

III. Estruturas de Dados Eficientes

Tópicos:

- Tipos Abstractos de Dados
- Estruturas de dados lógicas e físicas; refinamento
- Implementação de *Buffers* e *Dicionários*
- Tabelas de *hash*
- Árvore binárias de procura AVL
- *Heaps*

■ Tipos Abstractos de Dados (ADTs) ■

Constituem um instrumento fundamental de *abstracção*, separando a *interface* de uma estrutura de dados (o conjunto de operações disponíveis sobre ela) da sua *implementação* concreta.

Por exemplo, uma **fila de espera** é caracterizada pelas operações

enqueue :: **T** → **Queue** → **Queue**
first :: **Queue** → **T**
dequeue :: **Queue** → **Queue**
isEmpty :: **Queue** → **Bool**

É possível axiomatizar o comportamento a este nível abstracto, por exemplo:

$$\begin{array}{ll} \text{isEmpty}(q) \implies \text{first}(\text{enqueue}(x, q)) = x \\ \neg \text{isEmpty}(q) \implies \text{first}(\text{enqueue}(x, q)) = \text{first}(q) \\ \text{isEmpty}(q) \implies \text{dequeue}(\text{enqueue}(x, q)) = q \\ \neg \text{isEmpty}(q) \implies \text{dequeue}(\text{enqueue}(x, q)) = \text{enqueue}(x, \text{dequeue}(q)) \end{array}$$

■ Estruturas de Dados Lógicas ■

As estruturas de dados por sua vez podem ser caracterizadas a um nível mais abstracto a que chamaremos **lógico**, ou a um nível mais concreto a que chamaremos **físico**. Estruturas de dados lógicas incluem por exemplo as **sequências**, as **árvores**, ou os **grafos**.

Note-se que não se trata de ADTs, uma vez que lhes está já associada uma estrutura particular, linear, hierárquica, ou relacional.

É comum definir-se estruturas de dados por especialização de outras. Por exemplo,

- Uma árvore é um caso particular de grafo (acíclico e com raíz).
- Uma **árvore binária** é um caso particular de árvore (cada nó tem no máximo dois descendentes).
- Uma **árvore binária de pesquisa** é um caso particular de árvore binária (com um invariante que estabelece uma relação de ordem *in order*).

■ Estruturas de Dados Físicas ■

Cada estrutura de dados lógica pode ser implementada de diversas formas, a que correspondem diferentes estruturas de dados *físicas*.

Por exemplo uma sequência de elementos de um mesmo tipo pode ter

Implementação Contígua: um [array](#) (estrutura indexada com acesso em tempo constante, podendo ser estático ou dinâmico);

Implementação Ligada: o acesso ao elemento seguinte é feito através de um campo “próximo”.

Note-se que se um [array](#) é já uma estrutura física, a noção de sequência ligada é algo que se encontra ainda ao nível lógico, podendo ser implementada também sobre um [array](#), ou então como uma [lista ligada](#), alocada dinamicamente, com utilização de apontadores.

■ Classes fundamentais de ADTs: *Buffers* ■

Buffers: possuem operações de **inserção** e de **extracção** de elementos (além de teste de vazio).

Diferentes tipos desta classe diferem na *estratégia* ou ordem pela qual os elementos são extraídos depois de inseridos

Pilha / Stack. Estratégia LIFO *Last In, First Out.*
Operações *Push, Pop*

Fila / Queue. Estratégia FIFO *First In, First Out.*
Operações *Enqueue, Dequeue*

Fila com Prioridades / Priority Queue. Estratégia *Highest Priority.*
Operações *Insert-with-Priority, Pull*

■ Implementação Eficiente de *Buffers* ■

Pilha / *Stack*

- Implementação contígua: *array* (estático ou dinâmico, reallocável)
- Implementação ligada: *lista ligada*; *Push* e *pop* implementados à cabeça.

Fila / *Queue*

- Implementação contígua: *array com circularidade* (. . .)
- Implementação ligada: *lista ligada* com manutenção de um apontador adicional para o último elemento. *Enqueue* no fim da lista; *dequeue* à cabeça da lista.

Fila de Prioridades / *Priority Queue*

- *Heap* (às vezes traduzido como *amontoado*): tal como a árvore binária de pesquisa, é um tipo particular de árvore binária com um invariante de ordem. Veremos que a sua implementação típica é *contígua*, sobre um *array*.

■ Classes fundamentais de ADTs: Dicionários ■

Dicionário, Array Associativo, ou Mapeamento.

armazena pares $chave \mapsto valor$, tendo a semântica de uma *função finita*.

As operações básicas são

1. A **inserção** de um par $chave \mapsto valor$;
2. A **alteração** do *valor* associado a uma *chave*;
3. A **consulta** com base numa *chave*, podendo obter-se como resultado um *valor* ou a indicação de que a chave não ocorre no dicionário;
4. A **remoção** de um par, dada a respectiva *chave*.

Note-se que as operações 1 e 2 podem ser implementadas pela mesma operação.

■ Implementação de Dicionários ■

Alguns casos particulares:

- Se as chaves forem *números naturais*, um simples *array* implementa um dicionário. O tipo do *array* deve ser o tipo dos valores que se pretende associar às chaves. Um valor especial pode ser usado para sinalizar que uma chave não ocorre no dicionário.
- Se as chaves forem de um tipo que admita uma noção de *ordem*, podem ser implementadas por uma árvore binária de pesquisa ou por uma árvore *AVL*.

Caso geral:

- Se o universo de chaves for muito grande, torna-se incomportável a utilização de *arrays* de forma directa. As *tabelas de hash* são estruturas de dados (*lógicas*) que permitem implementar dicionários mantendo o tempo de execução das operações *tendencialmente constante*.

■ Tabelas de *Hash* ■

Comecemos por admitir que:

- as chaves são números naturais potencialmente muito grandes, e
- o número máximo de pares que se pode armazenar no dicionário (a sua *capacidade*) é muito menor do que o número de diferentes chaves possíveis.

Uma **função de hash** é uma função *não-injectiva*, que mapeia chaves para o domínio de um *array* concreto. Sendo uma função, quando invocada com um determinado argumento calcula sempre o mesmo resultado.

Uma característica desejável destas funções é a **uniformidade**: todas as posições do *array* devem ter a mesma probabilidade de ser calculadas como resultado. Quando *cap* é um número primo, a seguinte função é razoavelmente uniforme:

```
int hash (int k, int cap) {  
    return k%cap;  
}
```

■ Tabelas de *Hash* ■

Uma tabela de *hash* de capacidade cap é então uma estrutura de dados lógica que **implementa um dicionário** de pares de tipo $\mathbf{N} \rightarrow V$, e que consiste em:

1. Uma função de hash de tipo $\mathbf{N} \rightarrow \{0, \dots, cap - 1\}$;
2. Uma das seguintes estruturas físicas, de acordo com a *estratégia de resolução de colisões* adoptada:
 - Um *array* com posições $\{0, \dots, cap - 1\}$ de pares (\mathbf{N}, V) ,
ou
 - Um *array* com posições $\{0, \dots, cap - 1\}$ de (apontadores para) listas ligadas de pares (\mathbf{N}, V) .

As colisões são a consequência natural da redução de um espaço de chaves \mathbf{N} a um conjunto $\{0, \dots, cap - 1\}$.

■ Tabelas de *Hash* – condicionamento de chaves ■

O conceito é facilmente generalizável a qualquer conjunto de chaves.

Se as chaves não forem números naturais, devem ser previamente condicionadas, i.e. mapeadas em números naturais, novamente de forma determinista e tão uniforme quanto possível.

```
int condition (char *s) {  
    int r = 0;  
    while (*s) {  
        r += *s++;  
    }  
    return r;  
}
```

■ Tabelas de *Hash* – Factor de Carga e Redimensionamento ■

O tempo de execução “tendencialmente constante” pressupõe uma carga razoavelmente pequena da tabela:

$$\alpha = \frac{\text{número de chaves inseridas}}{\text{capacidade da tabela}}$$

Tipicamente considera-se $\alpha < 0.8$ adequado; caso este valor seja ultrapassado pode-se proceder ao redimensionamento da tabela com duplicação da sua capacidade.

Em termos amortizados esta operação (de tempo real linear na capacidade da tabela) comporta-se como executando em tempo $\Theta(1)$.

■ Tabelas de *Hash* – Tratamento de Colisões ■

Sendo a capacidade da tabela muito menor do que a cardinalidade do conjunto de chaves, ocorrerão inevitavelmente *colisões*: a função de *hash* calcula a mesma posição do *array* para duas chaves diferentes, e quando se tenta inserir a segunda observa-se que aquela posição se encontra já preenchida.

- **Open addressing:**
procurar-se-á inserir a segunda chave numa outra posição do *array*, usando um método que possa ser reproduzido (quando se efectuar consultas . . .);
- **Closed addressing:**
alterar-se-á a estrutura de dados por forma a permitir que mais do que um par *chave* \mapsto *valor* sejam acomodados na *mesma* posição do *array*. Neste caso a consulta poderá envolver uma pesquisa neste conjunto de pares.

■ Tabelas de Hash – Open Addressing ■

O método mais natural de endereçamento aberto é conhecido por *linear probing*: a colisão resolver-se inserindo na posição seguinte do vector.

Sejam k_1, k_2 duas chaves tais que $\text{hash}(k_1) = \text{hash}(k_2) = p$, e efectuamos as operações $\text{insert}(k_1, v_1)$ e $\text{insert}(k_2, v_2)$ por esta ordem:

empty, –		empty, –		empty, –
empty, –		empty, –		empty, –
empty, –		empty, –		empty, –
empty, –		p	k_1, v_1	empty, –
empty, –			empty, –	
empty, –			empty, –	
empty, –			empty, –	

\Rightarrow

empty, –		empty, –		empty, –
empty, –		empty, –		empty, –
empty, –		empty, –		empty, –
p	k_1, v_1			
		empty, –		
		empty, –		
		empty, –		

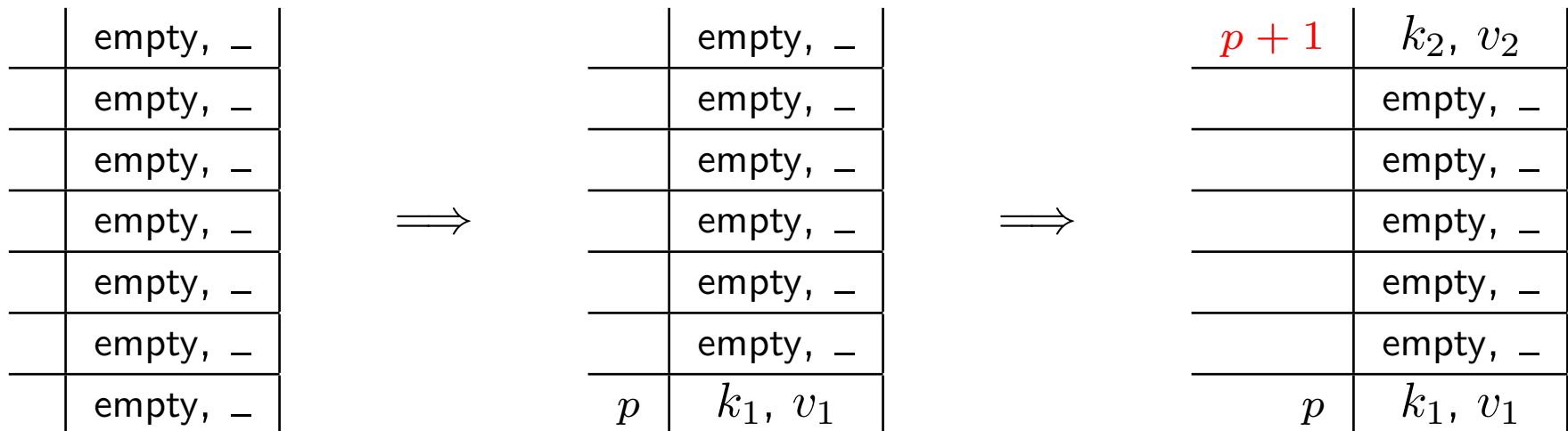
\Rightarrow

empty, –		empty, –		empty, –
		empty, –		empty, –
		empty, –		empty, –
p	k_1, v_1			
$p + 1$	k_2, v_2			
		empty, –		
		empty, –		

Note-se que é essencial armazenar as chaves juntamente com os valores, uma vez que na operação de consulta com k_2 não basta procurar na posição $\text{hash}(k_2) = p$, é necessário repetir o *linear probing* até se encontrar a chave k_2 !

■ Tabelas de *Hash* – Open Addressing ■

Linear probing é circular, $p + 1$ é de facto $(p + 1) \% cap$



Naturalmente, a operação de consulta deve reproduzir a mesma sequência utilizada na inserção.

■ Tabelas de *Hash* – Open Addressing ■

Além disso, “próxima posição” deve ser de facto interpretado como “próxima posição *livre*”. Seja $h(k_0) = p + 1$ e $h(k_1) = h(k_2) = p$.

The diagram illustrates a sequence of three states of a hash table with 8 slots:

- Initial State:** All slots are empty.
- After Insertion 1:** Slot $p+1$ contains k_0, v_0 . An arrow points to the next state.
- After Insertion 2:** Slot p contains k_1, v_1 . An arrow points to the final state.
- Final State:** Slot p contains k_1, v_1 , slot $p+1$ contains k_0, v_0 , and slot $p+2$ contains k_2, v_2 . All other slots are empty.

	empty, –
$p + 1$	k_0, v_0
	empty, –
	empty, –

	empty, –
	empty, –
	empty, –
p	k_1, v_1
$p + 1$	k_0, v_0
	empty, –
	empty, –

	empty, –
	empty, –
	empty, –
p	k_1, v_1
$p + 1$	k_0, v_0
$p + 2$	k_2, v_2
	empty, –

- ⇒ Qual o critério de paragem (capacidade esgotada) de uma operação de inserção?
- ⇒ Qual o critério de paragem (chave inexistente) de uma operação de consulta?

■ Tabelas de *Hash* – Open Addressing ■

Seja $h(k_1) = h(k_2) = p$, e efectuemos as operações `insert(k_1, v_1)`, `insert(k_2, v_2)`, and `remove(k_1)` por esta ordem:

	empty, –
	empty, –
	empty, –
p	k_1, v_1
$p + 1$	k_2, v_2
	empty, –
	empty, –

⇒

	empty, –
	empty, –
	empty, –
p	empty, –
$p + 1$	k_2, v_2
	empty, –
	empty, –

⇒ O que sucede quando se efectuar agora uma consulta com a chave k_2 ?

■ Tabelas de *Hash* – Open Addressing ■

Não é adequado marcar as posições onde ocorreram remoções como “empty”. Utiliza-se uma chave alternativa “**removed**”, que indica que uma pesquisa deve *continuar* para além daquela posição.

	empty, –
	empty, –
	empty, –
p	k_1, v_1
$p + 1$	k_2, v_2
	empty, –
	empty, –

⇒

	empty, –
	empty, –
	empty, –
p	removed, –
$p + 1$	k_2, v_2
	empty, –
	empty, –

⇒ Em cenários com remoções frequentes, “open addressing” pode não ser uma boa escolha

■ Tabelas de *Hash* – *Open Addressing* ■

Um segundo problema da técnica de *linear probing* é a formação de *clusters*: Seja $r = \frac{1}{cap}$ a probabilidade inicial de cada posição ser preenchida e efectuemos a mesma sequência de suas inserções:

(r)	empty, –

⇒

(r)	empty, –
(r)	empty, –
(r)	empty, –
(0) p	k_1, v_1
(2*r)	empty, –
(r)	empty, –
(r)	empty, –

⇒

(r)	empty, –
(r)	empty, –
(r)	empty, –
(0) p	k_1, v_1
(0) $p + 1$	k_2, v_2
(3*r)	empty, –
(r)	empty, –

Este fenómeno de aumento da probabilidade de inserção em posições subsequentes às já preenchidas resulta na formação de “clusters”, que deterioram o comportamento da tabela. Este fenómeno pode ser resolvido com a utilização de outras técnicas de *open addressing*, como *quadratic probing*.

■ Tabelas de Hash – *Closed Addressing* ■

A solução comum para fazer “caber” vários pares $chave \mapsto valor$ na mesma posição de um *array* é externalizar a informação, criando uma **lista ligada** cujo endereço inicial é guardado no array:

```
typedef struct node {  
    char key[MAXSTR];  
    Infotype info;  
    struct node * next;  
};  
  
typedef struct node *Hashtable[CAP];
```

Chama-se a esta implementação uma tabela *encadeada* (tabela com “chaining”)

■ Tabelas de Hash – Closed Addressing ■

A inserção na tabela faz-se agora mediante uma inserção na lista ligada apropriada.

Null		Null		Null	
Null		Null		Null	
Null		Null		Null	
Null	\Rightarrow	$p \rightarrow (k_1, v_1) \rightarrow \text{Null}$		\Rightarrow	$p \rightarrow (k_2, v_2) \rightarrow (k_1, v_1) \rightarrow \text{Null}$
Null		Null		Null	
Null		Null		Null	
Null		Null		Null	

■ Tabelas de Hash – *Closed Addressing* ■

A inserção na tabela faz-se agora mediante uma inserção na lista ligada apropriada (a versão seguinte assume que a chave key não foi ainda inserida).

```
void insert (Hashtable t, int n, char key[], Infotype i)
{
    node *new = malloc(sizeof(struct node));
    int h = hash(condition(key), n);
    strcpy(new->key, key);
    new->info = i;
    new->next = t[h];
    t[h] = new;
}
```

A consulta por sua vez implicará uma pesquisa linear nesta lista.

Exercício:

⇒ Escrever as funções de inicialização, consulta, remoção, e actualização

■ **Tabelas de Hash – Closed Addressing** ■

Pontos a reter:

- Não é suposto que estas listas cresçam indefinidamente: apesar de em teoria poder ser $\alpha > 1$, o tempo de execução das operações deixaria de ser “tendencialmente constante” . . .
 - As tabelas devem pois ser redimensionadas quando necessário, tal como nas tabelas com endereçamento aberto, assegurando-se um factor de carga pequeno.
- ⇒ Haverá vantagens em manter as listas ordenadas?

■ Tabelas de *Hash* – Dados Empíricos ■

O quadro seguinte contém o *número de comparações* efectuadas numa consulta de uma tabela com 900 chaves.

Factor de carga α	0.1	0.5	0.8	0.9	0.99	2.0
Consulta com sucesso, <i>chaining</i>	1.04	1.2	1.4	1.4	1.5	2.0
Consulta com sucesso, <i>quadratic probing</i>	1.04	1.5	2.1	2.7	5.2	–
Consulta com sucesso, <i>linear probing</i>	1.05	1.6	3.4	6.2	21.3	–
Consulta sem sucesso, <i>chaining</i>	0.10	0.5	0.8	0.9	0.99	2.0
Consulta sem sucesso, <i>quadratic probing</i>	1.13	2.2	5.2	11.9	126	–
Consulta sem sucesso, <i>linear probing</i>	1.13	2.7	15.4	59.8	430	–

Fonte:

Kruse R.L., Leung B.P., Tondo C.L., *Data Structures and Program Design in C*. Prentice-Hall, 2nd. ed., 1991

■ Tabelas de *Hash* – Conclusões ■

- Para factores de carga razoáveis (≤ 0.8), ambas as soluções alcançam número constante de comparações nas consultas. Mas esta contagem não é o único aspecto relevante!
- *Open addressing* devidamente optimizado (*quadratic probing*, redimensionamento dinâmico para evitar factores de carga elevados) pode ser uma boa escolha, porque
 - não é penalizador em termos de espaço, ao contrário de implementações baseadas em *closed addressing* / *chaining*.
 - é totalmente estático, o que apresenta vantagens em termos de *caching*.
A grande desvantagem é a dificuldade em lidar com remoções.
- *Closed addressing* / *chaining* é eficiente e de programação simples, mas com um custo adicional de espaço relevante, e com as desvantagens inerentes à utilização de memória dinâmica em termos de localidade.

■ Pesquisa com base numa Ordem ■

Quando existe uma noção de ordem sobre as chaves, uma **implementação alternativa de um dicionário** baseia-se na *pesquisa binária* (em tempo $\log n$) para a operação fundamental de consulta. Se a informação for estática, as opções naturais serão:

- um vector ordenado pelas chaves;
- uma árvore binária de procura (“binary search tree”, BST) *completa*, em que todos os níveis (excepto o último) estão totalmente preenchidos.

Em ambos os casos existe uma ordem total sobre as chaves, tendo cada elemento dois vizinhos directos, o seu antecessor e o seu sucessor.

Exercício:

⇒ Dado um vector ordenado, escrever um algoritmo que constrói uma árvore binária completa contendo a informação nele armazenada.

■ Pesquisa com base numa Ordem ■

Deve ser claro nesta altura que:

- o tempo de execução é $O(\log n)$ no caso da pesquisa binária num vector ordenado;
 - numa BST em geral a operação de pesquisa executa em tempo $O(n)$;
 - numa BST cuja altura seja *logarítmica* no número de nodos da árvore, a operação de pesquisa executa em $O(\log n)$;
 - é este precisamente o caso das árvores completas
- ⇒ Como alcançar este comportamento logarítmico no caso de a informação ser dinâmica, podendo haver pesquisas em qualquer momento, alternadas com inserções e remoções?

■ Árvores AVL ■

Numa árvore completa, todos os nós (excepto possivelmente um) são equilibrados: as alturas das suas duas sub-árvoreas são iguais, ou seja $|h_e - h_d| = 0$.

Nas árvores AVL (Adelson-Velskii & E.M. Landis) este requisito é relaxado para $|h_e - h_d| \leq 1$: cada nodo pode estar equilibrado, mas também desequilibrado para a esquerda ou para a direita, desde que a diferença entre as alturas não exceda 1.

Este relaxamento permite ainda que a altura da árvore permaneça logarítmica, pelo que o mesmo acontece com o tempo de pesquisa. Mas tão importante como isto, tratando-se de estruturas dinâmicas, é o seguinte:

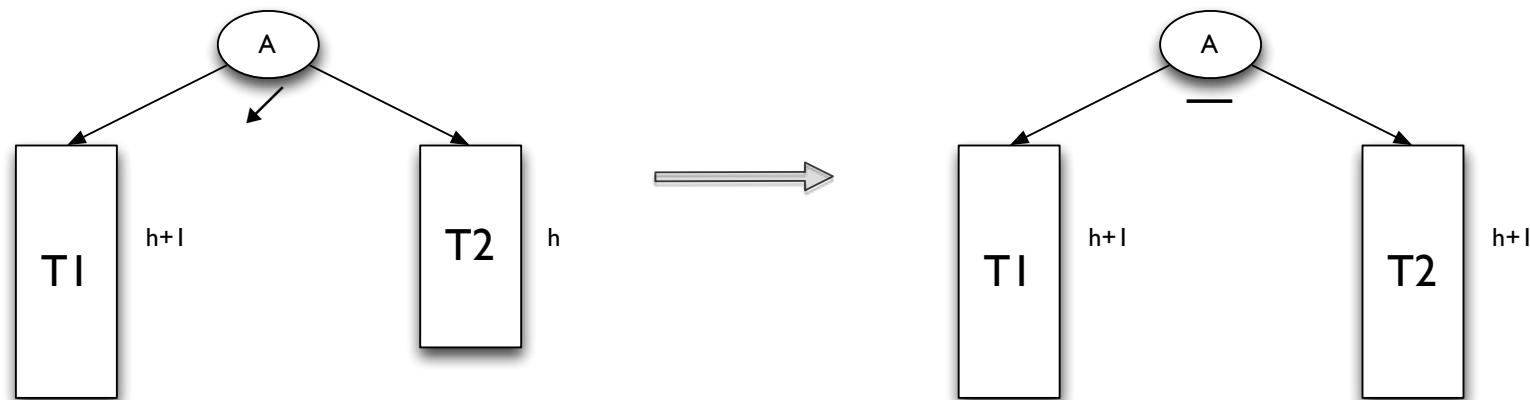
O tempo de execução dos algoritmos de inserção e remoção, modificados por forma a efectuarem o necessário ajuste das árvores preservando sempre o invariante $|h_e - h_d| \leq 1$, é também logarítmico.

Exemplificaremos com o algoritmo de inserção.

■ Árvores AVL ■

O algoritmo de inserção tem que detectar as situações em que a inserção numa das sub-árvores de um nodo provoca o aumento da altura dessa sub-árvore. Ilustraremos com a inserção na sub-árvore da *direita* (o outro caso é simétrico).

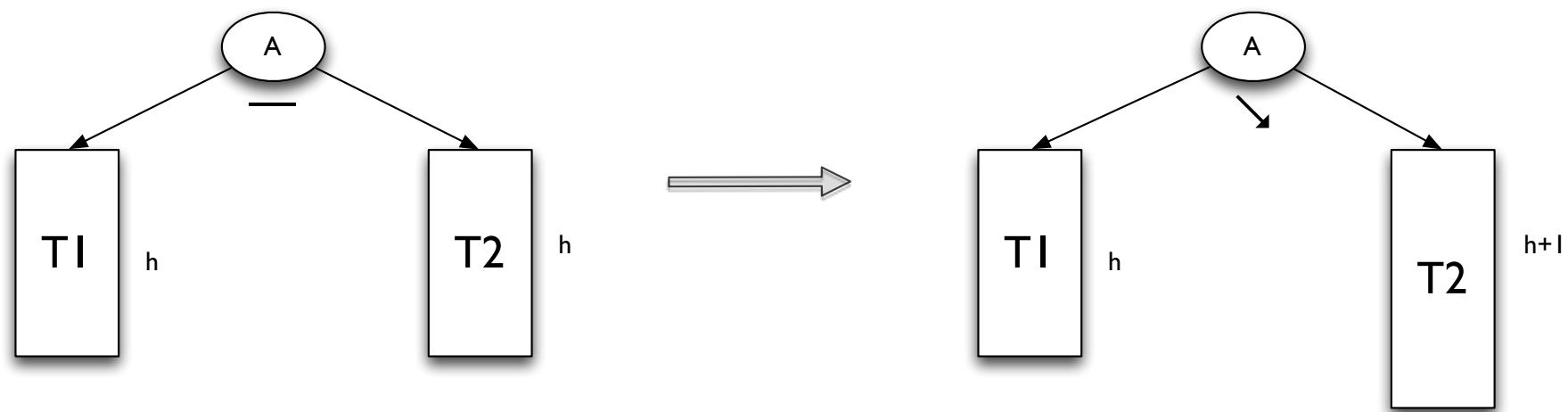
Caso 1: o nodo-raíz está à partida desequilibrado para a esquerda.



O nodo-raíz passa a **equilibrado**; a altura da árvore **mantém-se**.

■ Árvores AVL ■

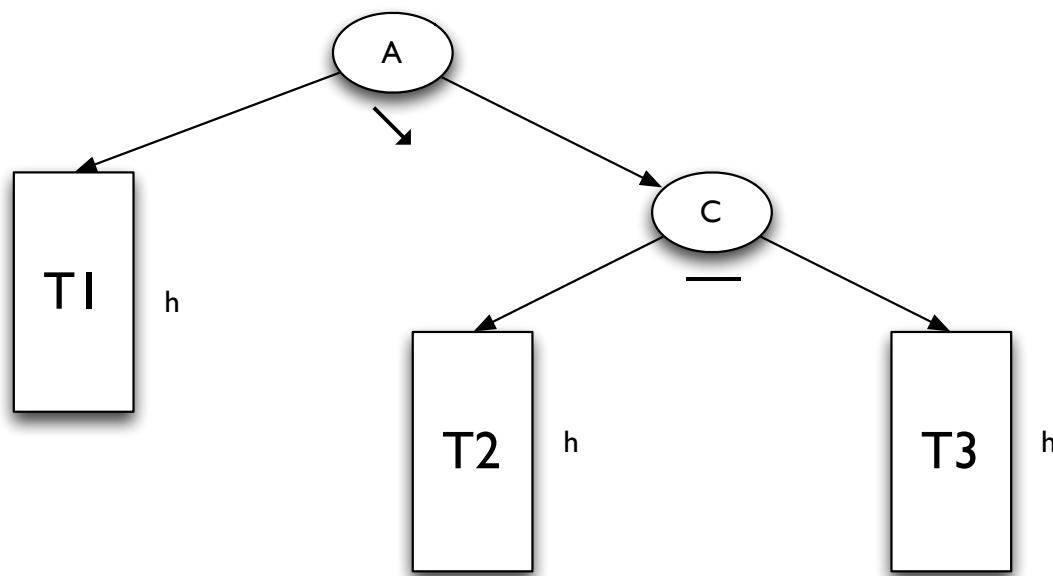
Caso 2: o nodo-raíz está à partida equilibrado.



O nodo-raíz passa apenas a estar desequilibrado para a **direita**, e a altura da árvore **aumenta**. Mas o invariante $|h_e - h_d| \leq 1$ não é violado.

■ Árvores AVL ■

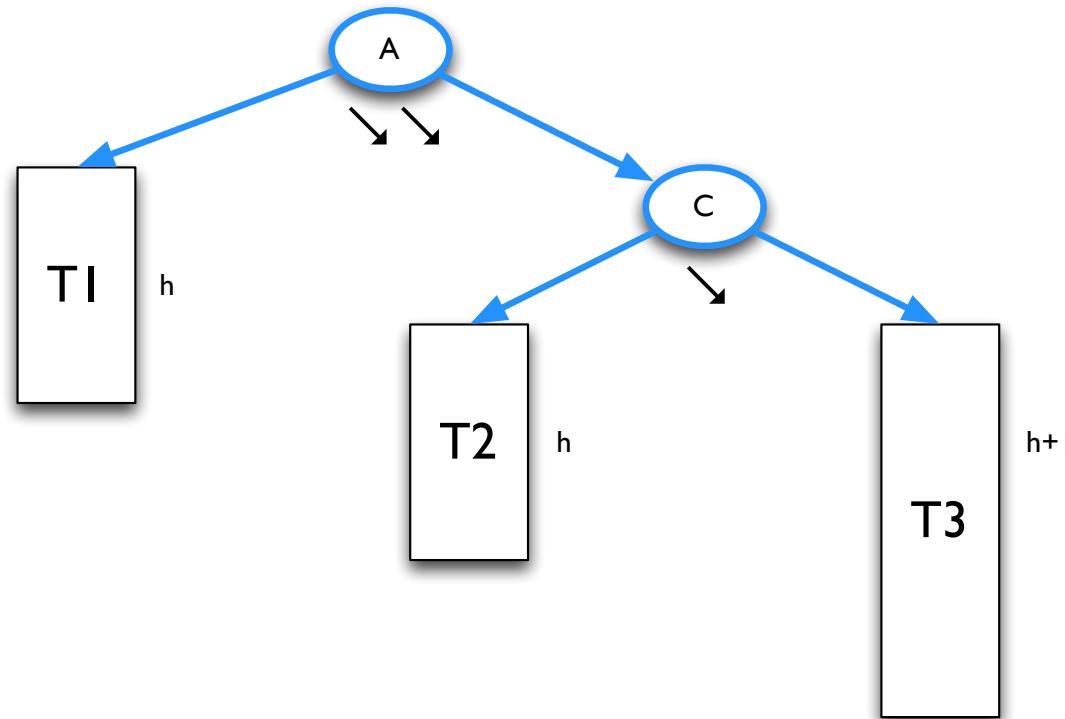
Para que ocorra uma violação do invariante num dos nós, esse nó tem que ter à partida sub-árvores com alturas diferentes ($|h_e - h_d| = 1$). Ilustremos com o caso de desequilíbrio para a direita, $h_d = h_e + 1$ (o outro caso é simétrico).



Note-se que teremos então que ter na sub-árvore da direita pelo menos um nó, e que este nó é necessariamente equilibrado. Sejam $h_e = h$ e $h_d = h + 1$.

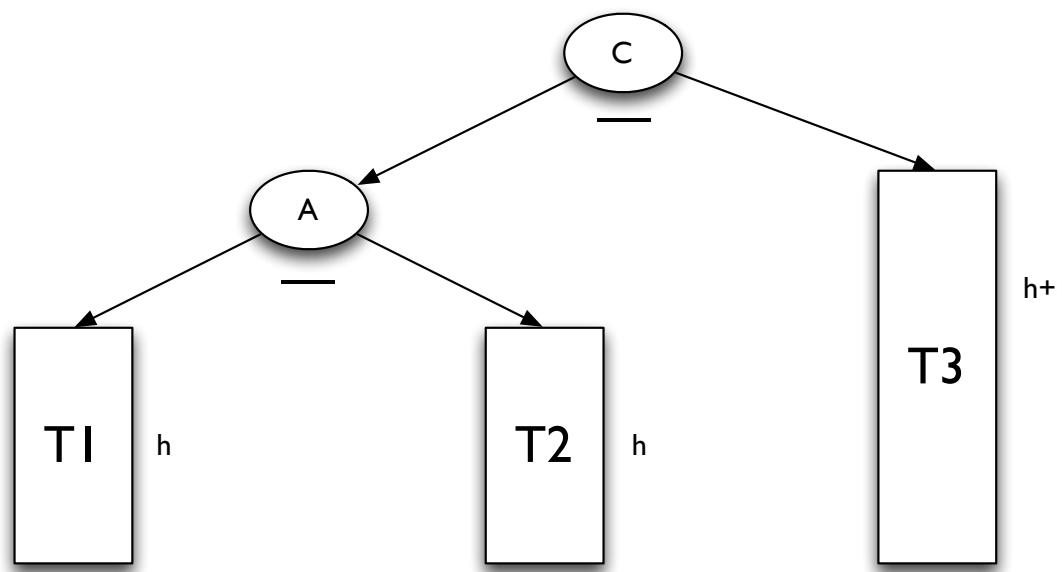
■ Árvores AVL ■

Caso 3a: a inserção provoca um aumento da altura de T3 para $h+1$, tornando-se assim $h_d = h + 2$.



■ Árvores AVL ■

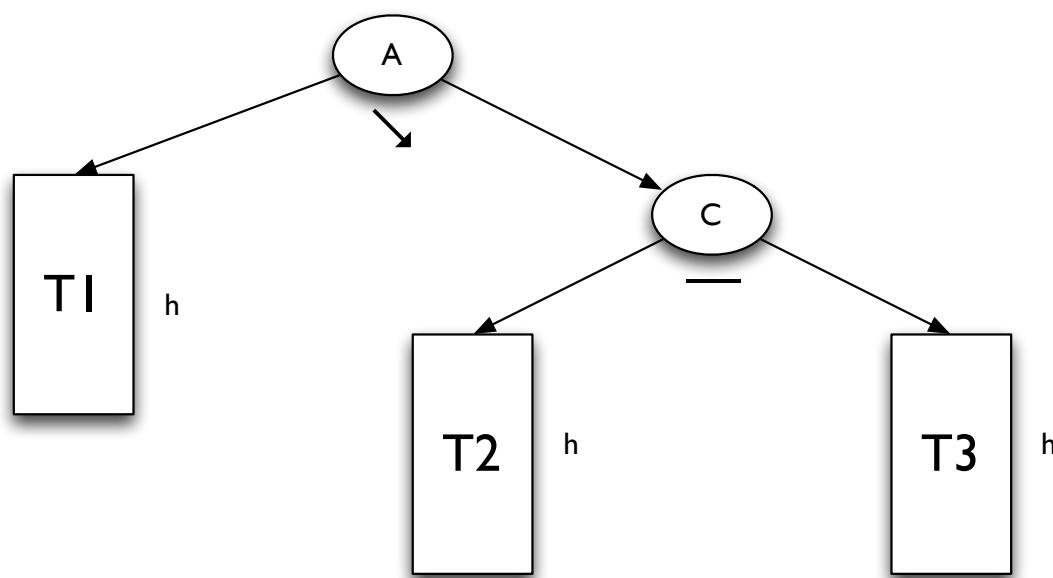
Caso 3a: Uma simples *rotação para a esquerda* repõe o equilíbrio em ambos os nós e reestabelece o invariante AVL com $|h_e - h_d| = 0$.



A altura da árvore **não aumenta**.

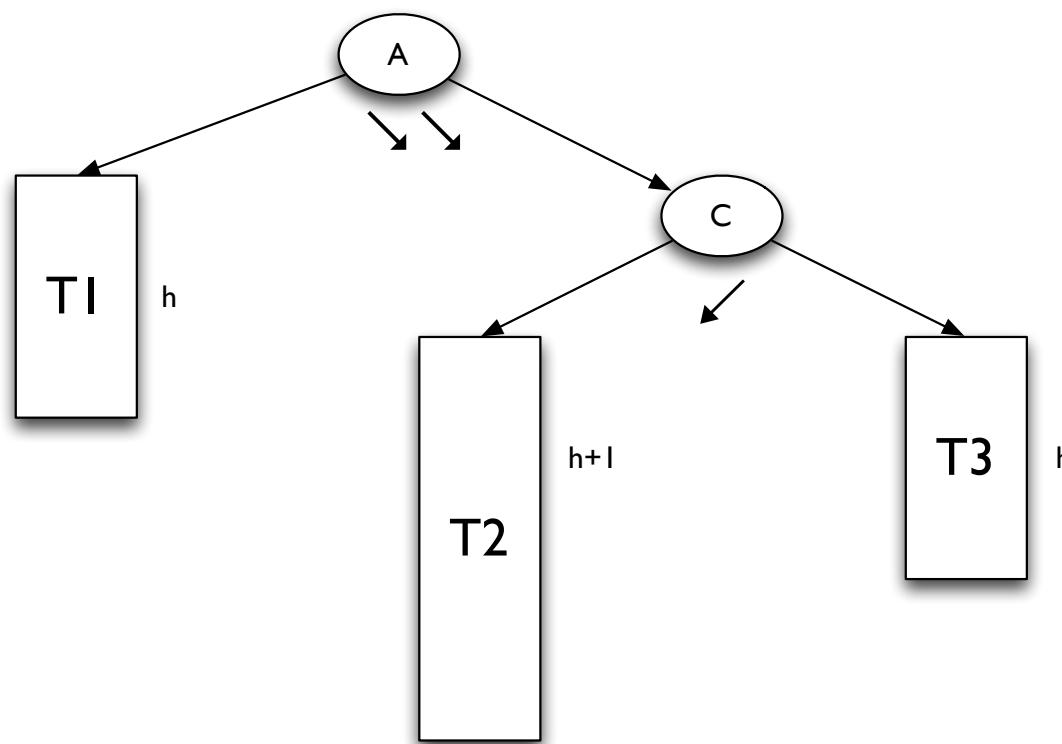
■ Árvores AVL

Retomando a árvore inicial:



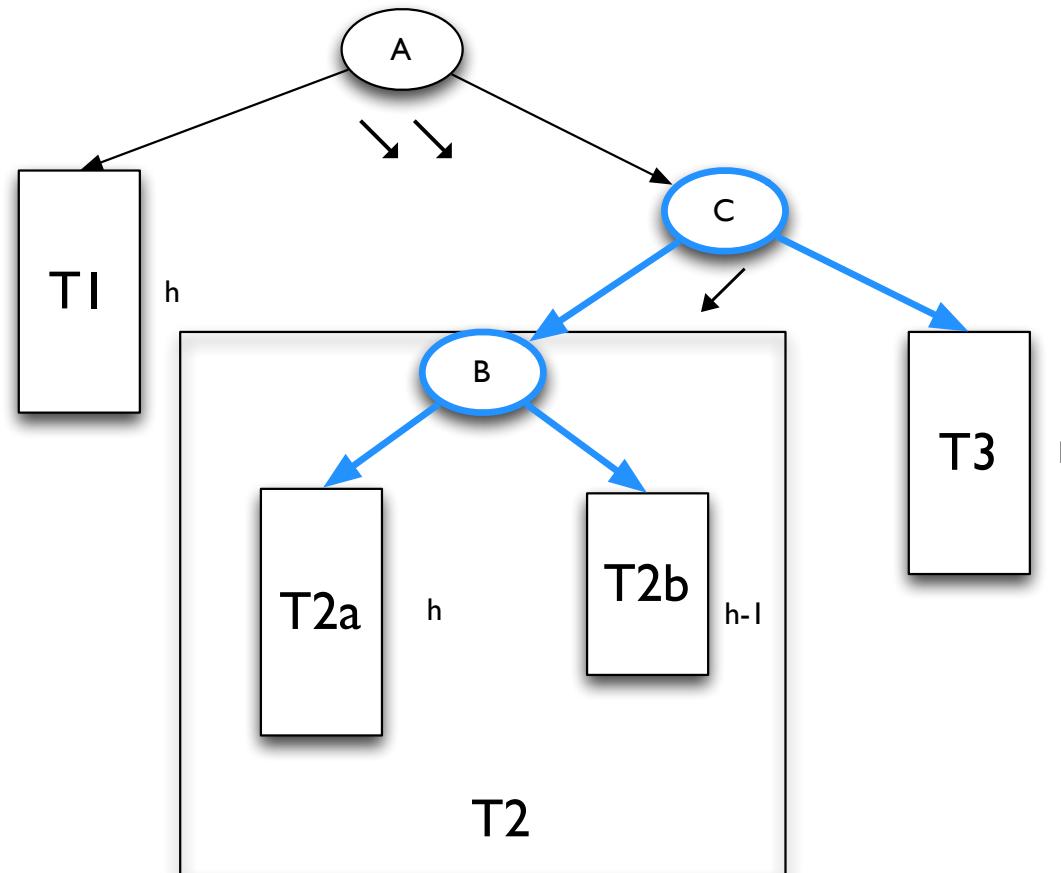
■ Árvores AVL ■

Caso 3b: a inserção provoca um aumento da altura de T2 para $h+1$, tornando-se assim $h_d = h + 2$.



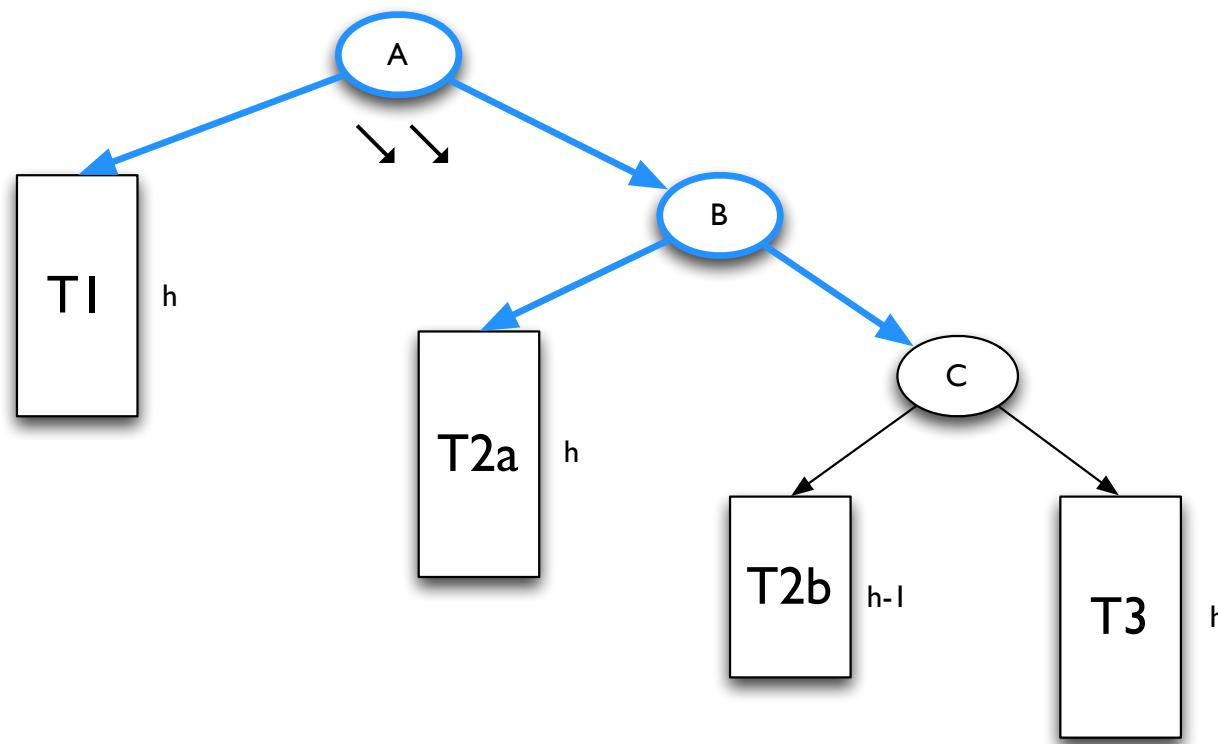
■ Árvores AVL

Caso 3b: observemos a estrutura de T2:



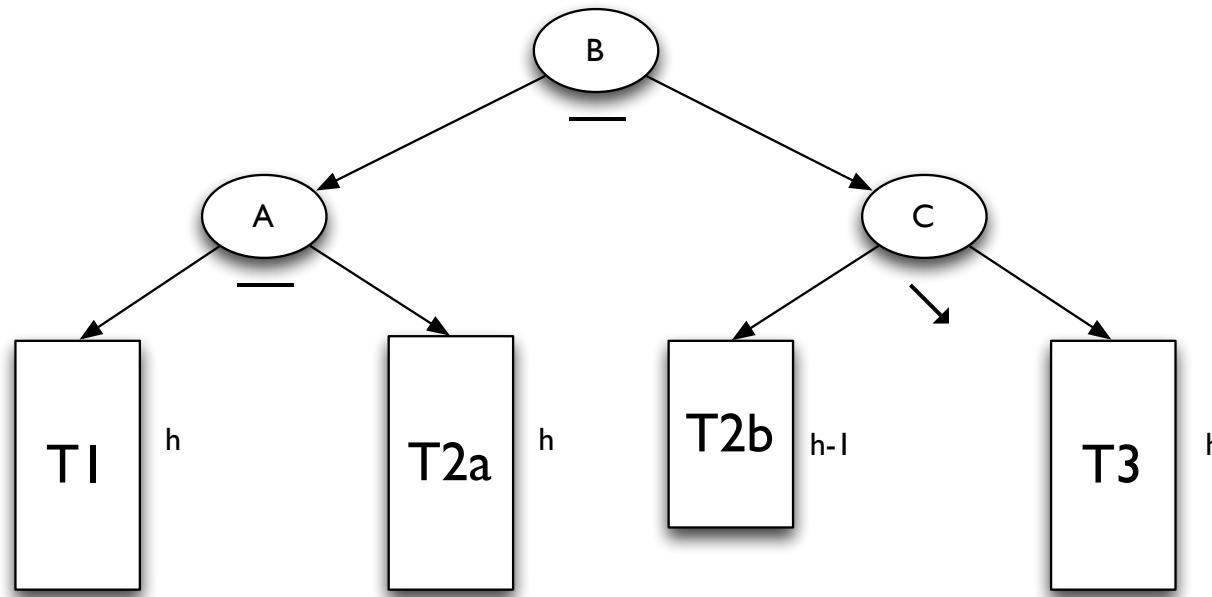
■ Árvores AVL

Caso 3b: primeiro passo: rotação com raiz em C para a direita :



■ Árvores AVL ■

Caso 3b: segundo passo: rotação global para a esquerda:



N.B. Pelo menos uma das árvores $T2a$ e $T2b$ tem altura h . O caso concreto altera os indicadores de A e C , mas não de B .

Novamente a altura da árvore **não aumenta**.

■ Árvores AVL – Conclusão ■

- As operações adicionais necessárias para a “auto-regulação” do equilíbrio dos nós (gestão de informação auxiliar e rotações) executam em tempo $\Theta(1)$.
- Tendo as árvores sempre altura logarítmica, as consultas são feitas em tempo $O(\log n)$.
- As operações de inserção e remoção, que numa BST de altura logarítmica executam em tempo também logarítmico, não são perturbadas pelas operações adicionais, mantendo-se em $O(\log n)$.

■ Esqueleto de uma Implementação ■

```
typedef int TreeKey;  
  
typedef ... TreeInfo;  
  
typedef enum balancefactor { LH , EH , RH } BalanceFactor;  
  
typedef struct treenode {  
    BalanceFactor bf;  
    TreeKey key;  
    TreeInfo info;  
    TreeNode *left;  
    TreeNode *right;  
} TreeNode;  
  
typedef TreeNode *Tree;
```

```
Tree rotateRight(Tree t) {
    Tree aux;

    if ((! t) || (! t->left)) {
        printf("Error\n");
    }
    else {
        aux = t->left;
        t->left = aux->right;
        aux->right = t;
        t = aux;
    }
    return t;
}
```

```
Tree insertTree(Tree t, TreeKey e, TreeInfo i, int *cresceu) {
    if (t==NULL) {
        t = (Tree)malloc(sizeof(struct treenode));
        t->key = e;
        t->info = i;
        t->right = t->left = NULL;
        t->bf = EH;
        *cresceu = 1;
    }
    else if (e==t->key) {
        t->info = i;      // actualização do valor associado à chave
    }
    else if (e>t->key)
        t = insertRight(t,e,i,cresceu);
    else
        t = insertLeft(t,e,i,cresceu);
    return(t);
}
```

```
Tree insertRight(Tree t, TreeKey e, TreeInfo i, int *cresceu) {
    t->right = insertTree(t->right,e,i,cresceu);
    if (*cresceu)
        switch (t->bf) {
            case LH:
                t->bf = EH;
                *cresceu = 0;
                break;
            case EH:
                t->bf = RH;
                *cresceu = 1;
                break;
            case RH:
                t = balanceRight(t);
                *cresceu = 0;
        }
    return t;
}
```

```
Tree balanceRight(Tree t) {
    Tree aux;

    if (t->right->bf==RH) {
        // Rotacao simples a esquerda
        t = rotateLeft(t);
        t->bf = EH;
        t->left->bf = EH;
    }
    else {
        //Dupla rotacao
        t->right = rotateRight(t->right);
        t = rotateLeft(t);
    }
}
```

```
switch (t->bf) {
    case EH:
        t->left->bf = EH;
        t->right->bf = EH;
        break;
    case LH:
        t->left->bf = EH;
        t->right->bf = RH;
        break;
    case RH:
        t->left->bf = LH;
        t->right->bf = EH;
    }
    t->bf = EH;
}
return t;
}
```

■ Heaps ■

Uma *heap* é uma estrutura de dados **lógica**; corresponde a uma árvore binária:

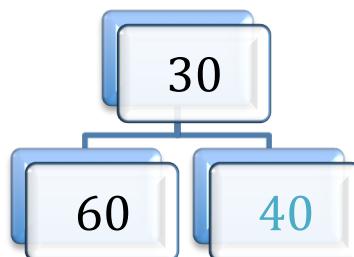
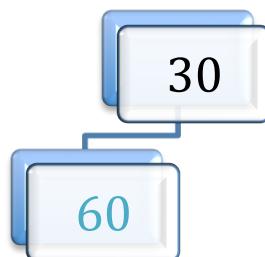
- **completa** (apenas o último nível pode não estar totalmente preenchido);
- cujo último nível é preenchido da esquerda para a direita, sem ‘lacunas’;
- com o seguinte **invariante de ordem**:
Min-heap: *o valor associado a cada nó é inferior ou igual aos valores de todos os seus descendentes*
Max-heap: *o valor associado a cada nó é superior ou igual aos valores de todos os seus descendentes.*

A altura é necessariamente logarítmica no número de nós, logo as operações de *inserção* e de *extracção de elementos* são executadas em tempo $O(\log n)$.

A extracção devolve sempre o menor (resp. maior) elemento na *heap*, pelo que esta estrutura é adequada para a implementação de filas com prioridades.

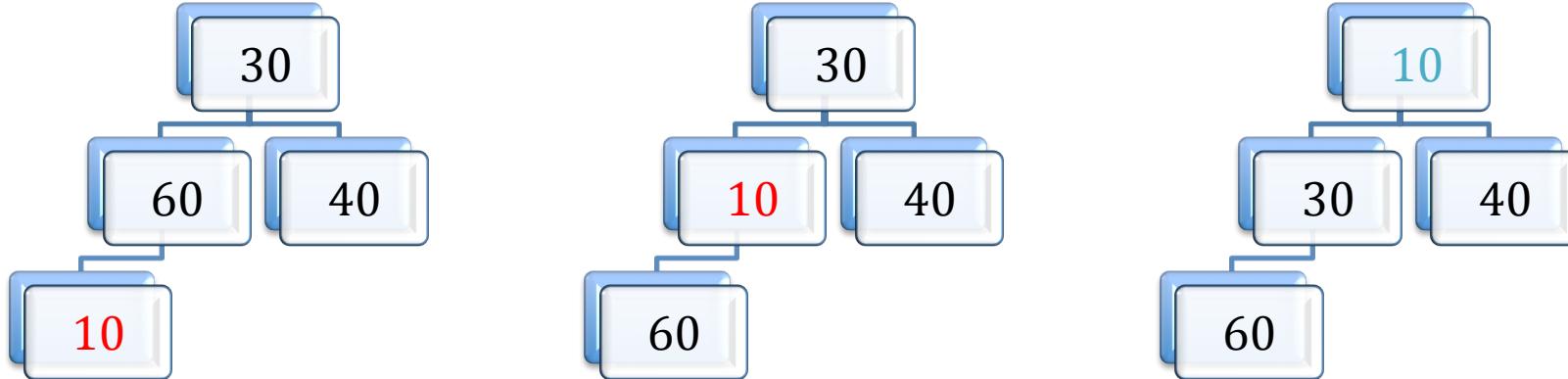
■ Exemplo: Sequência de Inserções ■

Insert 30; Insert 60; Insert 40;



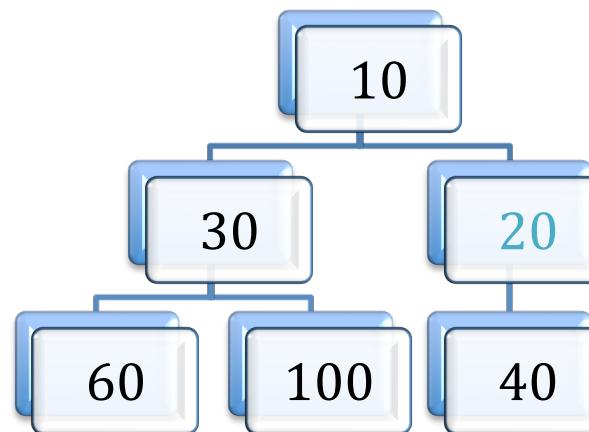
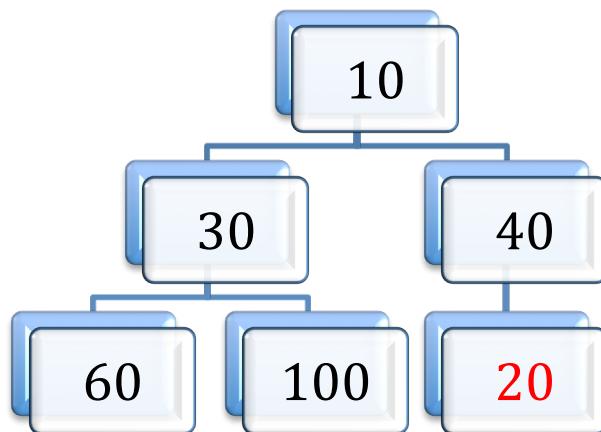
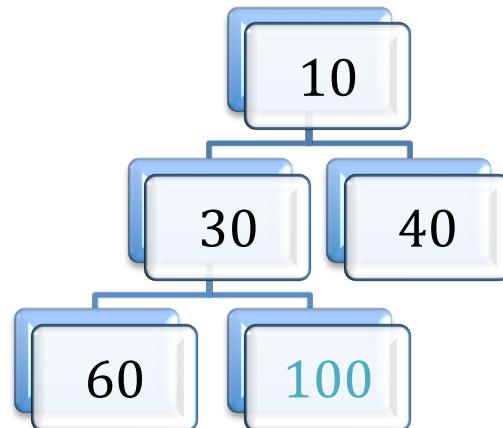
- Se o elemento inserido for de valor *não inferior* ao seu *pai*, a árvore resultante é uma *heap*.
- O algoritmo de inserção numa *heap* executa pois em tempo $\Omega(1)$.

Insert 10

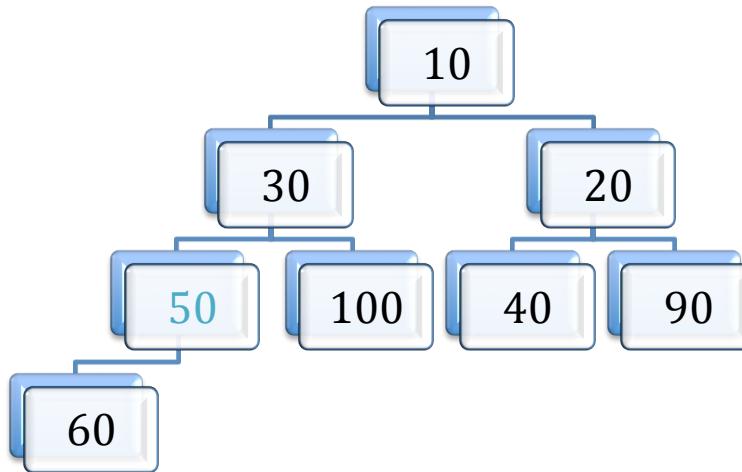
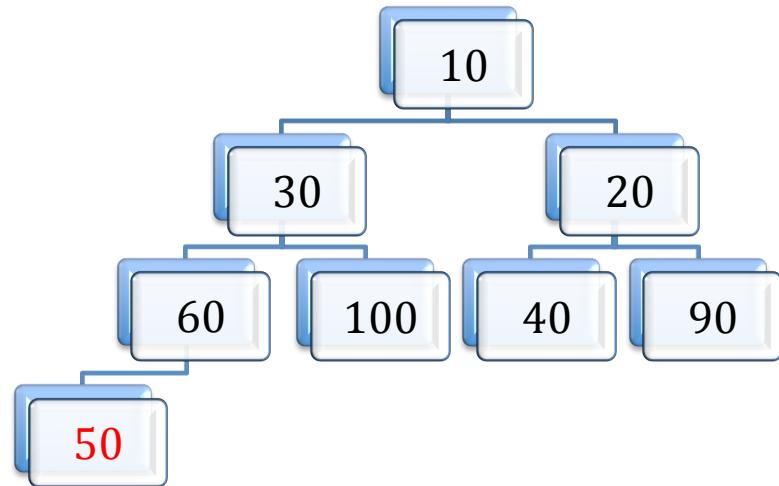
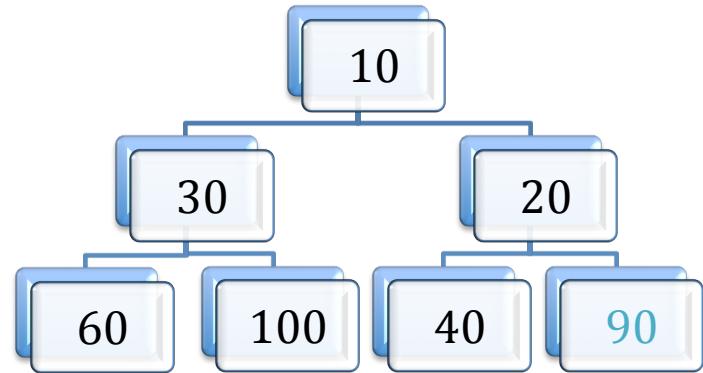


- Se o elemento inserido for de valor inferior ao seu *pai*, os dois trocam de posição.
- A árvore com raíz no elemento que acaba de subir é necessariamente uma *heap*.
- *Mas*, este elemento pode ainda ser de valor inferior ao seu novo pai, pelo que este passo pode ter que ser iterado.
- *No máximo*, será executado um número de trocas igual à altura da árvore.
- O algoritmo de inserção numa *heap* executa pois em tempo $\Omega(1)$, $O(\lg n)$.

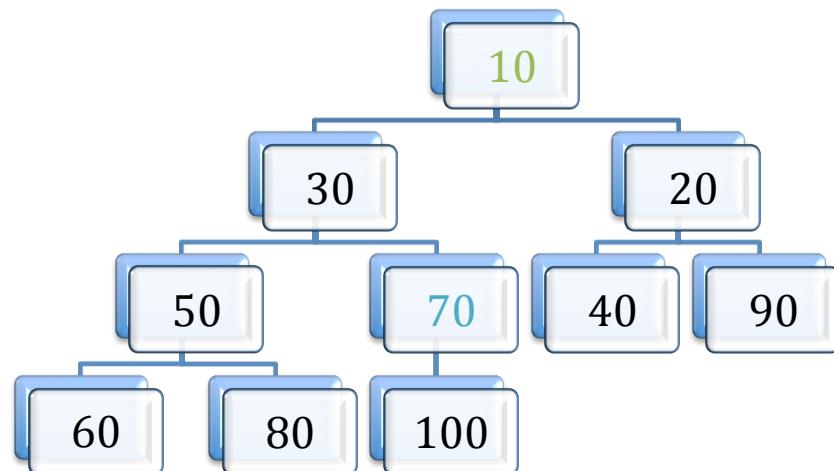
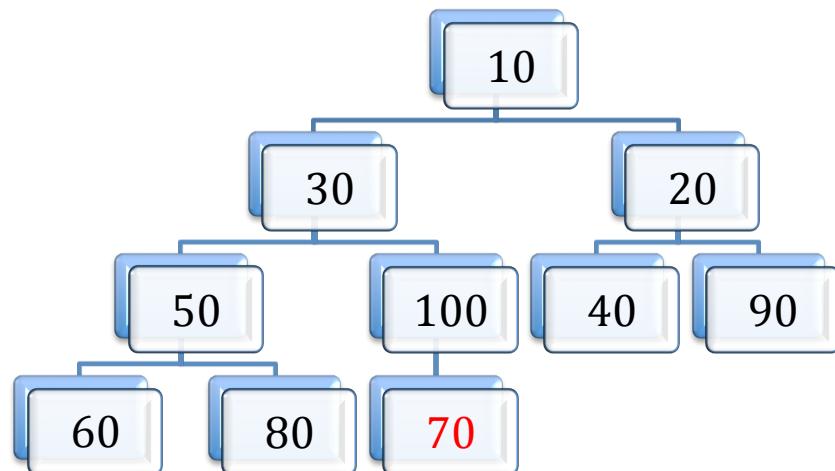
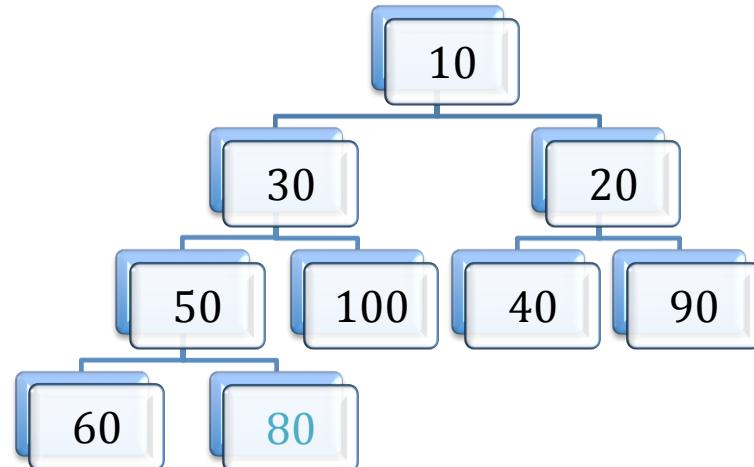
Insert 100, Insert 20



Insert 90, Insert 50

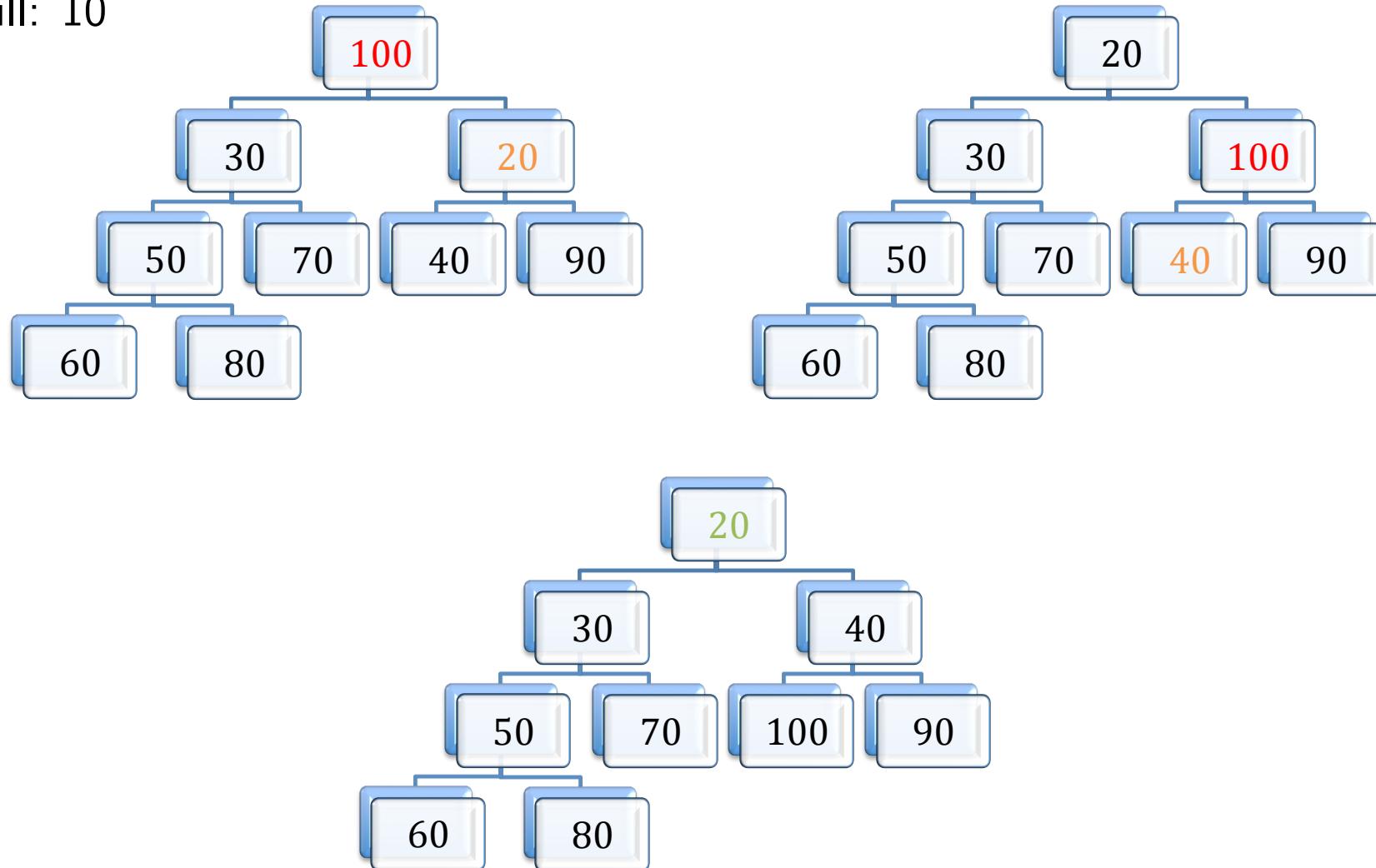


Insert 80, Insert 70

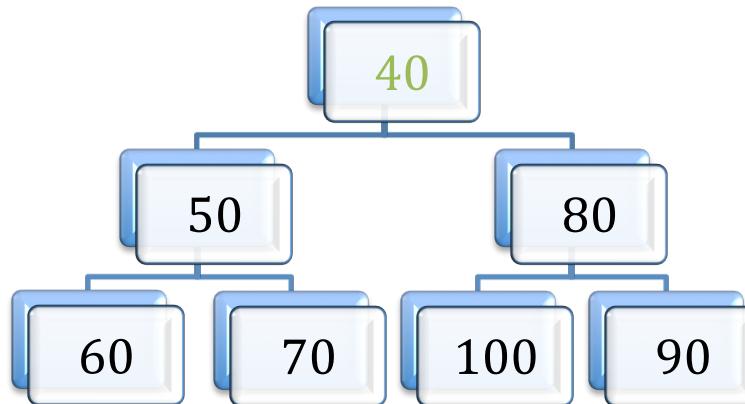
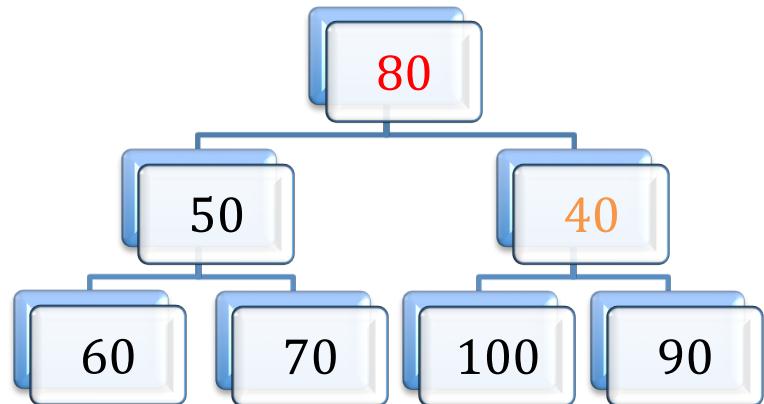
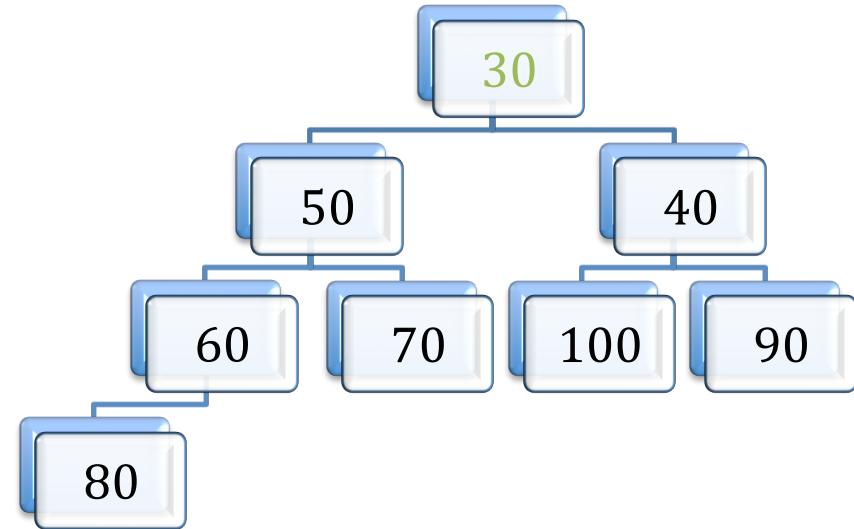
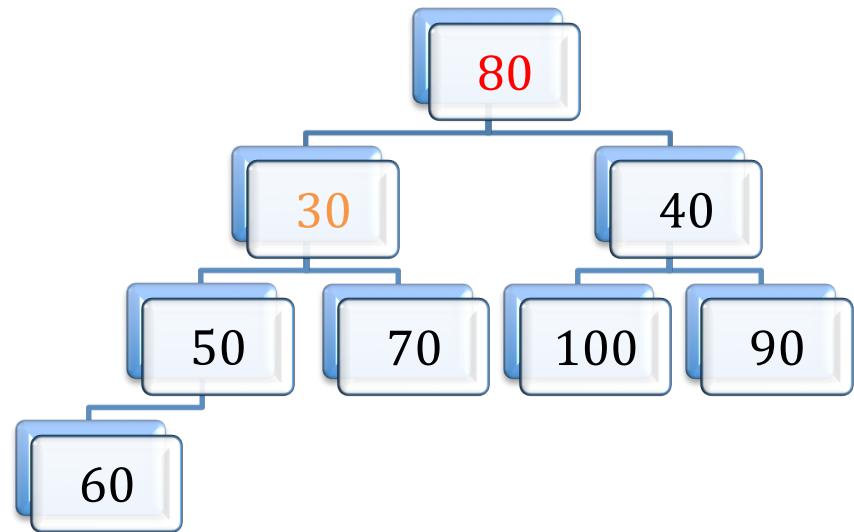


■ Exemplo: Sequência de Extracções ■

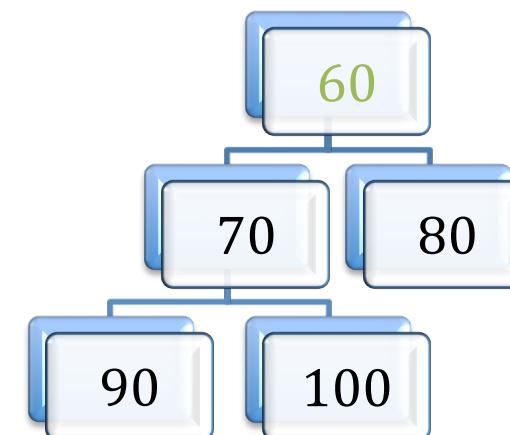
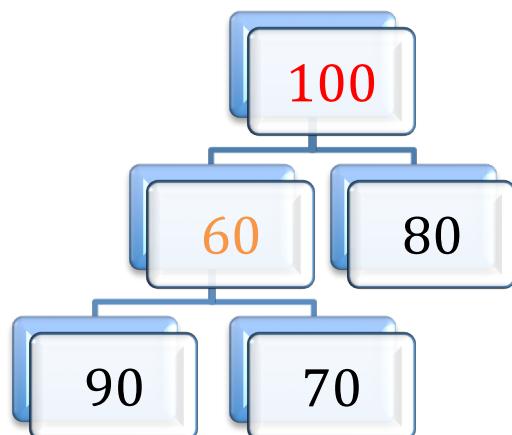
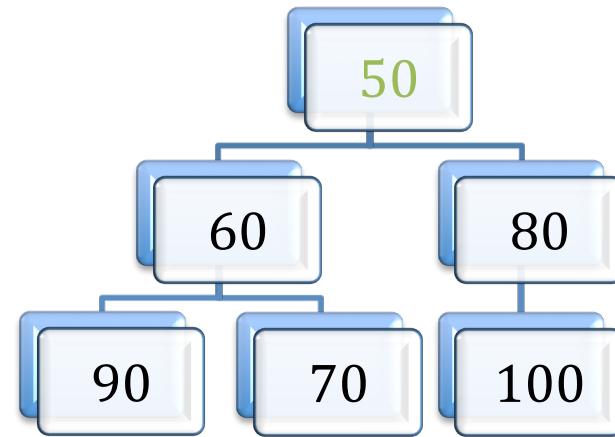
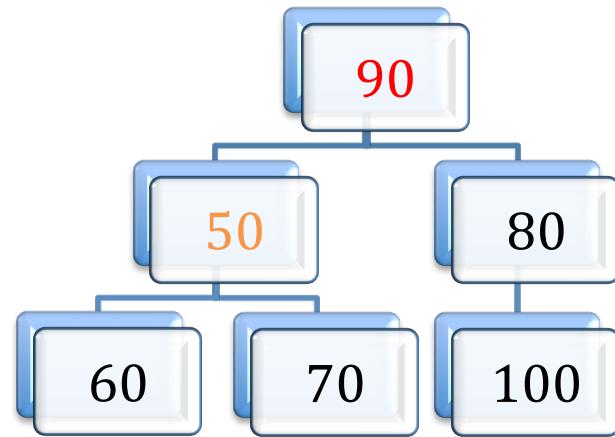
Pull: 10



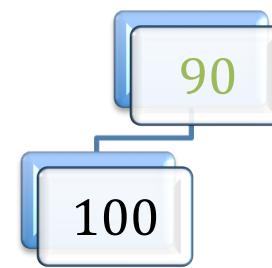
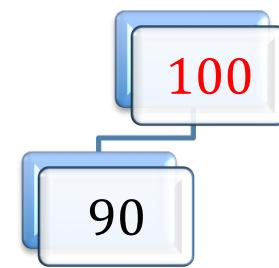
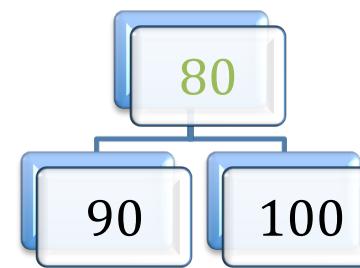
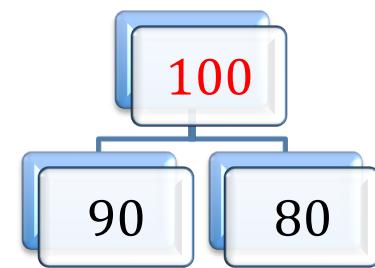
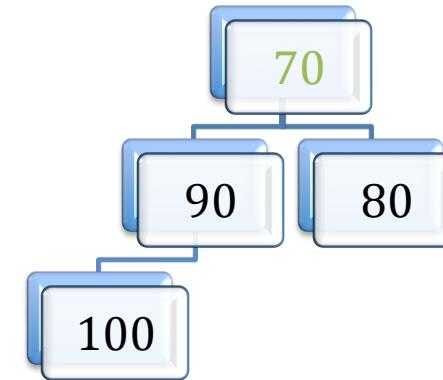
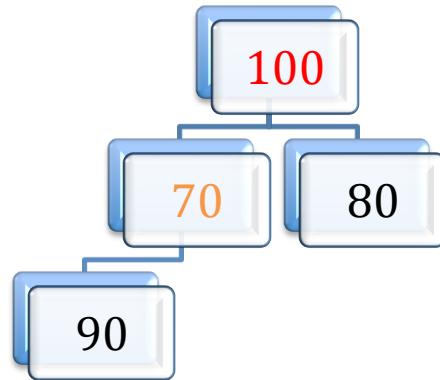
Pull: 20; Pull: 30



Pull: 40; Pull: 50



Pull: 60; Pull: 70; Pull: 80; . . .



■ Heaps: Implementação Física ■

A implementação típica é *contígua*, sobre um *array*, em que os elementos são dispostos *por níveis*, da esquerda para a direita:

i	0	1	2	3	4	5	6	7	8	9
$v[i]$	10	30	20	50	70	40	90	60	80	100
nível	1	2	2	3	3	3	3	4	4	4

⇒ Como é feito o acesso aos filhos / pai de um nó?

O exemplo anterior sugere um *algoritmo de ordenação* de tempo $\Theta(n \log n)$. Trata-se de facto de um algoritmo baseado em comparações, incremental, que atinge o tempo de pior caso óptimo $n \log n$ sem utilizar bi-recursividade.

⇒ Poderá *heapsort* ser implementado *in-place*?