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 $\mathcal{O}(\log n)$.

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 igual por su*perior ou igual* 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.

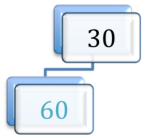
Algoritmo de Inserção:

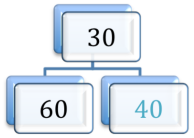
1. Insere-se o novo elemento na primeira posição livre da heap, i.e. na posição mais à esquerda do último nível da heap;

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.

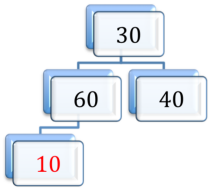
```
Insert 30;
Insert 60;
Insert 40;
```

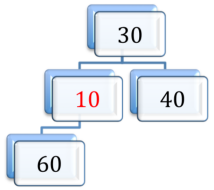


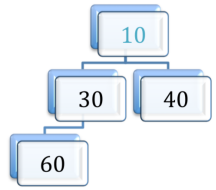


Nestes primeiros passos o invariante de ordem foi respeitado, pelo que não foi necessário executar *bubble-up.* No próximo passo isso já não será assim.

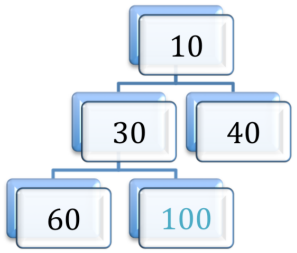
```
Insert 10;
```

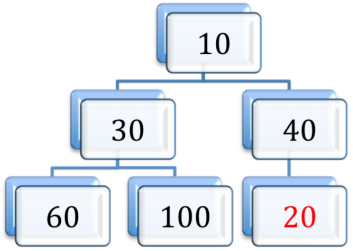


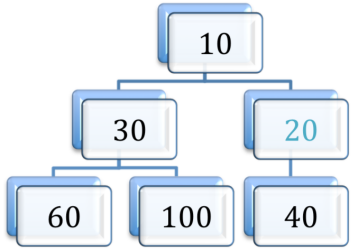




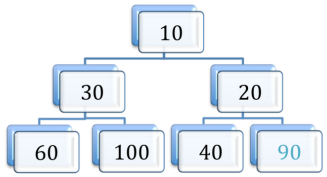

```
Insert 100;
Insert 20;
```

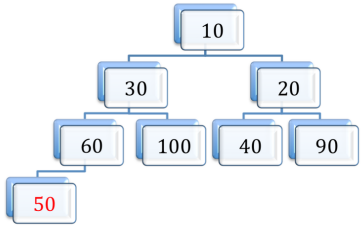


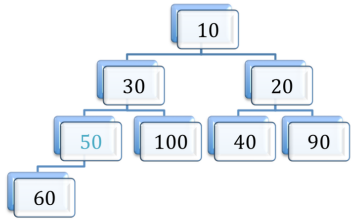




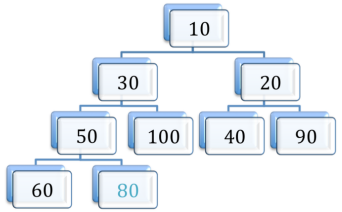

```
Insert 90;
Insert 50;
```

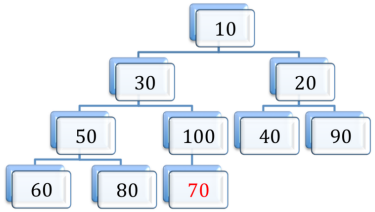


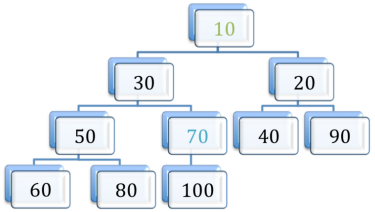





```
Insert 80;
Insert 70;
```







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* $\mathcal{O}(\log N)$*, 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*.

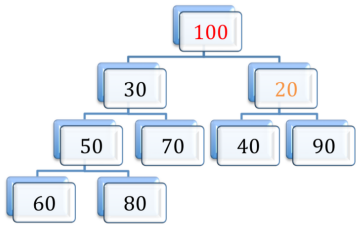
O algoritmo correcto é o seguinte:

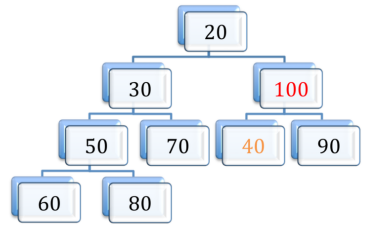
1. Remove-se o elemento inserido na última posição da heap, i.e. na posição mais à direita do último nível da heap, e inscreve-se este mesmo elemento na raíz da *heap,* em substituição da raíz extraída.

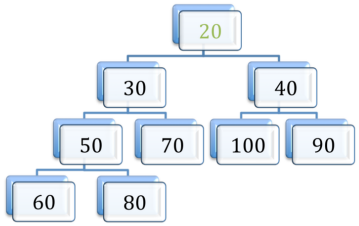
2. Faz-se uma operação de **bubble-down** desta nova raíz:
   Enquanto o nó actual for de valor superior a pelo menos um dos seus filhos, troca-se sucessivamente (ao longo de um caminho descendente da *heap*) o valor do nó com o do menor dos seus filhos

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

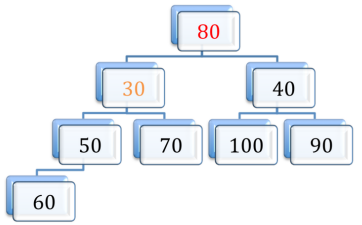
```
Pull;
> 10
```

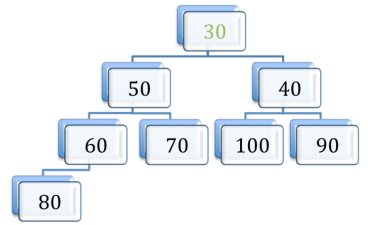


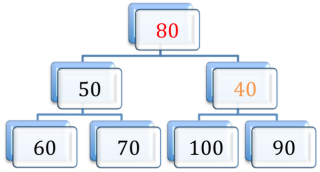


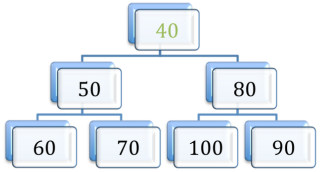



```
Pull;
> 20
Pull;
> 30
```



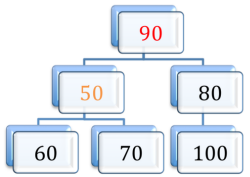


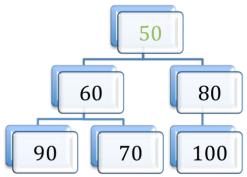


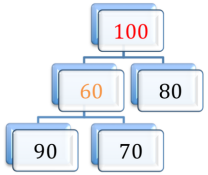


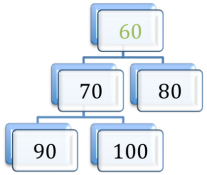

```
Pull;
> 40
```

```
Pull;
> 50
```

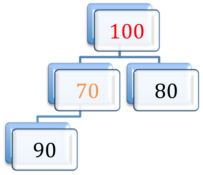


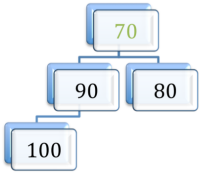


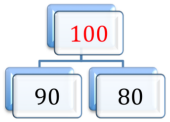


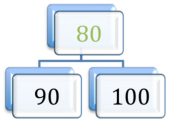



```
Pull;
> 60
Pull;
> 70
Pull;
> 80
Pull;
> 90
```









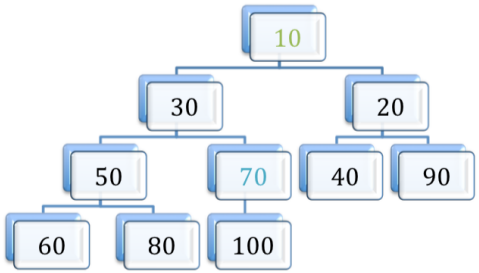
Heaps: Implementação Física

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* $\Omega(1), \mathcal{O}(\log N)$.

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 $\Omega(1), \mathcal{O}(\log N)$
- 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.