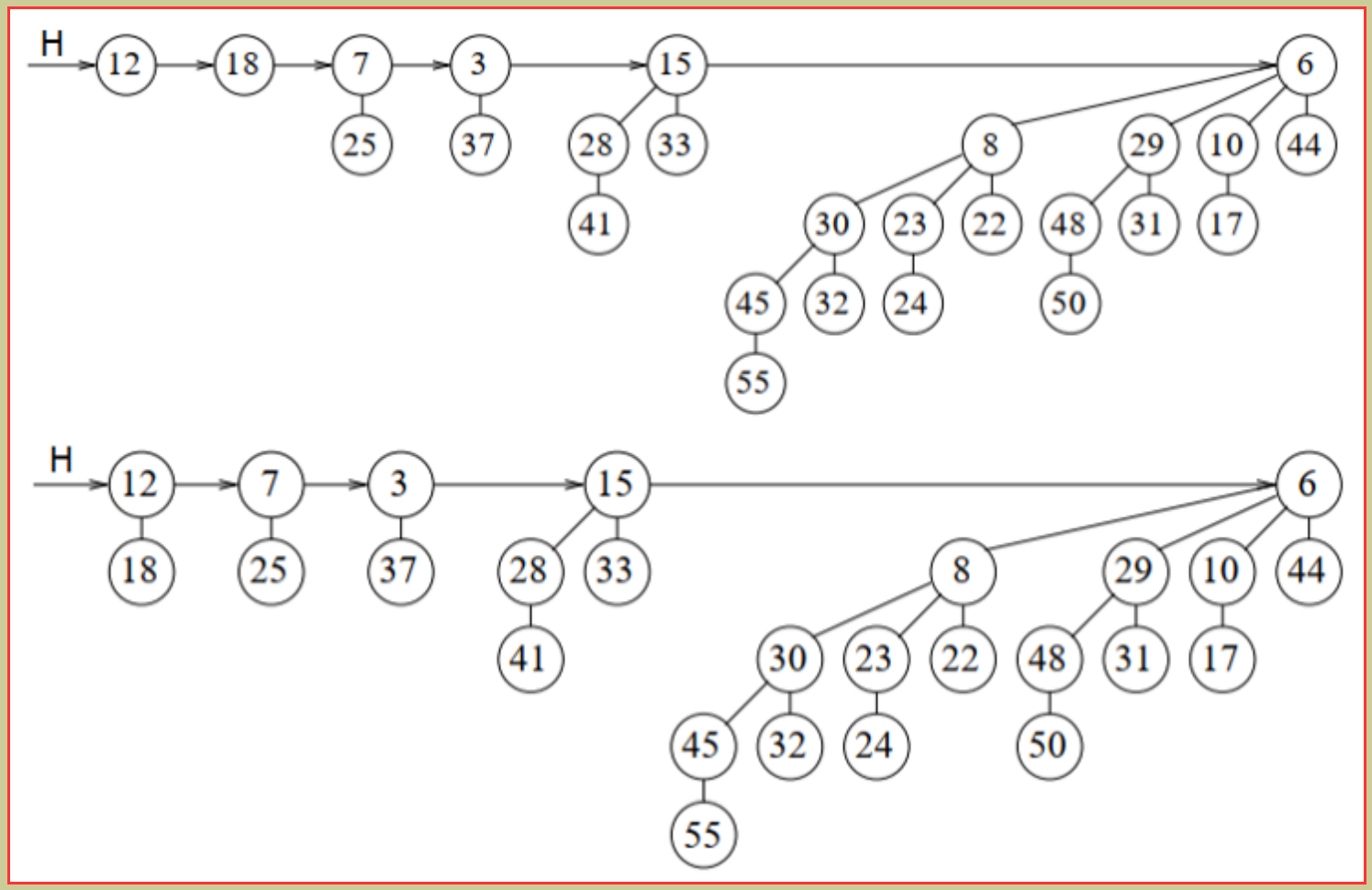
      ao unir-se as Árvores B_1, ficaram com três B_2 (e sem nenhuma B_1)
    - caso 3:

      as duas Árvores B_{k-1} são unidas para formar uma B_k, sendo que a árvore que tem a raiz com menor chave aparece **primeira** na lista
    - caso 4:

      as duas Árvores B_{k-1} são unidas para formar uma B_k, sendo que a árvore que tem a raiz com menor chave aparece **depois** na lista

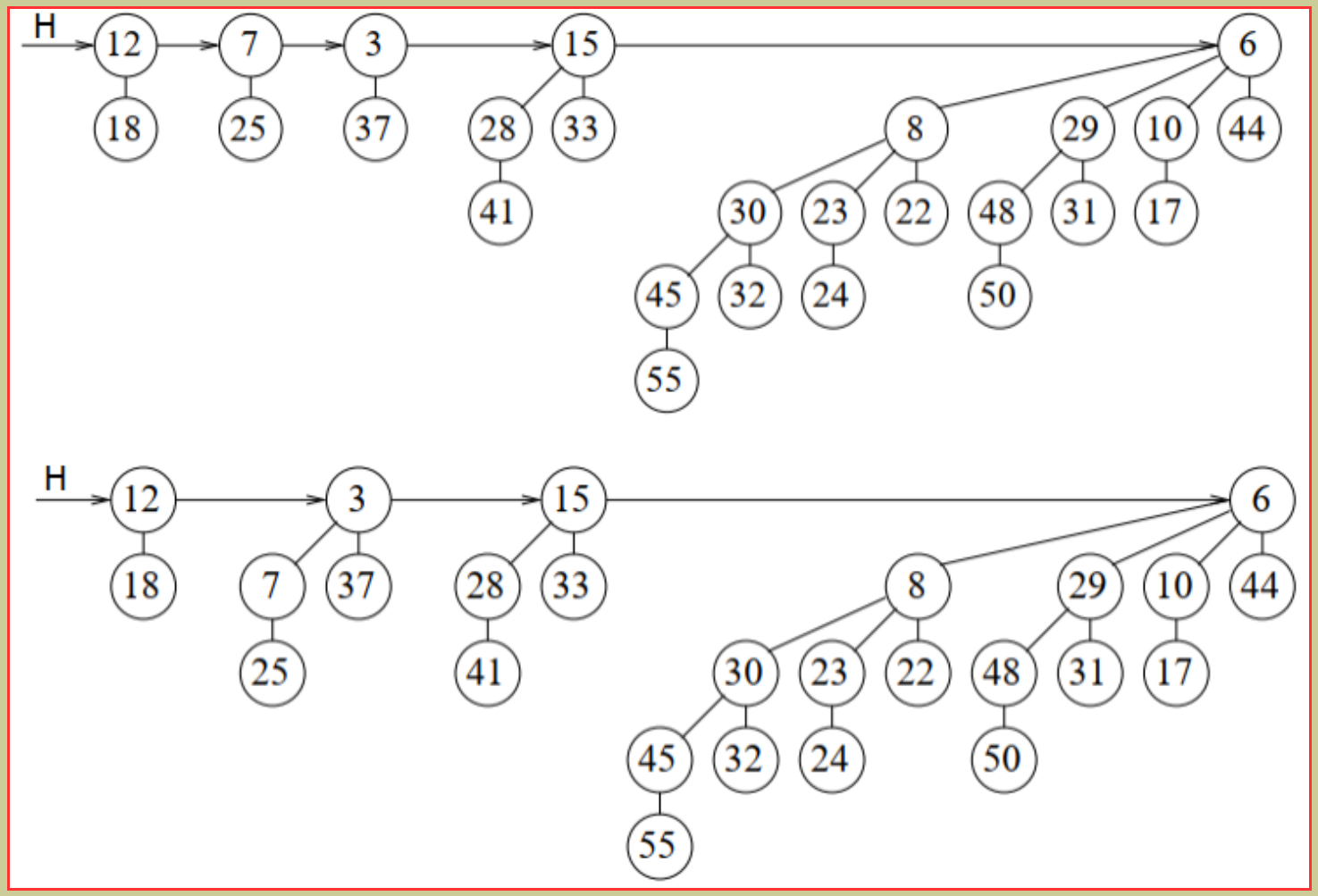
Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - União de dois Heaps (operação auxiliar) – caso 3



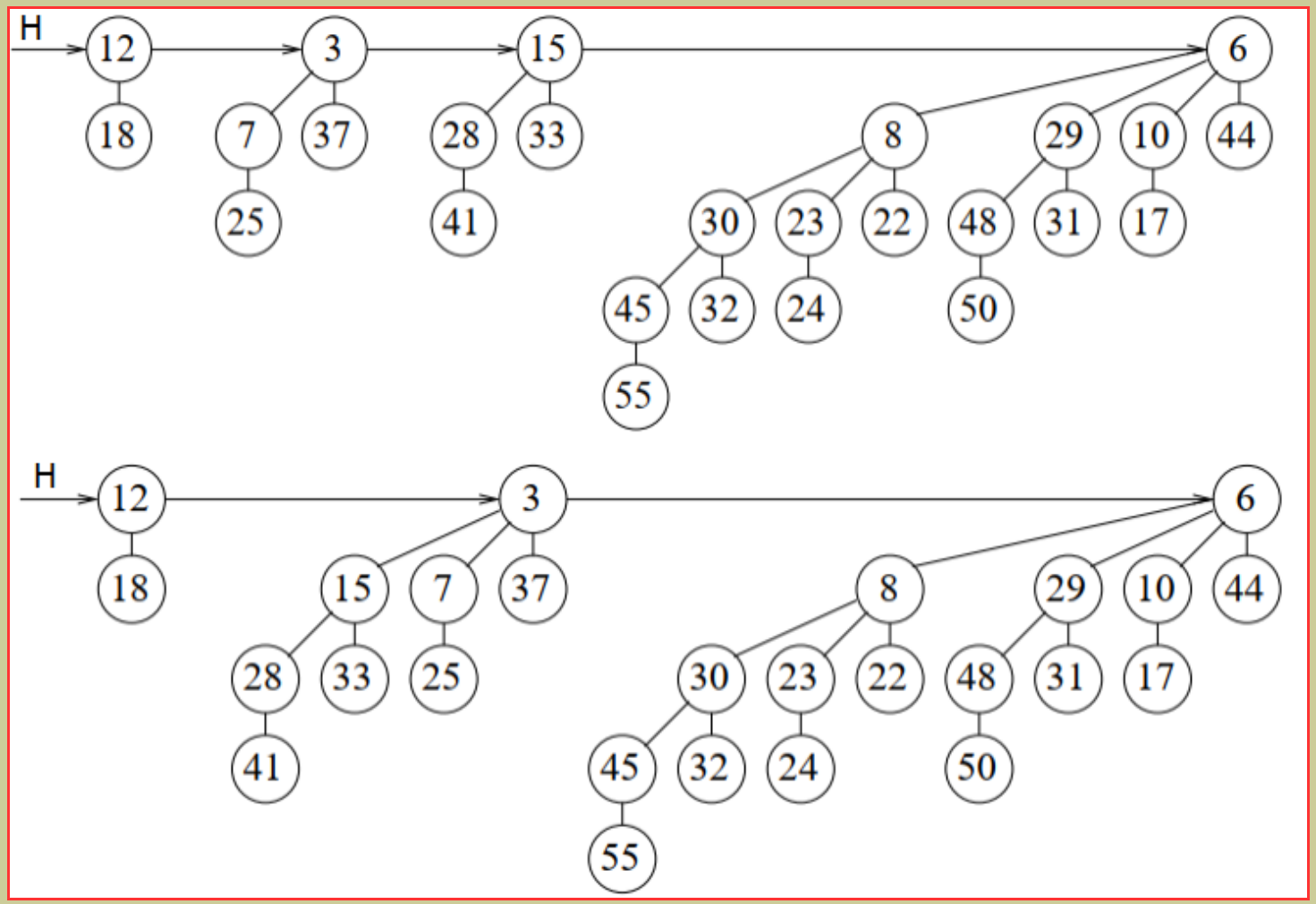
Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - União de dois Heaps (operação auxiliar) – caso 2 + caso 4



Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - União de dois Heaps (operação auxiliar) – caso 3



Operação principal inserir(x, H)

- Inserir um nodo com o elemento x num Heap H
  - União de dois Heaps (operação auxiliar)

```
algoritmo uniao(H1, H2)
  H ← criar()
  H ← juntar(H1, H2)
  se (H = NULL) então     devolver H
  fim_se
  antX ← NULL
  X ← H
  sucX ← irmao(X)
  enquanto (sucX ≠ NULL) fazer
     se (grau(X) ≠ grau(sucX) ou irmao(sucX) ≠ NULL e grau(irmao(sucX)) = grau(X))
     então
        antX ← X          // casos 1 e 2
        X ← sucX
     senão
...
```

Operação principal inserir(x, H)

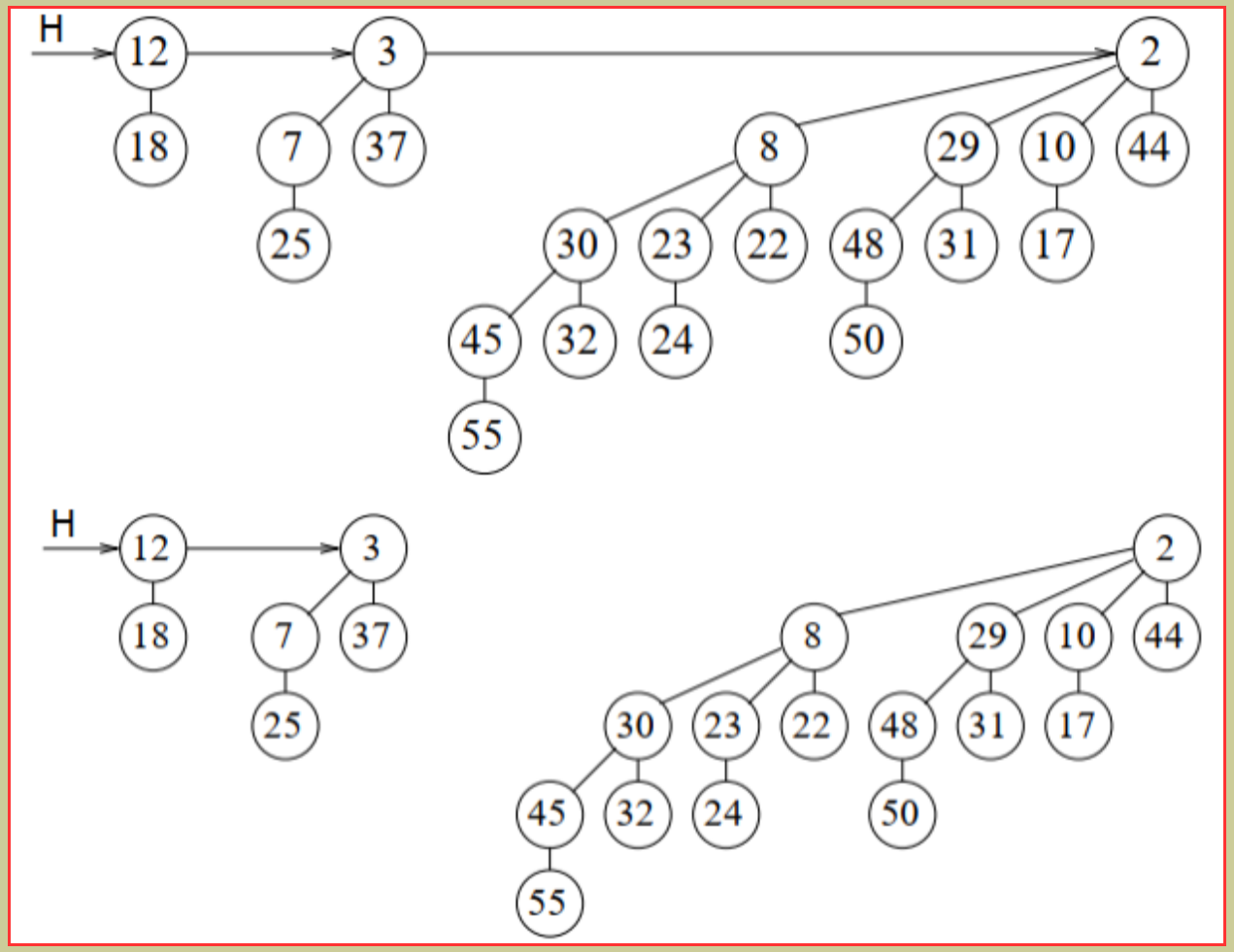
```
...
        se (chave(X) ≤ chave(sucX) então   //  caso 3
            irmão(X) ← irmão(sucX)
            juntarAB(sucX, X)
        senão
            se (antX = NULL) então          //  caso 4
                H ← sucX
            senão
                irmão(antX) ← sucX
            fim_se
            juntarAB(X, sucX)
            X ← sucX
        fim_se
        sucX ← irmão(X)
    fim_se
  fim_enquanto
  devolver H
fim_algoritmo
```

Operação principal remover(H)

- Extrair e remover o elemento com menor valor da chave
  - para extrair extrair o elemento com menor valor do Heap,
    - determinar a raiz (da lista de raizes de H) com menor valor da chave (O(log n))
  - para remover o elemento com menor valor do Heap,
    - remover a raiz anterior e pegar na árvore com aquela raiz, seja B_k
    - criar um novo heap H' com as as subárvores da B_k anterior, B_0, B_1, . . . ,B_{k-1}
    - realizar a operação de **união** entre os heaps H e H' (O(log n))
  - ordem de complexidade: O(log n)

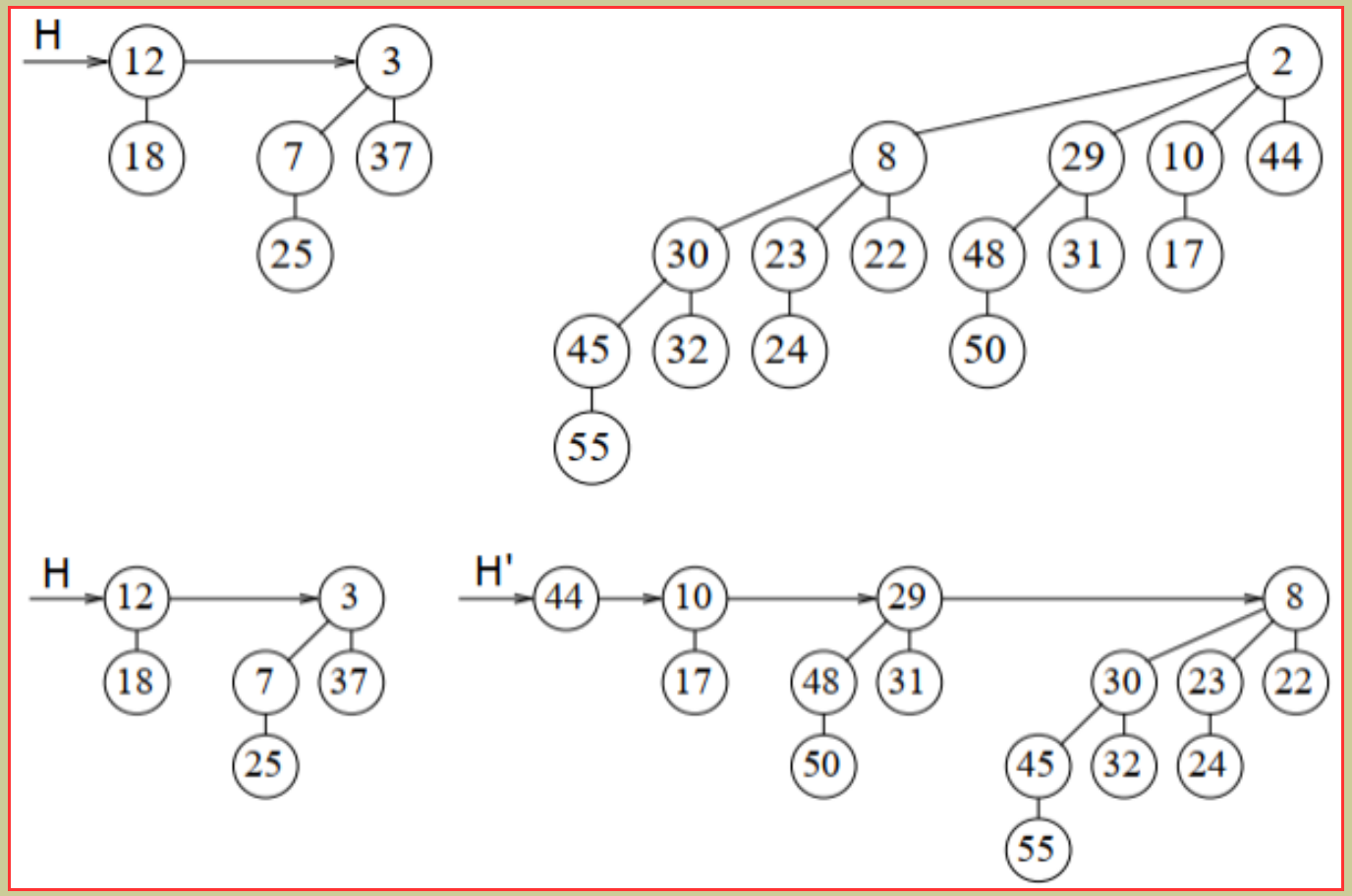
Operação principal remover(H)

- Extrair e remover o elemento com menor valor da chave
  - extrair de H a árvore binomial com o valor 2 na raiz, que é a com menor chave (B_4)



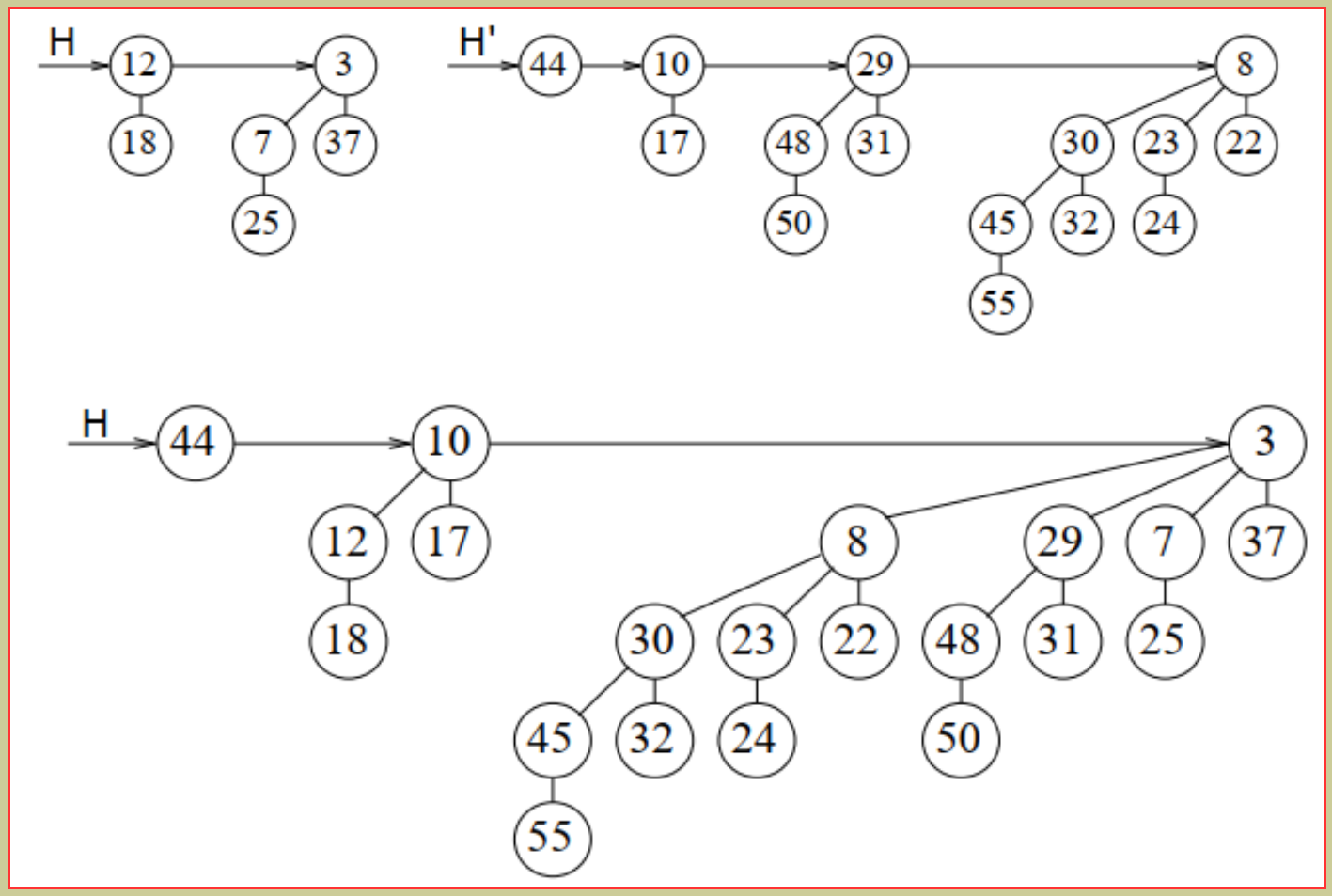
Operação principal remover(H)

- Extrair e remover o elemento com menor valor da chave
  - criar um novo heap H' a partir das subárvores de B_4: B_0, B_1, B_2 e B_3



Operação principal remover(H)

- Extrair e remover o elemento com menor valor da chave
  - fazer a união de H com H'



Operação principal remover(H)

- Extrair e remover o elemento com menor valor da chave

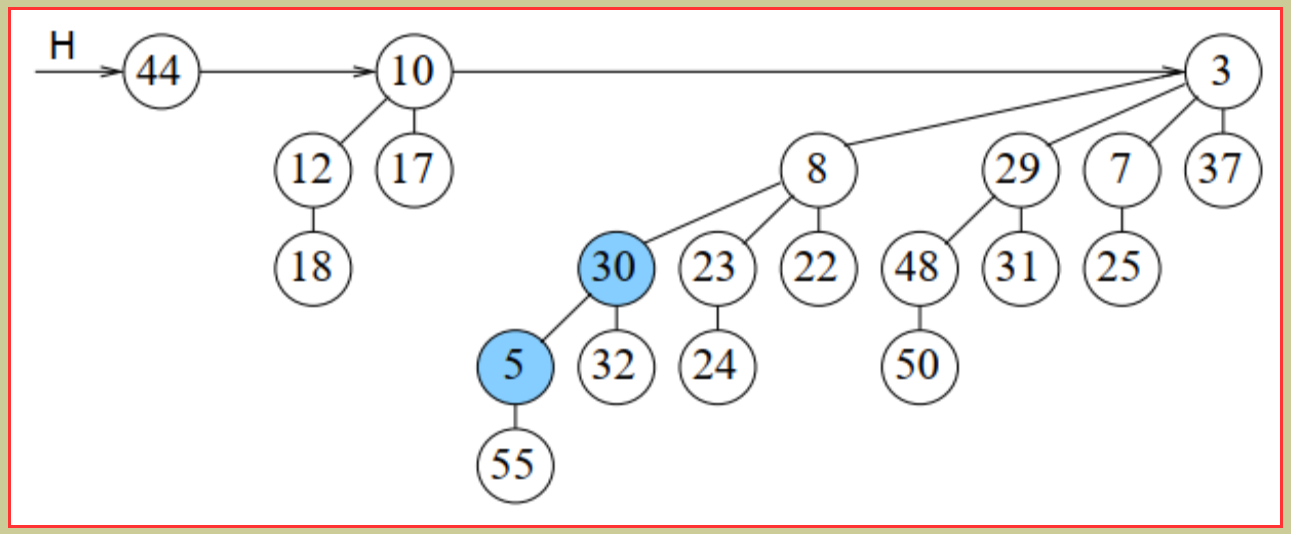
```
algoritmo remover(H)
   determinar a raiz P com a menor chave em H
   remover a árvore binomial com aquela raiz da lista de raizes de H
   H' ← criar()
   inverter a ordem da lista ligada de filhos de P e atribuí-la a H'
   H ← uniao(H, H')
   devolver P
fim_algoritmo
```

Operação secundária atualizarChave(x, k, H)

- Atualizar a chave de um elemento
  - operação que permite alterar o valor da chave de um elemento (atualizar a sua prioridade)
  - depois de atualizar a chave é necessário colocar o elemento na posição correta para mantermos a propriedade de heap
  - para tal, basta verificar se a nova chave é menor que o valor da chave do seu pai
    - enquanto isto acontecer, subir o nodo na árvore até à raiz (ver heap binária)
  - a complexidade do algoritmo está associada ao percurso de subida na árvore
    - como a altura da árvore é log n, este algoritmo tem complexidade O(log n)

Operação secundária atualizarChave(x, k, H)

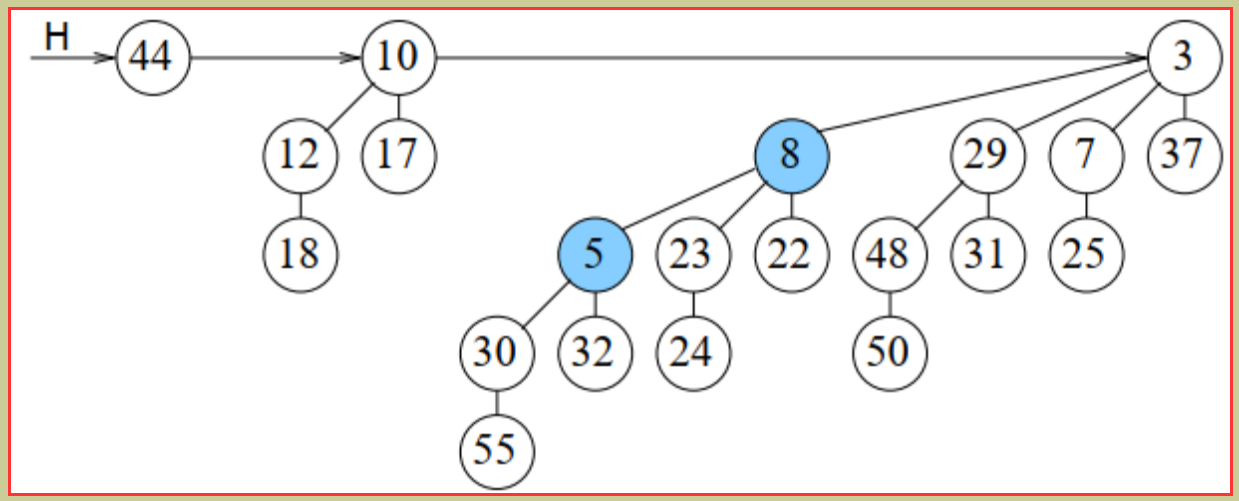
- Atualizar a chave de um elemento
  - O nodo com o valor 5 da chave 5 tinha antes o valor 45



  - Como 5 é menor do que o valor da chave do seu pai (30), deve-se trocar de valores

Operação secundária atualizarChave(x, k, H)

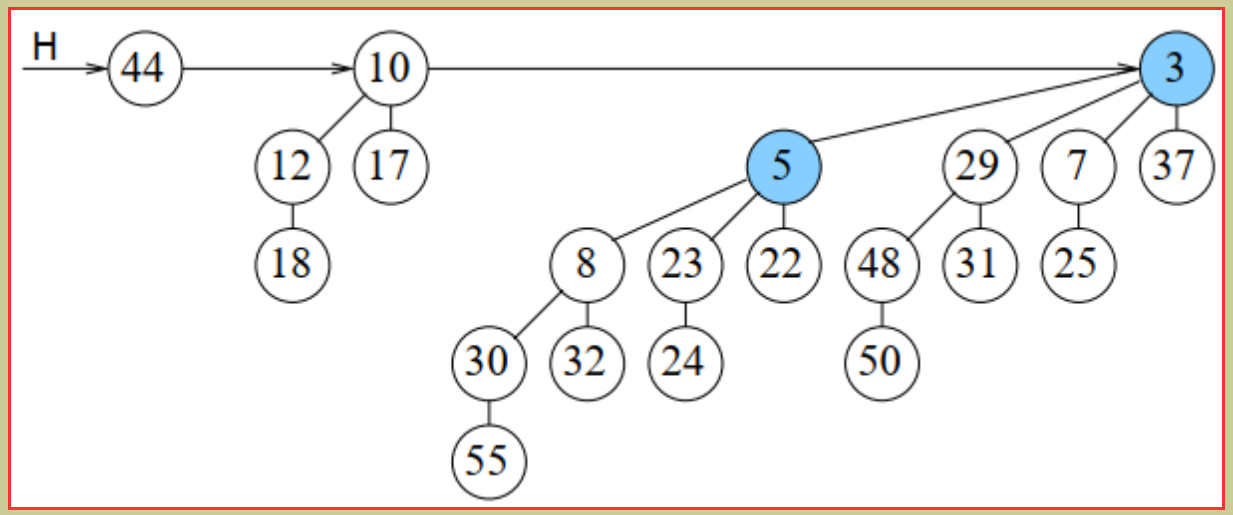
- Atualizar a chave de um elemento



- Como 5 é menor do que o valor da chave do seu pai (8), deve-se trocar de valores

Operação secundária atualizarChave(x, k, H)

- Atualizar a chave de um elemento



- Como 5 é maior do que o valor da chave do seu pai (8), o processo termina

Operação secundária atualizarChave(x, k, H)

- Atualizar a chave de um elemento

```
algoritmo atualizarChave(X, k, H)
  // atualizar o valor da chave do nodo X com o valor k
  se (k > chave(X)) então
     ERRO: "a nova chave é maior que a chave atual"
     TERMINAR
  fim_se
  chave(X) ← k
  P ← X
  Q ← pai(P)
  enquanto (Q ≠ NULL e chave(P) < chave(Q)) fazer
     trocar(chave(P), chave(Q))
     P ← Q
     Q ← pai(P)
  fim_enquanto
fim_algoritmo
```

Operação secundária removerElemento(x, H)

- Remover um elemento de um Heap
  - para remover um elemento de um heap binomial,
    - decrementar a sua chave para $-\infty$, e
    - extrair e remover o elemento com a menor chave

```
algoritmo removerElemento(x, H)
  // remover de H o nodo/elemento apontado por x
  atualizarChave(x, -∞, H)
  remover(H)
fim_algoritmo
```

Eficiência computacional

Ordens de complexidade das operações principais

Estrutura Abstrata de Dados	inserir	remover	consultar
Lista não ordenada (array)	O(1)	O(n)	O(n)
Lista ordenada (array)	O(n)	O(1)	O(1)
Lista não ordenada (listas ligadas)	O(1)	O(n)	O(n)
Lista ordenada (listas ligadas)	O(n)	O(1)	O(1)
Árvore binária pesquisa	O(n)	O(n)	O(n)
Árvore binária pesquisa balanceada	O(log n)	O(log n)	O(log n)
Heap binário	O(log n)	O(log n)	O(1)
Heap binomial	O(log n)	O(log n)	O(1)
Heap de Fibonacci			