UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

2025/2026

Projeto 2 – Coleções

Importante: nesta cadeira, partimos do princípio de que quem não sabe o problema, não sabe a solução. Por esta razão, alunos que não tenham lido o enunciado e que na validação, não saibam o que é pedido para implementar, terão **0 (zero)** neste projeto.

Introdução

No 2.º Projeto pretende-se consolidar os conhecimentos relativos à representação e manipulação de estruturas de dados dinâmicas, com particular foco nas listas ligadas. Neste projeto iremos implementar uma versão mais eficiente de uma lista ligada.

Tradicionalmente, as listas ligadas são implementadas através de nós alocados dinamicamente e interligados por referências (ou apontadores). Contudo, para tirar partido da localidade de referência[1], e respetivas otimizações, iremos implementar a nossa lista ligada utilizando **armazenamento contíguo em memória**, através de um **vetor**.

Adicionalmente, também não usaremos tipos genéricos, pois um tipo genérico em java não nos permite garantir que os objetos criados e armazenados no vetor se encontram efetivamente de forma contígua em memória.

Pretende-se também que os alunos apliquem técnicas de análise de complexidade temporal para determinar a complexidade temporal da sua implementação, comparando-a com a complexidade temporal de uma solução tradicional para uma lista ligada.

Parte A: Implementação Fast Int List (15 valores)

Implemente a classe *FintList* (abreviatura para FastIntList), que implementa uma lista lista ligada usando um vetor continuo em memória. A sua implementação deverá respeitar a seguinte especificação:

FintList– representa uma lista ligada de inteiros, implementada com várias otimizações, como por exemplo a utilização de memória continua para guardar os elementos.		
	`FintList()`	Cria uma lista vazia
`boolean`	`add(int item)`	Coloca um item no fim da lista.

[1] A localidade de referência refere-se à tendência de um processador de aceder a uma zona de memória de forma repetida durante um curto período.
https://en.wikipedia.org/wiki/Locality_of_reference

		Retorna true se a lista tiver sido modificada, e false caso contrário.
`void`	`addAt(int index, int item)`	Adiciona o item na posição *index* da lista indicada. Deve lançar uma *IndexOutOfBoundsException* se o índice recebido não for válido.
`int`	`get(int index)`	Retorna o item na posição index da lista Deve lançar uma *IndexOutOfBoundsException* se o índice recebido não for válido.
`int`	`getFirst()`	Retorna o 1º elemento da lista, ou lança uma excepção IndexOutOfBoundsException se a lista estiver vazia
`int`	`get()`	Retorna o último elemento da lista, ou lança uma excepção IndexOutOfBoundsException se a lista estiver vazia.
`void`	`set(int index, int value))`	Altera a posição index da lista para passar a ter o valor *value* . Deve lançar uma *IndexOutOfBoundsException* se o índice recebido não for válido
`boolean`	`isEmpty()`	Retorna *true* se a lista estiver vazia e *false* caso contrário
`int`	`size()`	Devolve o tamanho (número de elementos) da lista
`int`	`remove()`	Remove o último elemento da lista, retornando-o. Se não houverem elementos, deve lançar uma IndexOutOfBoundsException
`int`	`removeAt(int index)`	Remove o elemento na posição *index* da lista. O elemento removido é retornado. Deve lançar uma *IndexOutOfBoundsException* se o índice recebido não for válido.
`boolean`	`remove(int item)`	Remove o elemento recebido caso ele exista na lista. Retorna *true* se o elemento existia e foi removido, *false* caso contrário.
`boolean`	`contains(int item)`	Verifica se o elemento recebido se encontra na lista. Retorna *true* caso exista, e *false* caso contrário.
`int`	`indexOf(int item)`	Retorna o índice da primeira ocorrência do elemento na lista, ou -1 se o elemento não existir na lista.
`void`	`reverse()`	Troca a ordem dos elementos da lista . Por exemplo, a lista (1,2,3) deverá ser transformada na lista (3,2,1).
`FintList`	`deepCopy()`	Retorna uma cópia profunda da lista.
`Iterator<Integer>`	`Iterator()`	Retorna um iterador com estado para iterar sobre os elementos da lista Este iterador deverá implementar os métodos hasNext() e next() obrigatoriamente
`void`	`forEach( Consumer<Integer> c)`	Recebe uma função que processa inteiros, e aplica essa função a cada um dos elementos da lista.

`void`	`map(UnaryOperator <Integer> op)`	Recebe um operador unário de inteiros, e altera os elementos da lista, guardando o resultado da aplicação do operador recebido. Por exemplo, a aplicação da função quadrado, irá transformar a lista (1,2,3) na lista (1,4,9).
`int`	`reduce( BinaryOperator <Integer> op, int default)`	Recebe um operador binário de inteiros, e um valor por omissão usado para inicializar o redutor Aplica sucessivamente o operador binário aos elementos da lista, (pela ordem da lista) de forma a reduzir a lista de inteiros a um único inteiro. Por exemplo, ao invocarmos *reduce* com uma operação de soma, e 0 (elemento neutro da soma), obtemos a soma de todos os elementos da lista. Se invocarmos *reduce* com uma operação de multiplicação, e 1, obtemos a multiplicação de todos os elementos da lista.
`void`	`main(String[] args)`	Método estático main, que deverá ser usado para testar os métodos acima.

Requisitos técnicos: A lista deverá ser implementada recorrendo a um vetor (array) contínuo em memória.

Não poderá usar na sua implementação nenhuma das coleções nativas do Java (*ArrayList*, *LinkedList*, *Vector*,etc). Também não pode usar a classe *LinkedList* fornecida.

Implemente a classe *FintList* no ficheiro *FintList.java*. A classe deverá estar definida na package *aed.collections*[2]. Outras classes auxiliares deverão ser definidas no mesmo ficheiro.

Requisitos de desempenho:

A sua implementação dos vários métodos da classe deverá ser o mais eficiente possível. Para além da utilização de memória continua deverá ter em consideração as seguintes otimizações adicionais.

Os métodos que recebem um índice (addAt, get, set, removeAt) devem ser executados de forma eficiente quando usados num ciclo for da esquerda para a direita, ou da direita para a esquerda, mesmo quando o ciclo saltar mais do que uma posição de cada vez. Por exemplo, os seguintes ciclos devem ser eficientes:

```
for(int i = 0; i < n; i++)
{
      lista.addAt(i,i);
}

for(int i = n; i >= 0; i--)
{
      lista.removeAt(i);
}

for(int i = 0; i < lista.size(); i+=2)
{
```

[2] Poderá fazê-lo usando a instrução *"package aed.collections;"* (sem as aspas) no ínicio do ficheiro.

```
        System.out.println(lista.get(i));
}
```

Estas otimizações são fáceis de fazer se estes métodos guardarem um estado que prevê qual a próxima posição para a direita/esquerda.

Adicionalmente, os métodos deepCopy e reverse deverão ser implementados da forma mais eficiente possível.

Submeta **apenas o ficheiro *FintList.java*** no Problema A.

O método *main* deverá implementar os testes e ensaios de razão dobrada utilizados. Embora este método não seja validado de forma automática pelo *Mooshak* será tido em consideração na validação do projeto, e contará para a nota final do mesmo.

Parte B: Análise de complexidade temporal (5 valores)

Na parte B do projeto, não irá programar mais nenhuma classe, mas irá analisar a eficiência da implementação da classe FintList e compará-la com a implementação tradicional de uma lista ligada. Para além da submissão do ficheiro no Mooshak, deverá redigir um relatório (com no máximo 10 páginas), e submeter o relatório juntamente com o código do projeto final (incluindo todas as classes, todos os ficheiros *main*, e todos os testes efetuados) na tutoria. Deverá incluir os seguintes pontos no relatório:

1) Determine analiticamente **uma aproximação tilde** para a complexidade temporal amortizada para o melhor caso e para o pior caso de **um** dos seguintes métodos: addAt, get, set, e removeAt. Escolha o método fazendo o resto da divisão inteira do número de aluno mais pequeno do grupo, por 4. Por exemplo, 78345%4 = 1, o que implicaria a análise do método get.

2) Determine analiticamente **uma aproximação tilde** para a complexidade temporal amortizada para o melhor e pior caso do método equivalente ao método avaliado no ponto anterior, mas da classe LinkedList fornecida.

3) Utilize testes empíricos, incluindo ensaios de razão dobrada, para determinar o tempo de execução médio e a ordem de crescimento temporal dos métodos *addAt, removeAt, contains, deepCopy* para a classe *FintList*. Execute o mesmo tipo de testes para a classe *LinkedList* fornecida. Inclua no relatório uma explicação do tipo de exemplos gerados, os resultados incluindo os valores de r (razão dobrada), e uma análise dos resultados, indicando qual o valor estimado para a complexidade temporal assimptótica para cada um dos métodos

4) Faça uma comparação entre os resultados obtidos nos pontos anteriores para a sua implementação, e para a classe *LinkedList fornecida*

5) Indique, justificando, um exemplo de uma situação, problema computacional, ou estrutura de dados, onde a utilização de uma FintList seria mais eficiente, e, portanto, mais apropriada, do que a utilização de um vetor (array) de inteiros.

Importante: relatórios gerados por um Large Language Model não serão lidos pelo corpo docente, e terão **0 (zero)** na componente do relatório.

Condições de realização

O projeto deve ser realizado em grupos de 2 alunos. Recomendamos que os grupos sejam formados por alunos do mesmo turno de laboratório. Projetos iguais, ou muito semelhantes, serão anulados e poderão dar origem a reprovação na disciplina. O mesmo se aplica ao relatório entregue. O corpo docente da disciplina será o único juiz do que se considera ou não copiar num projeto.

O código do projeto deverá ser entregue obrigatoriamente por via eletrónica, através do sistema Mooshak, **até às 23:59 do dia 24 de outubro.** O relatório deverá ser entregue também por via eletrónica na página da cadeira na tutoria um dia mais tarde, ou seja **até às 23:59 do dia 25 de outubro**. As validações/discussões do projeto terão lugar na semana seguinte, de 27 a 31 de outubro. Os alunos terão de fazer a discussão juntamente com o docente **durante** o horário de laboratório correspondente ao turno em que estão inscritos. **A avaliação e correspondente nota do projeto só terá efeito após a discussão do projeto.** O corpo docente poderá atribuir **notas diferentes aos elementos do grupo, de acordo com a sua prestação individual na discussão.**

A avaliação da execução do código é feita automaticamente através do sistema Mooshak, usando vários testes configurados no sistema. O tempo de execução de cada teste está limitado, bem como a memória utilizada. Não é necessário o registo para quem já se registou no 1.º projeto, podendo usar o mesmo *username* e *password*.