Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores
**Algoritmos e Estruturas de Dados**
Semestre de Verão 2018/19
Segunda série de problemas, parte 1

---

**Observações:**

- Data de entrega: **2 de Maio de 2019**.
- **Não se pode utilizar estruturas de dados presentes em `java.util`.**

# 1 Exercícios

1. Assuma que se pretende definir um tipo de dados `IntArrayList` que lhe permite armazenar uma lista de `k` inteiros, em que `k` é conhecido previamente. Além do construtor, que inicializa este tipo de dados, deverá suportar as seguintes operações, garantindo que a complexidade das mesmas é `O(1)`:

   - `public boolean append(int x)` que adiciona o inteiro `x` à lista;
   - `public int get(int i)` que permite obter o `i`-ésimo elemento desta lista;
   - `public void addToAll(int x)` que adiciona a todos os inteiros presentes nesta lista o inteiro `x`.

   **Implemente** o tipo de dados `IntArrayList` e justifique a respetiva implementação.

2. Realize a classe `ListUtils`, contendo os seguintes métodos estáticos:

   2.1. `<E> Node<E> getMiddle(Node<E> list)`
   que, dada a lista duplamente ligada, sem sentinela e não circular, referenciada por `list`, retorna o nó do meio da lista. Caso a lista seja de dimensão par, retorna o primeiro nó do meio.

   2.2. `<E> void quicksort(Node<E> first, Node<E> last, Comparator<E> cmp)`
   que dada uma sublista duplamente ligada, não circular e sem sentinela, em que `first` é uma referência para o primeiro elemento da sublista e `last` é uma referência para o último elemento da sublista, ordena a sublista de modo crescente, segundo o algoritmo **quicksort** e o comparador `cmp`.

   2.3. `<E> Node<E> merge(Node<E>[] lists, Comparator<E> cmp)`
   que dado um *array* de listas duplamente ligadas, não circulares e sem sentinela, ordenadas pelo comparador `cmp`, retorna uma lista duplamente ligada, circular e com sentinela, resultante da junção ordenada, pelo comparador, das listas presentes em `lists`. A lista resultante deve reutilizar os nós presentes em `lists`. As listas em `lists` devem ficar vazias.

   Para as implementações destes métodos, assuma que cada objecto do tipo `Node<E>` tem 3 campos: um `value` e duas referências, `previous` e `next`.

3. Realize a classe `Iterables`, contendo os seguintes métodos estáticos:

   3.1. `<K,U> Iterable<K> filterBy(Iterable<K> src1, Iterable<U> src2, BiPredicate<K,U> predicate)`
   que retorna um iterável composto pelos elementos da sequência `src1` que satisfazem o predicado `predicate` em conjunto com os elementos da sequência `src2` que ocorrem na mesma posição. Caso a sequência `src1` seja maior do que a sequência `src2`, são descartados os elementos da sequência `src1` que ocorrem numa posição inexistente na sequência `src2`. O tipo `BiPredicate` encontra-se definido em `java.util.function`. Exemplo: se a `src1` representar a sequência $[2, 5, 3, 6, 2, 5, 10, 3]$, `src2` representar a sequência $[2, 1, 3, 6, 4, 5]$ e o objeto `predicate` retornar `true` se a soma dos dois inteiros for 6, então o objeto retornado deve representar a sequência $[5, 3, 2]$.

   3.2. `<K,V> Iterable<V> filterByMap(Iterable<K> src, Map<K,V> map)`
   que dada a sequência `src` de elementos, retorna o iterável que resulta de, para cada elemento da sequência original, caso o mesmo ocorra como chave no mapa `map`, obter o valor associado desse elemento.

   As implementações destes métodos devem minimizar o espaço ocupado pelo iterável. O iterador associado ao iterável retornado não suporta o método `remove`.

ISEL, 11 de Abril de 2019.