ED2: Filas com Prioridades e “Heaps”

[Projecto Codeboard de suporte a esta aula: https://codeboard.io/projects/10165]

O tipo abstracto de dados

Recorde-se que os tipos abstractos de dados (*Abstract Data Types*, 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.

Uma *fila com prioridades (priority queue)* é um destes ADTs. Em particular, trata-se de uma estrutura do género *Buffer*, uma vez que dispobilbilza uma operação de **inserção** e outra de **extracção** de elementos, sendo a relação entre as duas operações regida por uma estratégia específica.

No caso das filas com prioridades a estratégia é mais complexa do que as bem conhecidas estratégias *Last-In, First-Out* (LIFO) *e First-In, First-Out* (FIFO) características dos buffers mais comuns, as **pilhas** e as **filas de espera**.

Numa fila com prioridades são associados valores numéricos aos elementos inseridos, que correspondem a valores de prioridade.

Arbitremos que as prioridades são dadas por números inteiros, correspondendo números pequenos a prioridades mais elevadas. Consideremos a seguinte sequência de inserções:

```
insert("AA", 10);
insert("BB", 20);
insert("CC", 5);
insert("DD", 15);
```

Se esta sequência for seguida de uma sequência de extracções (operação `pull`), os elementos serão extraídos pela seguinte ordem:

```
"CC"
"AA"
```

```
"DD"
"BB"
```

Tratando-se de uma estrutura de dados definida a um nível abstracto, será necessário conceber uma implementação concreta.

Heaps

Uma *heap* é uma árvore binária, caracterizada por duas propriedades (invariantes de tipo):

- Invariante de **ordem**:
  O valor associado a cada nó é *inferior ou igua*l aos valores de todos os seus descendentes

- Invariante de **forma:**
  - A árvore binária é *completa* (apenas o último nível pode não estar totalmente preenchido), e
  - último nível é preenchido da esquerda para a direita, *sem "lacunas"*

Estas propriedades de forma implicam que a altura é necessariamente *logarítmica* no número de nós da árvore, logo as operações de inserção e de extracção de elementos podem ambas ser executadas em tempo $\mathrm{O}(\log n)$.

Note-se que este invariante de ordem implica que o mínimo da estrutura se encontra na raiz da árvore, e por isso uma *heap* com esta propriedade designa-se por *min-heap.*
Substituindo
*inferior ou igua*l por sup*erior ou igua*l obtém-se uma *max-heap.*

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

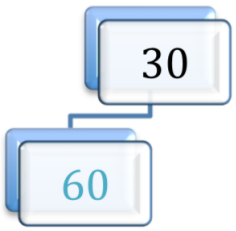
Algoritmo de Inserção:

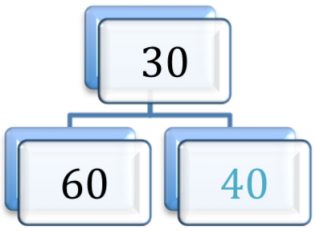
1. Insere-se o novo elemento na primeira posição livre da heap, i.e. na posição mais à esquerda do último nível da heap;

2. Faz-se uma operação de **bubble-up:**
   Enquanto o elemento inserido for de valor inferior o seu pai na árvore, troca-se sucessivamente (ao longo de um caminho ascendente da *heap*) estes dois elementos.

EXEMPLO
Consideremos uma sequência de inserções numa *min-heap*, começando com uma estrutura vazia.

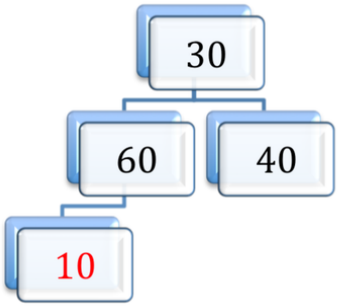
```
Insert 30;
Insert 60;
Insert 40;
```

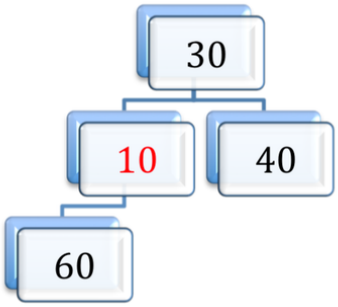


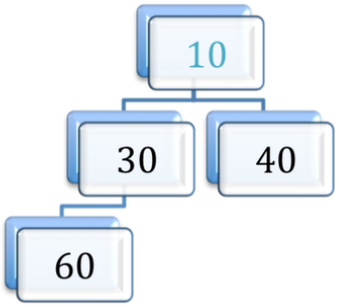


Nestes primeiros passos o invariante de ordem foi respeitado, pelo que não foi necessário executar *bubble-up.* No próximo passo isso já não será assim.

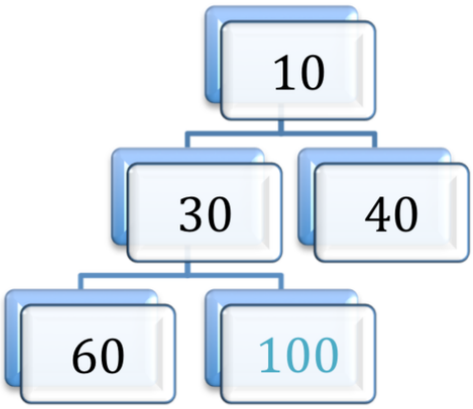
```
Insert 10;
```

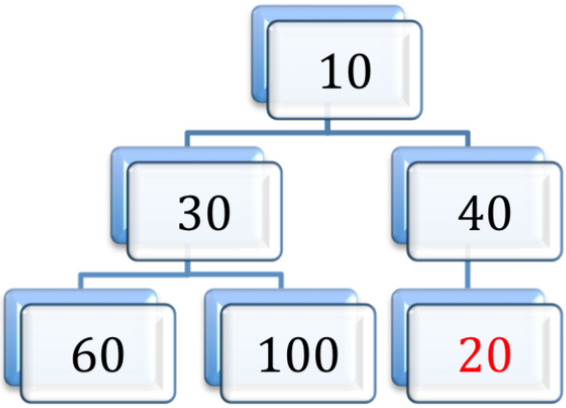


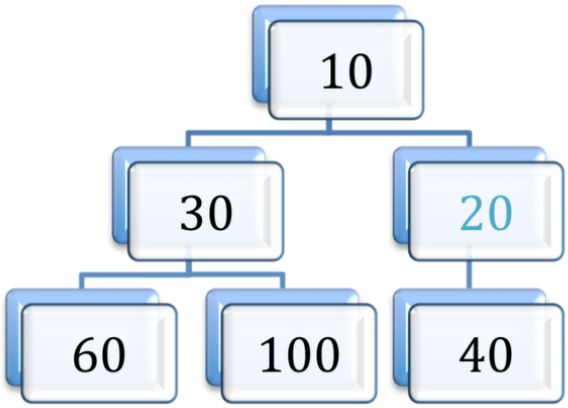





```
Insert 100;
Insert 20;
```

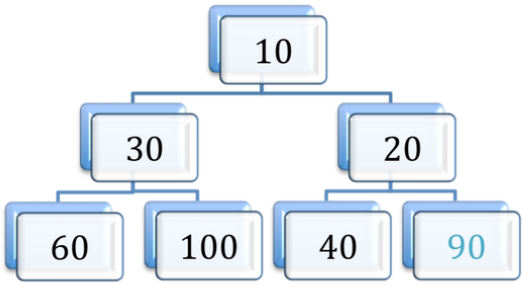


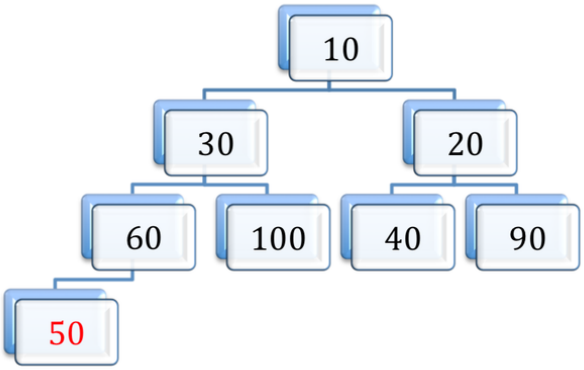


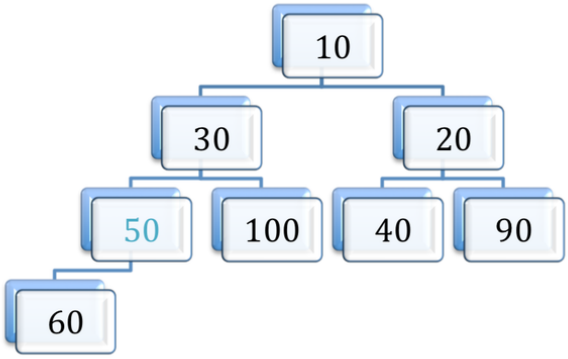



```
Insert 90;
```

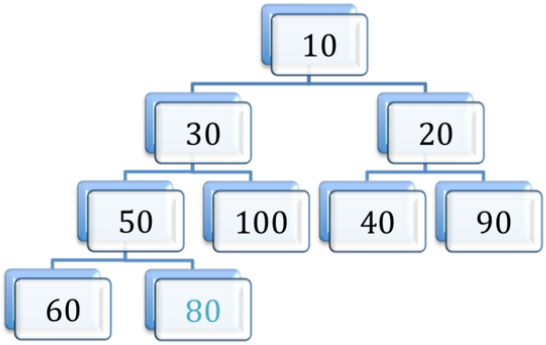
```
Insert 50;
```

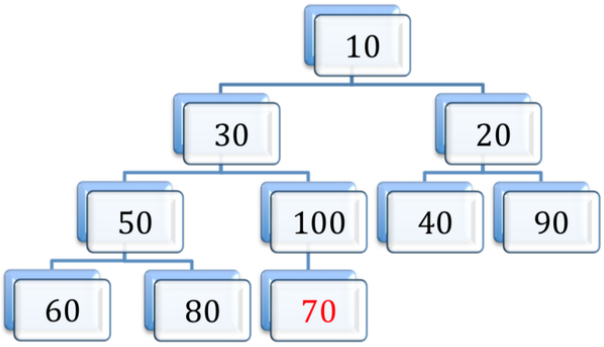


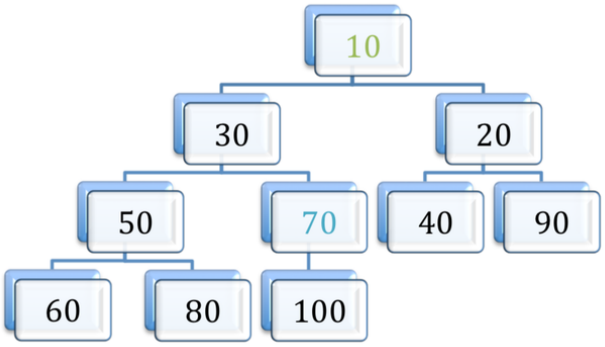





```
Insert 80;
Insert 70;
```







Observe-se que:

> *Se for possível o acesso directo ao pai de cada elemento da* heap*, o algoritmo de inserção (incluindo a op. de* bubble-up*) executará em tempo* $\mathrm{O}(\log N)$*, uma vez que a altura da árvore é logarítmica em* N

Algoritmo de Extracção (operação *pull*):

Naturalmente, o elemento a extrair será sempre a raíz da árvore (quer se trate de uma *min-heap* quer se trate de uma *max-heap*). A questão que se coloca é como reajustar a estrutura para eliminar a lacuna gerada na raíz, respeitando ainda todos os invariantes.

A intuição aponta no sentido de fazer subir o menor dos filhos da raíz, repetindo sucessivamente este passo. No entanto, é imediato constatar (por exemplo na *heap* construída acima) que este algoritmo não preserva os invariantes de forma de uma *heap*.

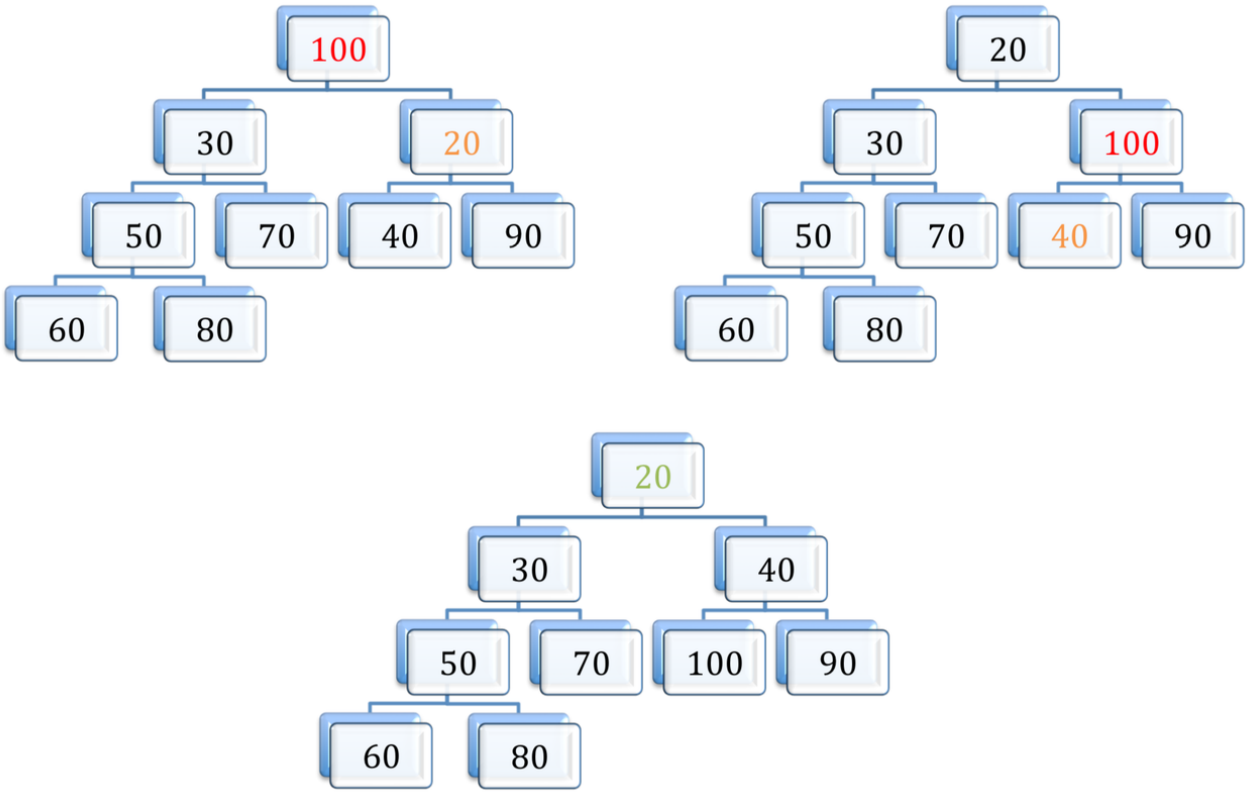
O algoritmo correcto é o seguinte:

1. Remove-se o elemento inserido na última posição da heap, i.e. na posição mais à direita do último nível da heap, e inscreve-se este mesmo elemento na raíz da *heap,* em substituição da raíz extraída.

2. Faz-se uma operação de **bubble-down** desta nova raíz**:**
   Enquanto o nó actual for de valor superior a pelo menos um dos seus filhos, troca-se sucessivamente (ao longo de um caminho descendente da *heap*) o valor do nó com o do menor dos seus filhos

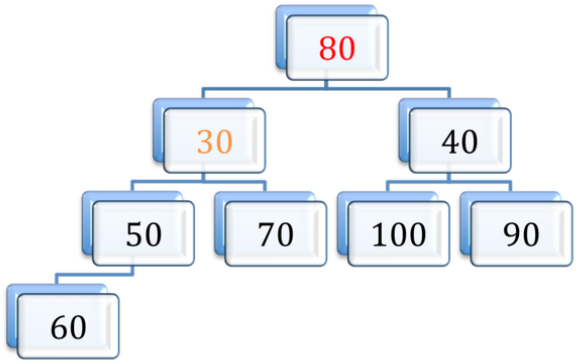
EXEMPLO
Executemos uma sequência de extracções a partir da *heap* do exemplo anterior.

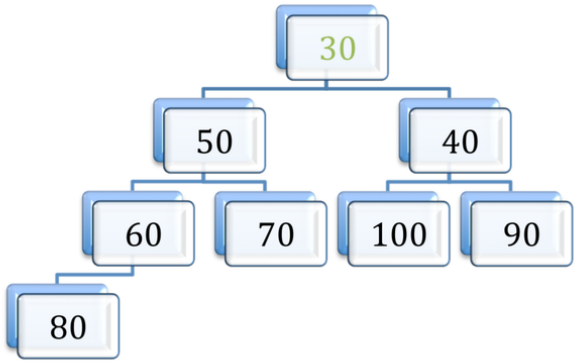
```
Pull;
> 10
```

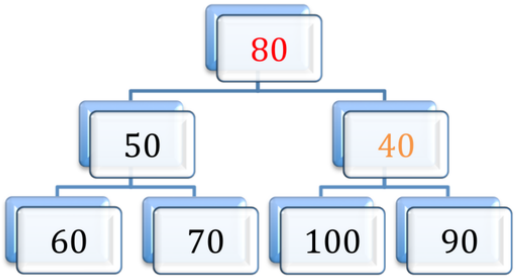


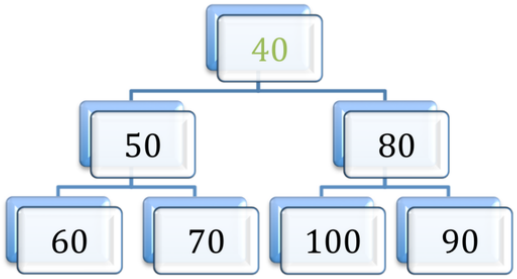

```
Pull;
> 20
```

```
Pull;
> 30
```

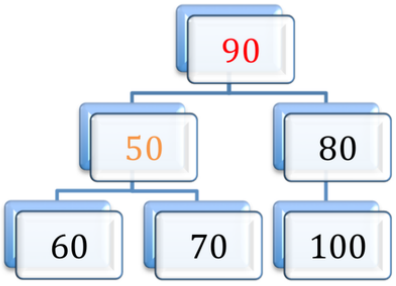


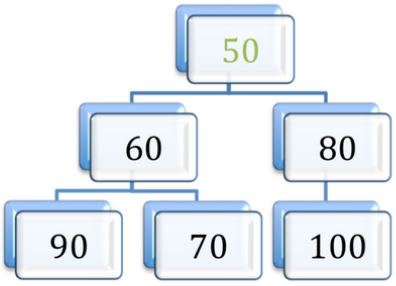


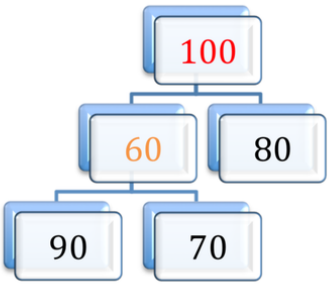


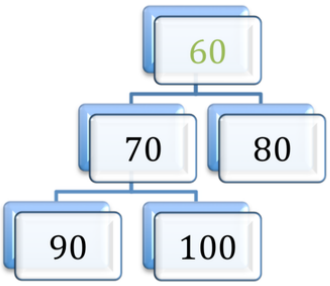



```
Pull;
> 40
Pull;
> 50
```

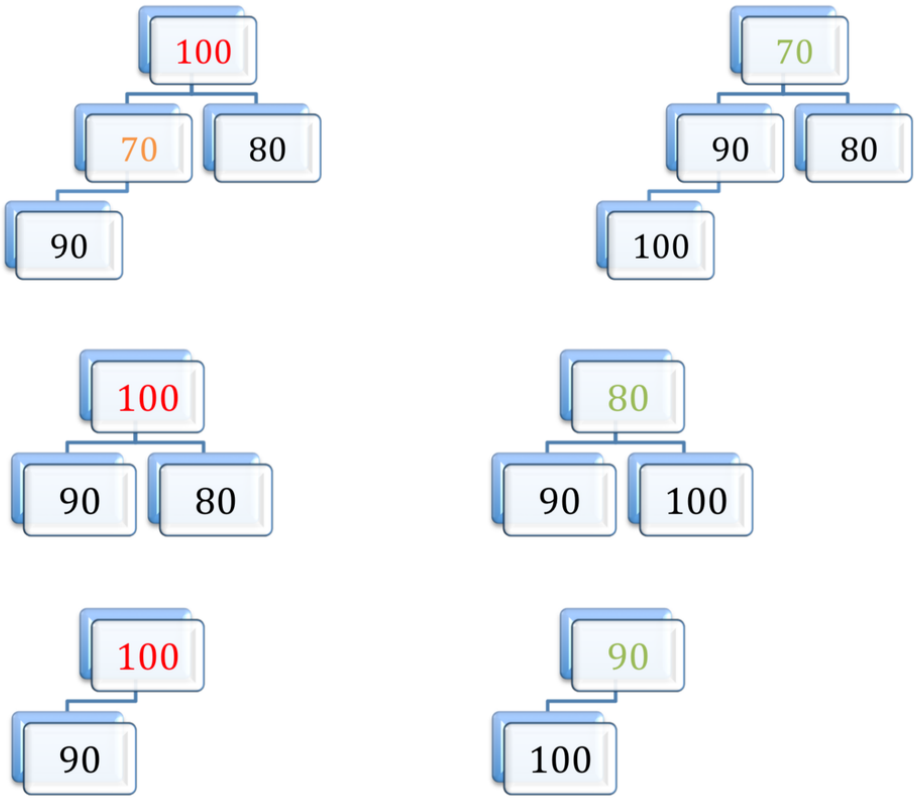









```
Pull;
> 60
Pull;
> 70
Pull;
> 80
Pull;
> 90
```



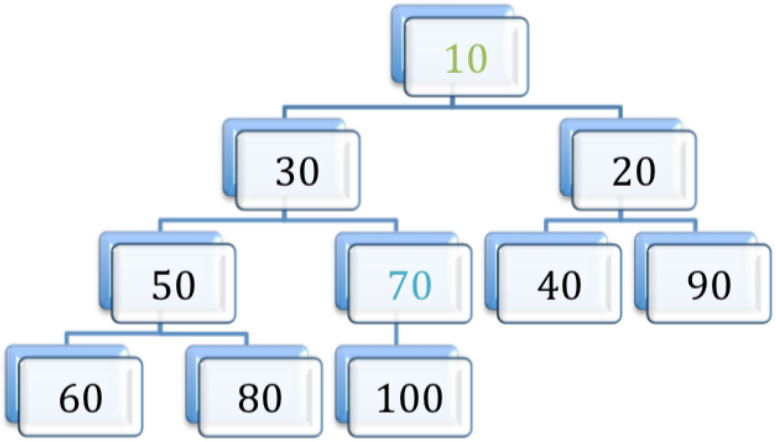
Heaps: Implementação Física

Tal como descrita acima, uma *heap* é uma estrutura de dados ao nível *lógico.*

Ao contrário do que acontece com uma árvore binaria de pesquisa, que é tipicamente implementada por uma estrutura física ligada em memória dinâmica, as *heap* são tipicamente implementadas sobre *arrays* (podendo ser alocadas estática ou dinamicamente).

Basta dispor os elementos por ordem da raíz de árvore para as folhas, e percorrendo os níveis da esquerda para a direita

EXEMPLO
A *heap:*



pode ser implementada ao nível físico pelo seguinte vector:

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

Observe-se que a implementação sobre um *array* permite o acesso directo (em tempo constante) não só aos filhos de um determinado nó, como também ao seu pai. Além disso, possibilita também o acesso em tempo constante ao último elemento da *heap*, o que é relevante para a execução dos algoritmos vistos atrás.

Uma consequência deste facto é que no melhor caso os algoritmos executam em tempo constante, o que não seria possível numa implementação ligada típica em que seria necessário localizar o último nó.

> *Os algoritmos de inserção e extracção numa* heap *executam em tempo* $\Omega(1), \mathrm{O}(\log N)$.

EXERCÍCIOS
[a resolver em https://codeboard.io/projects/10165]

Para a implementação de uma *min-heap* sobre um *array* dinâmico consideraremos as seguintes definições de tipos e protótipos de funções, em que `used` é o tamanho actual da *heap,* e `size` é a sua capacidade máxima (correspondente ao comprimento do *array* alocado).

```
typedef int Elem;  // elementos da heap.
```

```
typedef struct {
  int   size;
  int   used;
  Elem  *value;
} Heap;

void initHeap (Heap *h, int size);
int insertHeap (Heap *h, Elem x);
int extractMin (Heap *h, Elem *x);
int minHeapOK (Heap h);
```

Implemente as 4 funções com os protótipos dados, notando o seguinte:

- A função `initHeap` inicializa uma *heap* (passada por referência), alocando para isso um *array* de comprimento `size`
- Se preferir, poderá começar por implementar a *heap* sobre um *array* estático
- Na implementação dinâmica, o comprimento do *array* deverá ser *duplicado* quando a capacidade se encontra completamente preenchida, por forma a assegurar que, em termos amortizados esta operação executa em tempo $\Omega(1), \mathrm{O}(\log N)$
- Os valores de retorno podem ser utilizados para um código de erro

O projecto Codeboard inclui uma função `main` que executa a sequência de inserções e extracções exemplificada acima.

ED3. Árvores AVL

PROJECTO CODEBOARD DE SUPORTE A ESTE MÓDULO:
https://codeboard.io/projects/46752

Tipos Abstractos de Dados e Estruturas de Dados (revisão de conceitos)

Os tipos abstractos de dados (*Abstract Data Types*, 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.

Os tipos de dados abstractos são implementados com base em **estruturas de dados concretos,** como sejam por exemplo as *sequências*, as *árvores*, ou os *grafos,*
a que está já associada uma forma ou organização interna 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 procura** (*Binary Search Tree*, BST) é um caso particular de árvore binária (com um invariante que estabelece uma relação de ordem *inorder*).

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 ela sobre um *array*, ou então como uma *lista ligada*, alocada dinamicamente, com utilização de apontadores.

O ADT Dicionário / Array Associativo / ou Mapeamento

Armazena pares chave → valor, tendo a semântica de uma função finita. As operações básicas são

1. A **inserção** de um par chave → 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.

Se as chaves forem de *um tipo que admita uma noção de ordem*, um dicionário pode ser implementado por uma árvore binária de procura.

Árvores Binárias de Procura

Uma *árvore binária de procura* (“binary search tree”, BST) é uma estrutura de dados que pode ser utilizada para implementar *dicionários* ou simplesmente (multi-)conjuntos, e cujas operações se caracterizam por um comportamento

largamente dependente da *forma* da árvore.

Assim, temos os dois seguintes casos extremos:

- Uma árvore *totalmente desequilibrada* assume a forma de uma lista de elementos (cada nó tem sempre um descendente vazio); a operação de procura pode executar no melhor caso em tempo constante, e no pior caso em tempo linear: $T(N) = \Omega(1), \mathcal{O}(N)$. O mesmo sucede com a operação de inserção.

- Já numa árvore *equilibrada*, os elementos inseridos ocupam um número de níveis próximo do mínimo possível, e o pior caso da procura passa para logarítmico: $T(N) = \Omega(1), \mathcal{O}(\log N)$. As inserções executam *todas* em tempo logarítmico, $T(N) = \Theta(\log N)$.

É claramente desejável trabalhar com árvores equilibradas. Num cenário em que ocorram muito mais operações de procura do que inserções, uma solução possível é reequilibrar a árvore periodicamente (por exemplo, após cada 100 inserções). Mas no caso geral será preferível manter a árvore permanentemente equilibrada.

Árvores AVL

Uma árvore AVL (Adelson-Velskii & E.M. Landis) é uma árvore binária de procura em que todos os nós satisfazem adicionalmente o seguinte invariante estrutural:

> *As alturas da sub-árvore da esquerda e da sub-árvore da direita diferem no máximo numa unidade:* $|h_e - h_d| \leq 1$.

```
    10

10
  \
  20

```

```
      10
    /    \
   5     20
           \
            30

      10
    /    \
   5     20
 /      /   \
3      15    30
           /
          25
```

Note-se que este invariante admite árvores que não são completas, i.e. é possível que um nível da árvore contenha elementos sem que o nível anterior esteja completamente preenchido.

No entanto, tal como nas árvores completas, a altura de uma árvore AVL é assimptoticamente logarítmica, o que garante que a operação de procura executa em tempo $T(N) = \mathcal{O}(\log N)$. Mas o que é mais interessante ainda é o seguinte:

> *O tempo de execução das operações de inserção e remoção,* modificadas por forma a efectuarem o necessário ajuste das árvores para preservar o invariante AVL*, é também* $T(N) = \mathcal{O}(\log N)$*.*

Veremos em seguida como modificar o algoritmo tradicional de inserção numa árvore binária de procura por forma a lidar com árvores AVL.

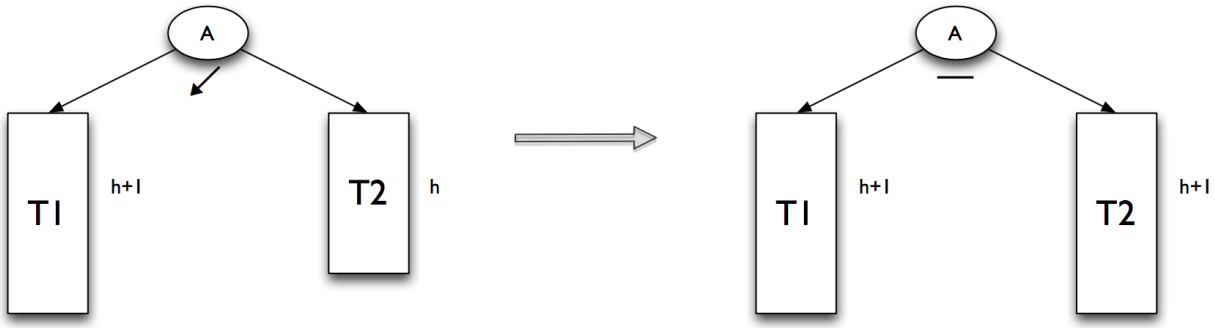
Algoritmo de inserção numa árvore AVL

Em algumas situações, a inserção de um novo elemento numa árvore AVL preservará o invariante estrutural ($|h_e - h_d| \leq 1$) em todos os nodos. Consideremos agora em detalhe o que poderá suceder quando se faz, recursivamente, uma inserção *à direita da raíz* (naturalmente, o outro caso é simétrico).

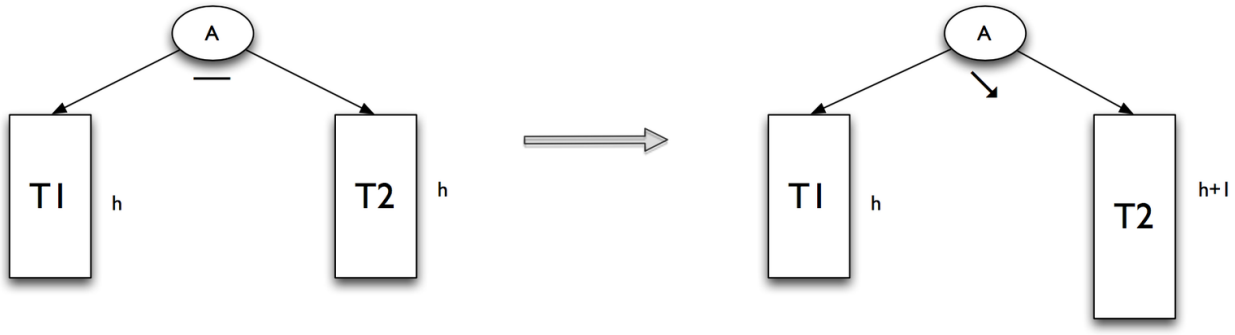
Caso 0: inserção à direita não provoca aumento da altura da sub-árvore da direita.
Neste caso não há nada a fazer, a relação entre h_e e h_d mantém-se.

Nos restantes casos haverá aumento da altura da árvore da direita.

Caso 1: $h_e = h_d + 1$
A sub-árvore da esquerda é mais pesada, e neste caso passaremos a ter, depois da inserção à direita, $h_e = h_d$.



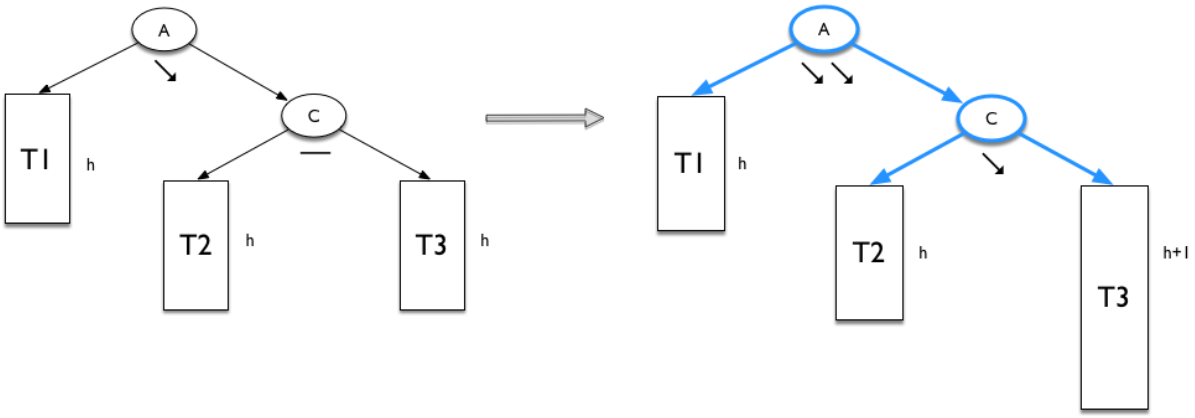
Caso 2: $h_e = h_d$
As duas sub-árvores têm à partida a mesma altura. Neste caso passaremos a ter, depois da inserção à direita, $h_d = h_e + 1$. A raíz (A) continua a satisfazer o invariante.



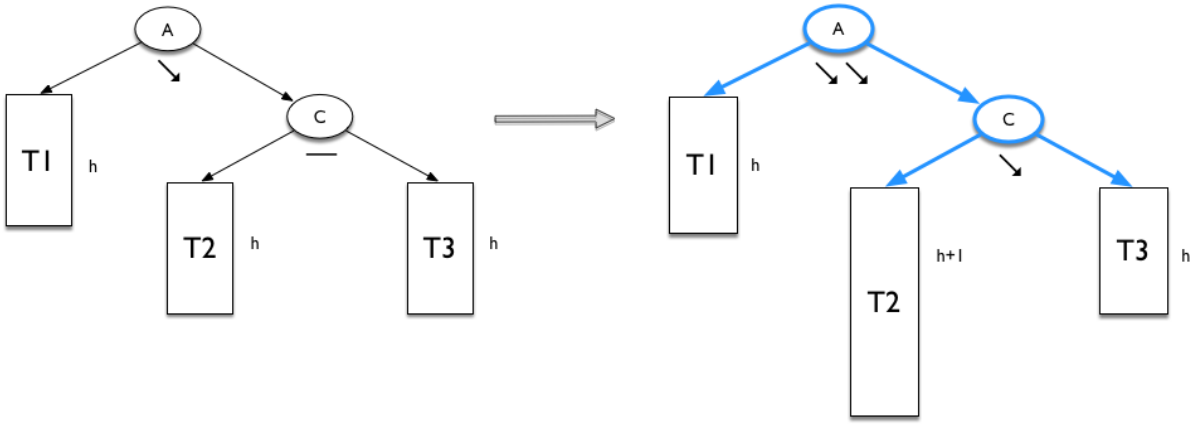
Já se adivinha que o caso problemático, que levará à necessidade de reajustar a árvore, ocorre quando a sub-árvore da direita já é à partida a mais pesada:

Caso 3: $h_d = h_e + 1$.
Note-se que neste caso a sub-árvore da direita tem necessariamente duas sub-árvores com a mesma altura h. Chamemos-lhes T2 e T3.
Então, podem agora surgir dois casos diferentes, consoante o aumento de altura ocorra em T2 ou em T3. Ambos os casos levam à violação do invariante na raíz (A), com $h_d = h_e + 2$.

Caso 3a: a inserção produziu um aumento da altura de T3

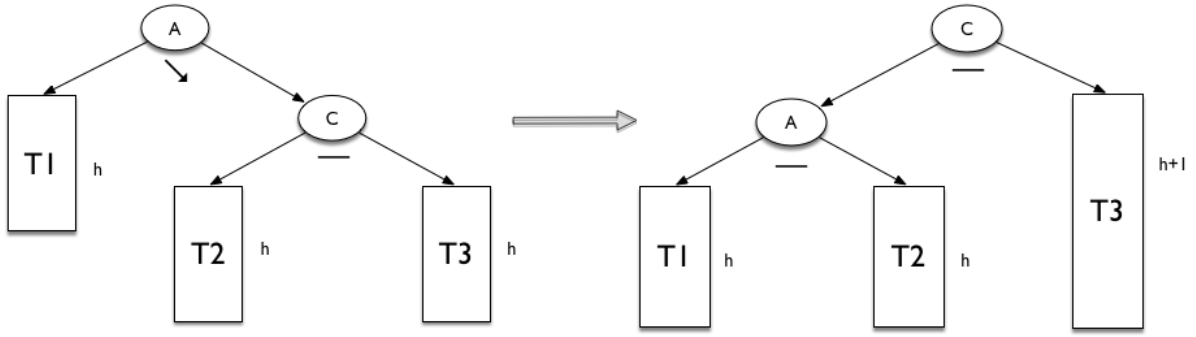


Caso 3b: a inserção produziu um aumento da altura de T2



Vejamos como será reposto o invariante em cada caso, ajustando-se a estrutura da árvore.

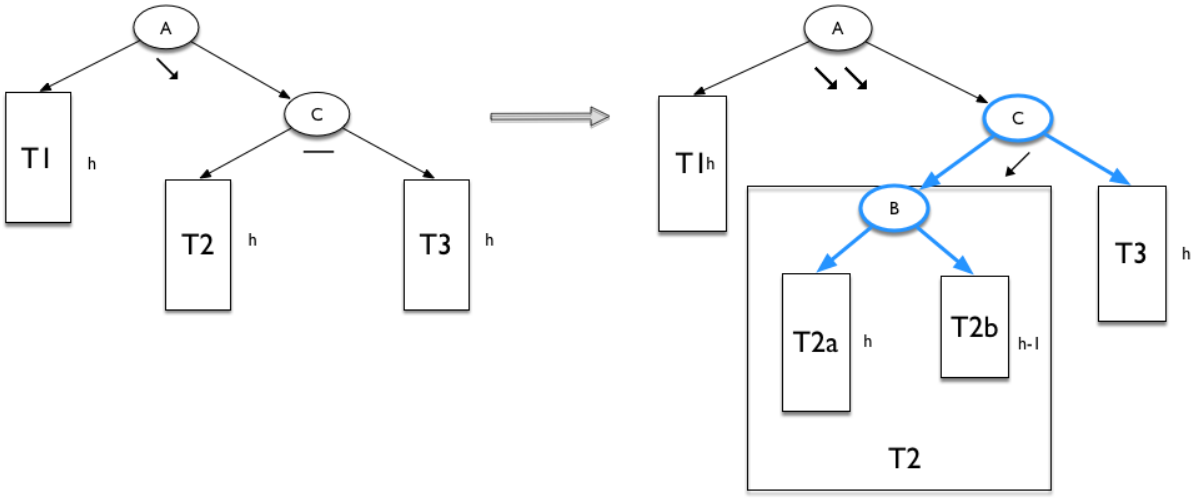
Comecemos pelo caso **Caso 3a.** Neste caso procede-se a uma operação de *rotação* à esquerda da árvore: o nó C tomará o papel da raíz, descendo o nó A para a esquerda de C. A árvore T2 passará a estar à direita de A.



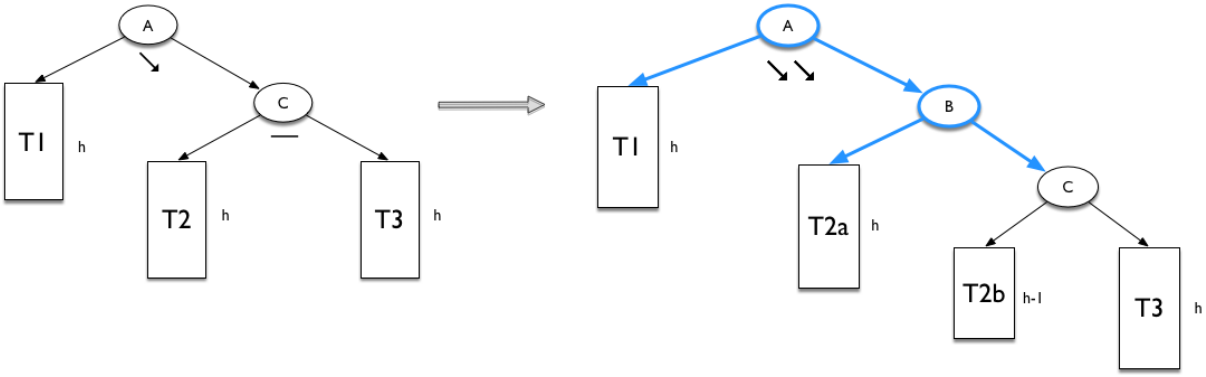
A figura acima mostra que as sub-árvores de A têm a mesma altura, e o mesmo acontece com as de C. Sendo assim, o invariante $|h_e - h_d| \leq 1$ é satisfeito em ambos os nós.

EXERCÍCIO: Mostre que o resultado da rotação continua a ser uma árvore binária de procura, i.e. que a ordem relativa dos elementos é preservada.

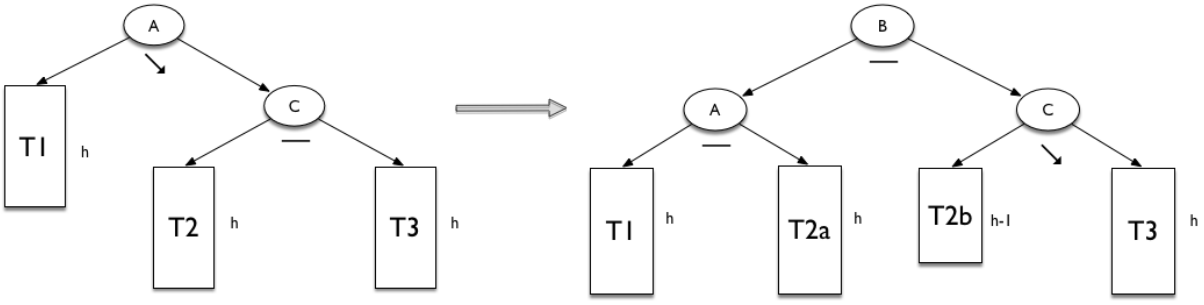
Caso 3b: a inserção produziu um aumento da altura de T2. Necessitamos neste caso de observar a estrutura de T2, com raíz B e sub-árvores T2a e T2b:



Uma simples rotação à esquerda não resolveria neste caso o problema. Antes disso é necessário efectuar uma *rotação à direita, com eixo no nó C. B sobe para o lugar de C, que desce para a sua direita:*



Em seguida faz-se a rotação à esquerda com eixo na raíz A, como no Caso 3A:



Mais uma vez as sub-árvores da nova raíz B têm a mesma altura, e o mesmo acontece com as de C. Note-se que este reajustamento repõe o invariante das

árvores AVL, independentemente dos pesos relativos de T2a e T2b.

EXERCÍCIO: Represente graficamente a evolução de uma árvore AVL quando é efectuada a seguinte sequência de inserções: 10, 20, 30, 70, 40, 50. Não se esqueça de indicar os factores de balanceamento de cada nó.

```
    10,E


    10,R
        \
        20,E


    10,R
        \
        20,R
           \
          30,E


        20,E
      /      \
   10,E       30,E

```

```
        20,R
      /      \
   10,E       30,R
                \
                 70,E


        20,R
      /      \
   10,E       30,R
                \
                 70,L
                /
              40,E


        20,R
      /      \
   10,E       30,
                \
                 40,
                   \
                    70


        20,R
      /      \
   10,E       40,E
             /    \
          30,E    70,E
```

```
        20,R
      /      \
   10,E       40,R
             /    \
          30,E    70,L
                 /
              50,E


           40,E
         /      \
       20,E       70,L
      /   \       /
    10,E  30,E  50,E
```

Implementação em C do Algoritmo de Inserção

Tipos de dados

Consideremos uma árvore binária de números inteiros. A estrutura habitual dos nós de uma árvore binária comum será aumentada com um campo correspondente ao *estado de balanceamento* do nó, que poderá ser um de três valores:

- LH (sub-árvore da esquerda tem maior altura),
- RH (sub-árvore da direitura tem maior altura), ou
- EH (alturas iguais).

```
typedef int TreeEntry;
typedef enum balancefactor { LH , EH , RH } BalanceFactor;

struct treenode {
    BalanceFactor bf;
    TreeEntry entry;
    struct treeNode *left;
    struct treeNode *right;
};

typedef struct treenode *Tree;
```

Funções auxiliares

Vimos acima que o ajuste da estrutura das árvores tem por base operações simples de *rotação.*
A seguinte função realiza uma rotação simples à esquerda da árvore com raíz apontada por t, devolvendo o endereço da nova raíz. Note que a função não pode ser executada se a árvore da direita for vazia!

```
// requires:
// (t != NULL) && (t->right != NULL)
//
Tree rotateLeft(Tree t)
{
    Tree aux = t->right;
    t->right = aux->left;
    aux->left = t;
    t = aux;
```

```
    return t;
}
```

EXERCÍCIO: Implemente a função de rotação à direita.

Escreveremos em seguida uma função que, recebendo uma árvore *cuja raíz deixou de satisfazer o invariante AVL* e que está *desequilibrada para a direita* (i.e. $h_d = h_e + 2$), corrige a estrutura da árvore.

Para isso, a função:

- começa por determinar se o caso exige uma rotação simples ou dupla, examinando o indicador de balanceamento do nó à direita;
- no segundo caso (rotação dupla), e tendo em conta o que foi dito anteriormente, é necessário ajustar os indicadores de balanceamento dos nós à esquerda (A na figura) e à direita (C) da nova raíz da árvore (B), de acordo com o estado actual do indicador de balanceamento de B;
- finalmente ajusta-se o indicador da raíz.

Tal como a anterior, esta função não pode ser executada se a árvore da direita for vazia.

```
// requires:
// (t != NULL) && (t->right != NULL)
//
Tree balanceRight(Tree t)
{
    if (t->right->bf==RH) {
        // Rotacao simples, caso 3a
        t = rotateLeft(t);
        t->bf = EH;
        t->left->bf = EH;
    }
    else {
        // Dupla rotação, caso 3b
        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;
}
```

EXERCÍCIO: Implemente a função `balanceLeft`.

Função de inserção numa árvore binária de procura comum

O tipo esperado para uma função de inserção não deverá ser novidade: recebe um apontador para a raíz da árvore e o elemento a inserir, e devolve o endereço da raíz (que poderá ter sido alterado). O algoritmo recursivo é bem conhecido:

```
Tree insertTree(Tree t, TreeEntry e) {
    if (t==NULL){
        t = (Tree)malloc(sizeof(struct treenode));
        t->entry = e;
        t->right = t->left = NULL;
```

```
    }
    else if (e > t->entry)
        t->right = insertTree(t->right, e);
    else
        t->left = insertTree(t->left, e);
    return t;
}
```

+

Função de inserção numa árvore AVL

A função de inserção numa árvore AVL deve devolver, além do endereço da nova raíz da árvore, também *informação sobre se a altura da árvore cresceu ou não após esta inserção*. Para isso incluiremos no protótipo um parâmetro de tipo int passado por referência:

```
Tree insertTree(Tree t, TreeEntry e, int *cresceu);
```

A ideia é que depois da chamada `insertTree(t, e, &c)` com c de tipo `int`, c terá o valor 1 se a altura de t cresceu, e 0 em caso contrário.

O algoritmo de inserção numa árvore AVL segue a estrutura do anterior, com as seguintes diferenças:

- detecta os casos em que o invariante AVL é violado, e chama a função `balanceRight` ou `balanceLeft` para corrigir a estrutura da árvore
- reajusta os indicadores de balanceamento dos nós afectados pela inserção
- determina se a altura da árvore aumentou ou não, e atribui o valor adequado 0 ou 1 a `*cresceu`

EXERCÍCIO: Complete a definição da seguinte função de inserção:

```
Tree insertTree(Tree t, TreeEntry e, int *cresceu)
{
    if (t==NULL){
        t = (Tree)malloc(sizeof(struct treenode));
```

```
        t->entry = e;
        t->right = t->left = NULL;
        t->bf = EH;
        *cresceu = 1;
    }
    else if (e > t->entry) {
        t->right = insertTree(t->right, e, 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;
            }
        }
    }
    else {
        t->left = insertTree(t->left,e,cresceu);
        if (*cresceu) {
          ...
        }
    }
    return t;
}
```

Outros Exercícios

1. A seguinte função calcula a altura de uma qualquer árvore binária, em tempo $\Theta(N)$:

```
int nonAVL_treeHeight(Tree t) {
    int l, r;
    if (t==NULL) return 0;
    l = treeHeight(t->left);
    r = treeHeight(t->right);
    if (l>r) return l+1;
    else return r+1;
}
```

Redefina a função por forma a calcular a altura de uma árvore AVL em tempo $\Theta(\log N)$.

2. É possível testar se uma árvore binária é ou não AVL da seguinte forma:

```
int isAVL (Tree t) {
    int l, r;
    if (t == NULL) return 1;

    l = treeHeight (t->left);
    r = treeHeight (t->right);

    return (abs (l-r) <= 1 &&
            isAVL(t->left) &&
            isAVL(t->right));
}
```

   a. Analise o tempo de execução no pior caso desta função.

   b. É possível optimizar esta função alterando-a por forma a calcular simultaneamente a altura da árvore, dispensando assim a utilização da função treeHeight. Complete a seguinte definição e analise o seu tempo de

execução no pior caso:

```
// altura da árvore será colocada em *p
int isAVL_aux (Tree t, int *p) {
    ...
}

int isAVL_opt (Tree a) {
    int p;
    return (isAVL_aux (a, &p));
}
```

ED4: Tabelas de “hash”

Projecto Codeboard:
https://codeboard.io/projects/65989

Arrays como Dicionários

Um simples *array* de comprimento N implementa um dicionário, com o conjunto $\{0 \ldots N-1\}$ como universo de chaves.

- 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.

No entanto, se o universo de chaves for muito grande, torna-se incomportável a utilização directa. Por exemplo, se as chaves forem inteiros de 32 bits, teríamos de utilizar um array de comprimento 2^{32}, superior a 4 mil milhões!!!

A técnica de *hashing* permite separar o universo de chaves do conjunto de índices do array. Por exemplo, se se pretende dimensionar um dicionário com chaves de 32 bits, mas que se prevê nunca tenha uma ocupação superior a 10000, basta usar uma função h que distribua, ou disperse (*hash*) as chaves pelas posições do array:

$$h : \{0 \ldots 2^{32}-1\} \rightarrow \{0 \ldots 9999\}$$

Esta função é não-injectiva por natureza, o que significa que ocorrerão **colisões**: duas chaves poderão ser mapeadas para a mesma posição do array, e só uma pode ser inserida.

Uma característica desejável destas funções de *hash* é a **uniformidade**: todas as posições do array devem ter a mesma probabilidade de ser calculadas como resultado. Isto permitirá distribuir uniformemente a informação inserida no array, minimizando o número de colisões, bem como a formação de *clusters* em zonas específicas do array.

Quando `cap` é um número primo, a seguinte função básica comporta-se de forma razoavelmente uniforme:

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

O desenho de uma tabela de hash implica a escolha de uma *estratégia para a resolução de colisões*, que permita inserir na tabela chaves que são à partidas mapeadas na mesma posição do array

Condicionamento de Chaves

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

Por exemplo, tratando-se de chaves de tipo *string* de caracteres:

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

Tabelas de hash

Uma tabela de hash de capacidade *cap* é uma estrutura de dados física que implementa um dicionário de pares de tipo $K \rightarrow V$, e que consiste em:

1. Uma função de *hash* de tipo $h : K \rightarrow \{0, \ldots cap - 1\}$;

2. De acordo com a estratégia de resolução de colisões adoptada:

   - **Open adressing**: um array com posições $\{0 \ldots cap - 1\}$ de pares (k, v), ou
   - **Closed adressing:** um array com posições $\{0 \ldots cap - 1\}$ de (apontadores para) listas ligadas de pares (k, v).

O desenho de uma tabela de hash pretende sempre equilibrar a eficiência espacial e temporal.

- No limite, se $cap = \#(K)$, não há colisões e as operações de inserção, pesquisa, e remoção executam em tempo constante. Mas claro, desperdiça-se potencialmente muito espaço, porque o tamanho útil da tabela poderá ser muito inferior a $\#(K)$

- Ao diminuir substancialmente o tamanho cap do array, melhora-se a gestão do espaço, às custas de piorar a performance temporal das operações

- O princípio por que se rege o desenho das tabelas é que o tempo deverá sempre ser *tendencialmente constante*

Factor de Carga e Redimensionamento

Qualquer que seja a estratégia de resolução de colisões, terá de ser mantido num valor razoavelmente baixo a taxa de ocupação da tabela, ou *factor de carga:*

$$\alpha = \frac{\#\text{chaves inseridas}}{cap}$$

Ao desenhar uma tabela deve estipular-se um valor máximo para este factor, por exemplo $\alpha_{max} = 0.8$. Quando $\alpha = \alpha_{max}$ deverá redimensionar-se a tabela, que por esta razão deverá ser implementada por um *vector dinâmico*.

Relembre-se que as operações de redimensionamento (duplicação do tamanho da tabela) executam em *tempo amortizado constante.*

Closed Addressing

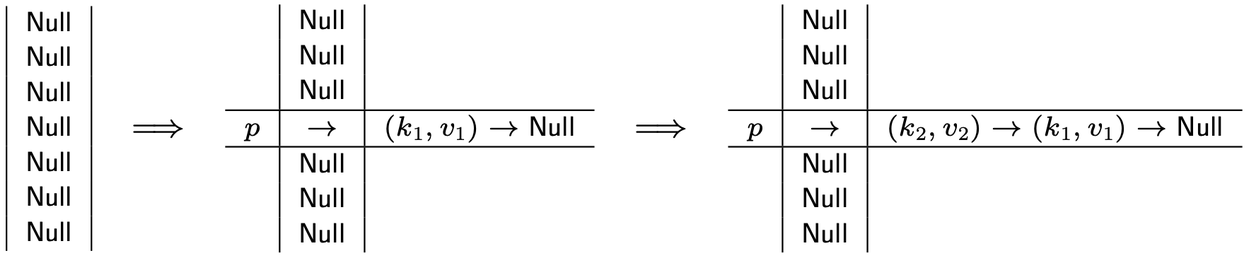
Uma solução possível para fazer "caber" vários pares chave → valor na mesma posição de um array é externalizar a informação, criando uma *lista ligada* cujo endereço inicial é guardado no array:

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

Chama-se a esta implementação uma tabela encadeada (tabela com "chaining")

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

Consideremos inserção dos pares (k_1, v_1) e (k_2, v_2), com $h(k_1) = p$ e também $h(k_2) = p$.



Pontos a reter:

- Não é suposto que estas listas cresçaam indefinidamente: apesar de em teoria poder ser $\alpha > 1$, com um valor alto do factor de carga o tempo de execução das operações deixaria de ser "tendencialmente constante"

- As tabelas devem pois ser redimensionadas quando necessário, assegurando-se um factor de carga pequeno.

- O tempo de execução no pior caso de uma inserção ou consulta será $\Theta(1)$, desde que a função de hash seja uniforme e o factor de carga (que corresponderá ao comprimento médio das listas) pequeno.

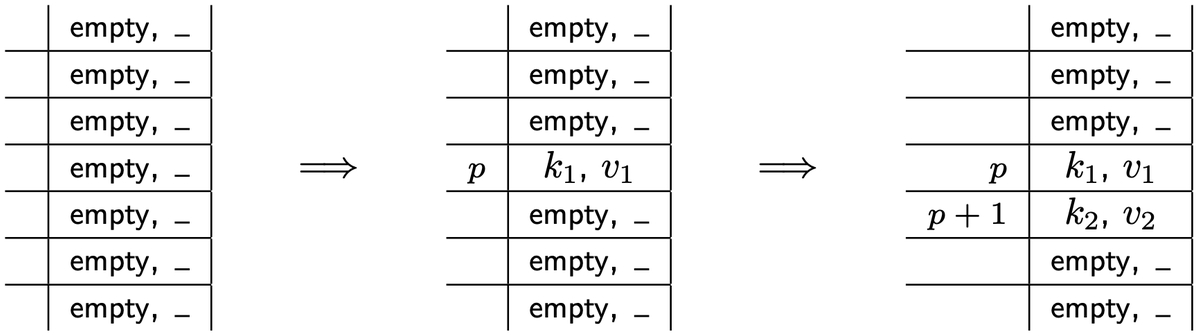
Open Addressing

Ocorrendo uma colisão, procurar-se-á inserir a segunda chave numa outra posição do array, usando um método que possa ser reproduzido (nomeadamente quando se efectuar consultas)

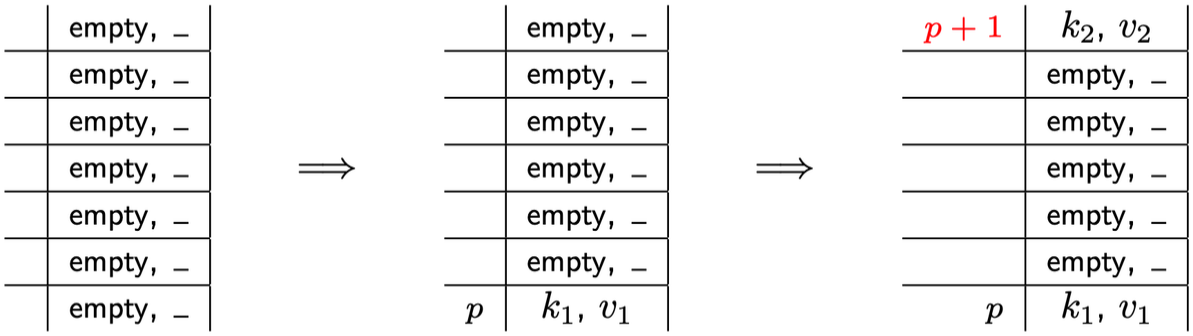
Linear Probing

O método mais trivial de endereçamento aberto é conhecido por *linear probing*: a colisão resolve-se inserindo na posição seguinte (com circularidade) do vector.

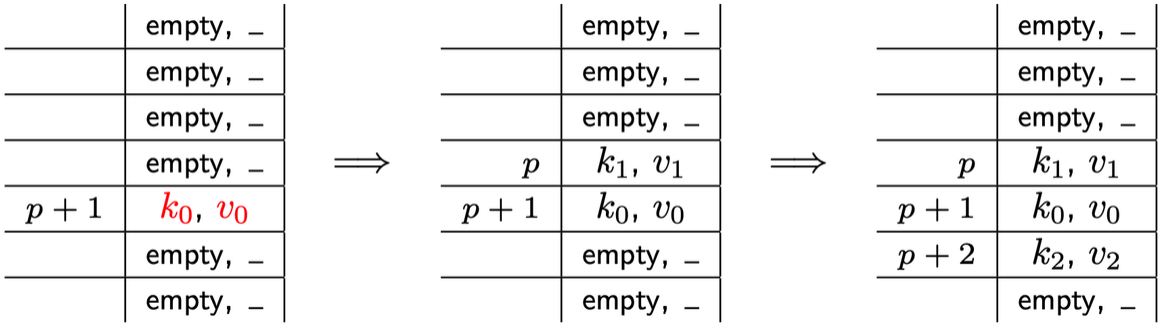
Consideremos de novo a inserção dos pares (k_1, v_1) e (k_2, v_2), por esta ordem, com $h(k_1) = p$ e $h(k_2) = p$.



Ou:



Além disso, "próxima posição" deve ser de facto interpretado como "próxima posição livre". Seja $h(k_0) = p + 1$ e $h(k1) = h(k2) = p$, com k_0 inserido antes de k_1:

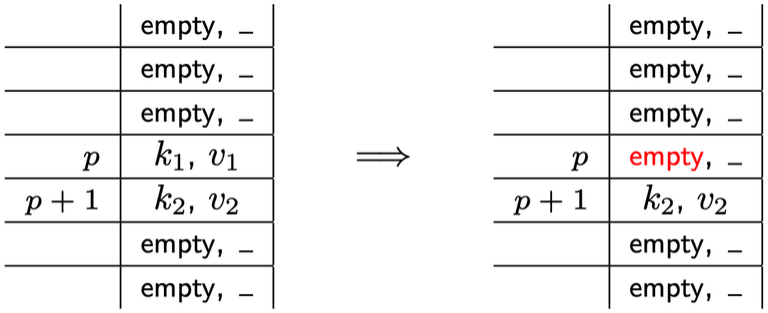


Naturalmente, a operação de consulta deve reproduzir a mesma sequência de *probes* utilizada na inserção.

> *Qual deverá ser o critério de paragem de uma operação de consulta?*

Remoção de Chaves

Efectuemos agora as operações `insert(k1, v1)`, `insert(k2, v2)`, e `remove(k1)` por esta ordem, ainda com $h(k_1) = p$ e $h(k_2) = p$:



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

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.

No entanto, a utilização desta chave `removed` contribui para a degradação da performance da operação de consulta. Ao fim de algum tempo já não haverá chaves `empty`, o que significa que as pesquisas de chaves inexistentes na tabela executarão todas em tempo linear.

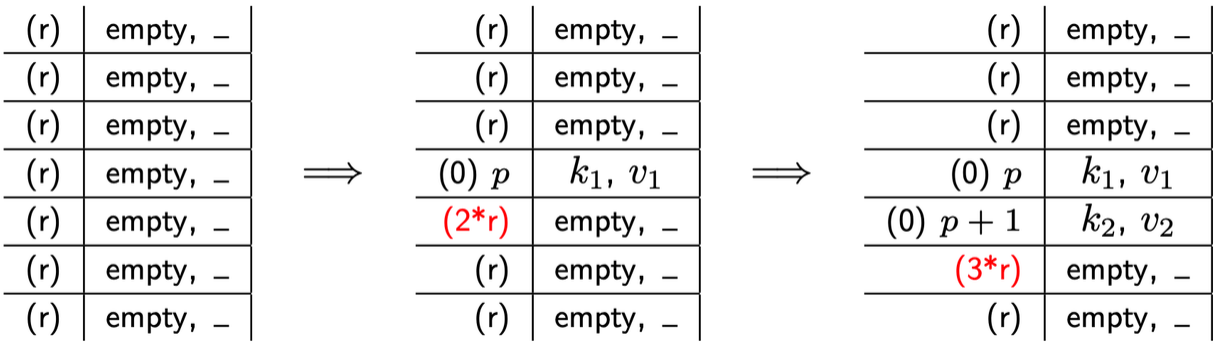
É pois necessário proceder periodicamente a um "refrescamento" da tabela, reinicializado-a e voltando a inserir todos os pares, por forma a eliminar as chaves `removed`. Se a tabela for redimensionada frequentemente isto não será necessário, uma vez que ao redimensionar eliminam-se naturalmente as chaves `removed`.

Clustering

Um problema da estratégia de *linear probing* é a formação de clusters.

A probabilidade de cada posição ser preenchida é inicialmente dada por $r = \frac{1}{cap}$.

Revisitemos a sequência de inserções anterior, calculando a probabilidade de inserção em cada posição:



Este fenómeno de aumento da probabilidade de inserção em posições subsequentes às já preenchidas resulta na formação de "clusters", que deterioram localmente o comportamento das operações sobre a tabela.

Este fenómeno pode ser mitigado com a utilização de outras técnicas de *open addressing*, como *quadratic probing*.

Quadratic Probing

Em vez de fazer os probes

$$p, p+1, p+2, p+3 \ldots$$

faz-se:

$$p, p + 1^2, p + 2^2, p + 3^2 \ldots$$

Esta técnica reduz substancialmente a formação de clusters, às custas de piorar o potencial para aproveitamento de *caching* por apresentar menor grau de localidade.

Dados Empíricos e Comparação

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, *chaining*	1.04	1.2	1.4	1.4	1.5	2.0
Consulta com sucesso, *quadratic probing*	1.04	1.5	2.1	2.7	5.2	–
Consulta com sucesso, *linear probing*	1.05	1.6	3.4	6.2	21.3	–
Consulta sem sucesso, *chaining*	0.10	0.5	0.8	0.9	0.99	2.0
Consulta sem sucesso, *quadratic probing*	1.13	2.2	5.2	11.9	126	–
Consulta sem sucesso, *linear probing*	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

Conclusões

- Para factores de carga razoáveis (≤ 0.8), ambas as soluçõses alcançam némero 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*
  - apresenta vantagens em termos de *caching* (mesmo com *quadratic probing*). 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 estruturas ligadas em termos de localidade.