Estruturas de dados funcionais -- Pairing Heaps

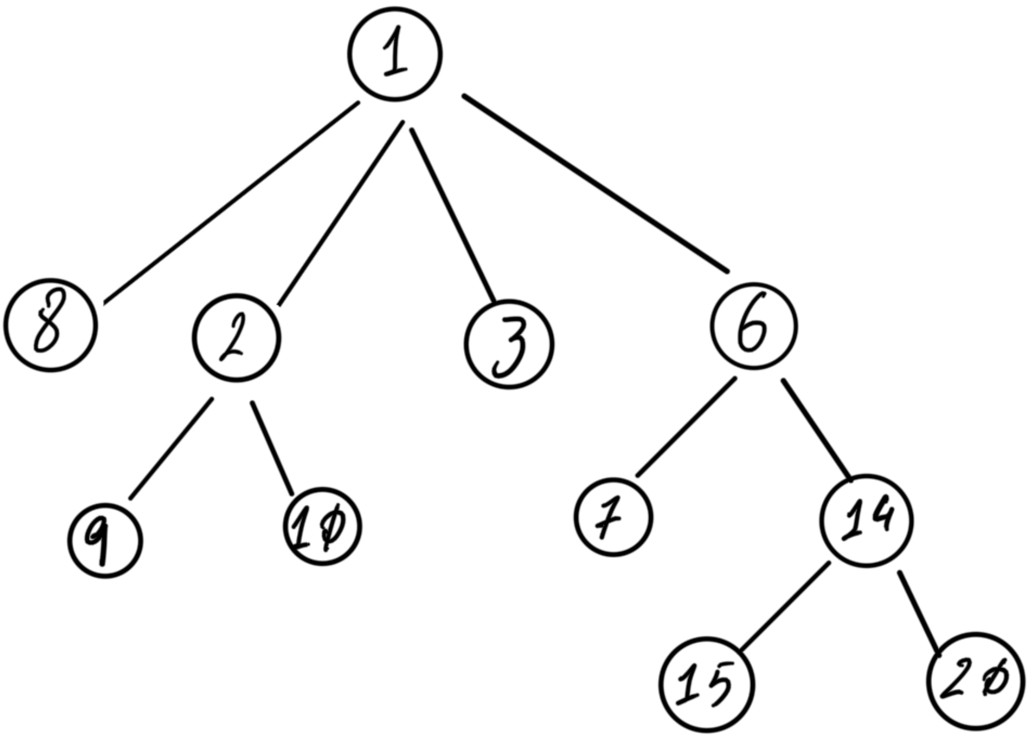
Este *Notebook* propõe a implementação, em OCaml, de uma **Fila de Prioridade** baseada numa estrutura de dados funcional. A variante que apresentamos denomina-se *Pairing Heap* e é codificada usando uma árvore n-ária.

Uma boa descrição do comportamento das Pairing Heaps pode ser encontrado na página da Wikipedia. Uma apresentação ainda melhor desta estrutura de dados pode ser encontrada na obra prima de C. Okasaki, *Purely Functional Data Structures*, Capítulo 5.5.

Apesar da sua simplicidade de implementação, Pairing Heaps são de facto muito eficientes em tempo de execução e competitivas com implementações imperativas de Heaps (*e.g.*, árvore binária codificada num *array*).

Definição do tipo de dados

A imagem seguinte apresenta uma Pairing Heap válida:



Uma Pairing Heap é definida da seguinte maneira:

1. é a árvore vazia, ou
2. é a árvore com uma raiz e uma lista de Pairing Heaps, *i.e.*, os descendentes da raiz, e
3. a raiz de uma Pairing Heap é o menor elemento da árvore (menor ou igual). Pode conter repetições.

Se seguirmos fielmente esta descrição para definir o tipo de dados de uma Pairing Heap poderiamos introduzir a árvore vazia como descendente de um nó. Assim, para evitar esta armadilha, começamos por introduzir o tipo de dados de uma árvore *interior*, que será sempre não-vazia:

In [ ]:
```
type tree = T of int * tree list
```

O constructor `T` representa a raíz de uma sub-árvore composta por uma chave (valor inteiro) e uma lista de descendentes. Estamos agora em posição de definir a noção *top-level* de uma Pairing Heap:

In [ ]:
```
type t = E | N of tree
```

Operações sobre Pairing Heaps

Exercício 1:

Escreva uma definição para a função `create`, com a seguinte assinatura:

```
val create : t
```

Esta função deve simplesmente devolver uma nova Pairing Heap vazia.

In [ ]:
```
let create =
  assert false (* COMPLETAR AQUI *)
```

In [ ]:
```
(* espaço para testes do exercício 1 *)
```

Exercício 2:

Escreva uma definição para a função `find_min`, com a seguinte assinatura:

```
val find_min : t -> int option
```

Esta função espera como *input* uma Pairing Heap `h` e, caso `h` não seja a Heap vazia, devolve o seu valor mínimo (*i.e.*, a sua raiz). Caso `t` seja a Heap vazia, `find_min` deve devolver `None`.

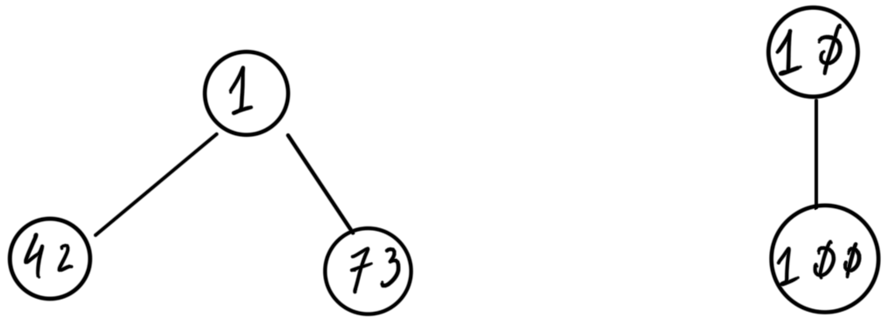
```
In [ ]: let find_min h =
          assert false (* COMPLETAR AQUI *)
```

```
In [ ]: (* espaço para testes do exercício 2 *)
```

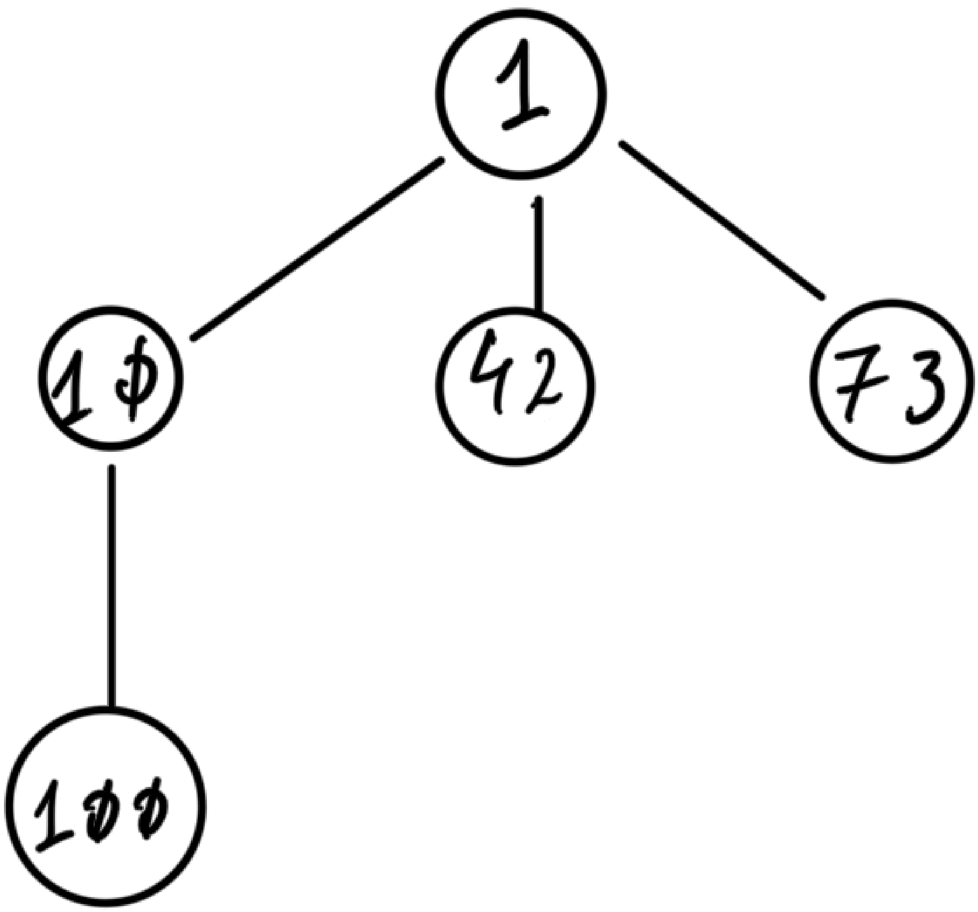
Exercício 3:

A operação de `merge` sobre Pairing Heaps é relativamente simple: dadas duas heaps `h1` and `h2`, se a raíz de `h1` é menor que a raíz de `h2`, devolve uma nova heap em que `h2` se torna o descendente mais à esquerda de `h1`; caso contrário, devolve uma nova heap em que `h1` é o descendente mais à esquerda de `h2`.

Considere, por exemplo, as seguintes Pairing Heaps:



Aplicar a função de `merge` sobre estas duas heaps resultaria na Pairing Heap seguinte:



Portanto, `merge` **não é** uma operação recursiva. Esta função apresenta a seguinte assinatura:

```
val merge : t -> t -> t
```

Escreva uma definição para a função `merge`.

In [ ]:
```
let merge h1 h2 =
  assert false (* COMPLETAR AQUI *)
```

In [ ]:
```
(* espaço para testes do exercício 3 *)
```

Exercício 4:

Escreva uma definição para a função `add`, com a seguinte assinatura:

```
val add : int -> t -> t
```

A função `add` deve usar `merge` como uma função auxiliar. Inserir um novo elemento `x` numa Pairing Heap `h` é tão simples quanto:

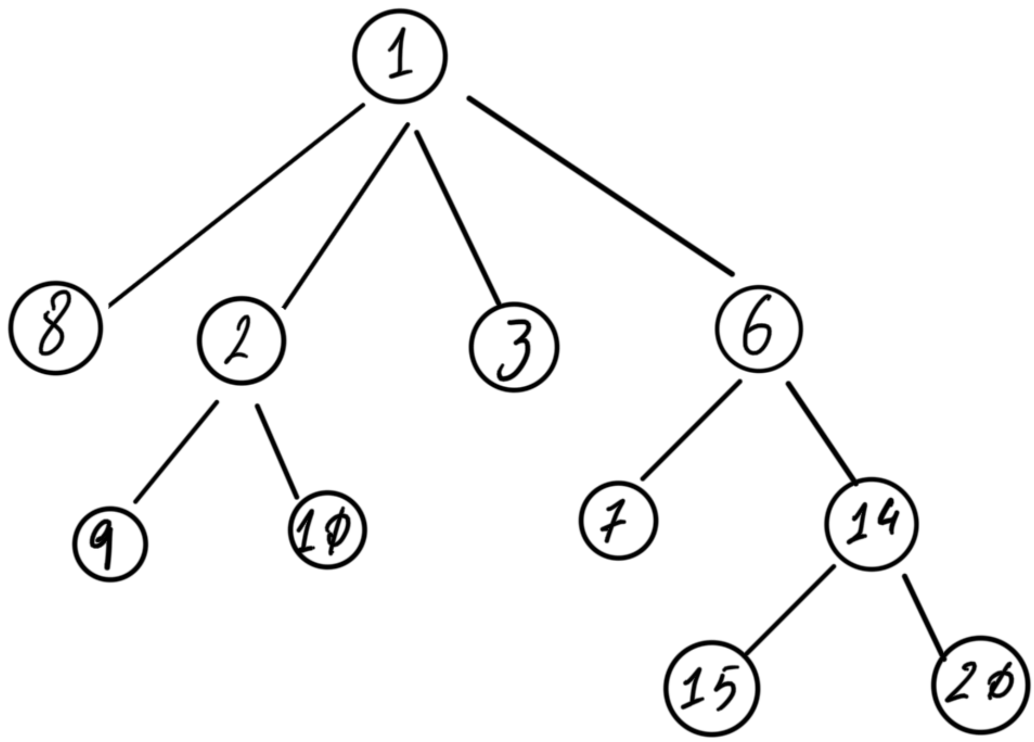
1. criar uma heap temporária `h'` que contém apenas `x`;
2. applicar `merge` a `h'` e `h`.

In [ ]:
```
let add x h =
  assert false (* COMPLETAR AQUI *)
```

In [ ]:
```
(* espaço para testes do exercício 4 *)
```

Exercício 5:

O nome Pairing Heaps advém do uso de uma função auxliar, usada durante a operação de remoção do elemento mínimo da heap. Considerando, novamente, a seguinte heap:



Quando tentamos remover o elemento mínimo, *i.e.*, a raíz `1`, então temos agora de produzir uma única Pairing Heap a partir de uma lista de Pairing Heaps. Seguimos, assim, uma estratégia em dois passos:

1. aplicamos `merge` a cada par de Pairing Heaps consecutivas na lista (a primeira com a segunda, a terceira com a quarta, e assim sucessivamente);
2. aplicar `merge_list`, recursivamente, a todas as Pairing Heaps intermédias resultantes, da esquerda para a direita até que reste apenas uma Pairing Heap.

Para o exemplo acima, obteriamos a seguinte Pairing Heap:

```
N
 (T (2,
   [T (3, [T (6, [T (7, []); T (14, [T (15, []); T (20,
```

```
[])])])]); T (8, []);
    T (9, []); T (10, [])]))
```

Escreva uma definição da função `merge_list`, com a seguinte assinatura:

```
val merge_list : tree list -> t
```

In [ ]:
```
let rec merge_list h =
  assert false (* COMPLETAR AQUI *)
```

In [ ]:
```
(* espaço para testes do exercício 5 *)
```

Exercício 6:

Escreva uma definição para a função `delete_min`, com a seguinte assinatura:

```
val delete_min : t -> t
```

A função `delete_min` recebe como argumento uma heap `h` e devolve uma nova com os mesmos elementos de `h`, excepto o elemento mínimo. Caso `h` seja a heap vazia, então `heap h = h`.

In [ ]:
```
let delete_min h =
  assert false (* COMPLETAR AQUI *)
```

In [ ]:
```
(* espaço para testes do exercício 6 *)
```

Exercício bónus:

Escreva uma definição para a função `is_heap`, com a seguinte assinatura:

```
val is_heap : t -> bool
```

A função `is_heap` recebe como argumento uma heap `h` e indica se esta se trata, ou não, de uma heap válida. Uma heap é válida se respeita a propriedade de heap: a chave da raíz de qualquer sub-árvore apresenta um valor menor que o das chaves dos seus descendentes.

Dica: comece por definir algumas funções auxiliares, nomeadamente:

- `le_tree_list e l`, com `e` um valor inteiro e `l` uma lista de `tree`, que verifica que `e` é um valor menor que o das chaves de todas as raízes das heaps em `l`.

- `le_tree e t`, com `e` um valor inteiro e `t` um valor do tipo `tree`, que verifica que `e` é menor que a chave da raíz de `t`.

In [ ]:
```
let is_heap h =
  assert false (* COMPLETAR AQUI *)
```