UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Projeto 2 – Coleções (v2.2)

Introdução

No 2.º Projeto pretende-se explorar a criação e utilização de estruturas de dados usadas para representar coleções de objetos. Em particular iremos explorar a implementação de Fila de tamanho limitado usando um *array*, e uma Lista Sovina.

Os problemas A e B são completamente independentes. É possível fazer o problema A sem fazer o problema B e vice-versa. Não é necessário ter um problema totalmente implementado para obter a cotação correspondente no projeto. Mesmo que não tenha tempo para implementar algum dos métodos pedidos, deverá escrever o cabeçalho do mesmo, para que o *Mooshak* consiga compilar o código e executar os testes sobre os métodos que implementou.

Problema A: Implementação de Filas Limitadas através de Arrays (7 valores)

Implemente a classe *QueueArray*, que representa uma fila com tamanho máximo limitado (i.e. a fila não poderá conter mais elementos que o número máximo permitido). A implementação deverá seguir a seguinte especificação:

QueueArray<Item> – representa uma fila de elementos do tipo *Item*. Uma fila respeita a ordem de entrada dos elementos na mesma, sendo que o primeiro elemento a ser colocado (*enqueue*) será o primeiro a ser processado (*dequeue*). Esta fila tem uma capacidade máxima de elementos, a partir dos quais não será permitirá a entrada de novos elementos. A classe *QueueArray<Item>* deve implementar a interface *Iterable<Item>*
`QueueArray(int max)`
Cria uma fila vazia, preparada para conter um número máximo *max* de elementos
`void enqueue(Item item)`
Coloca um item no fim da fila. O item não pode ser *null.* Caso seja *null* deverá ser lançada uma *IllegalArgumentException*. Se a fila já estiver cheia é lançada uma exceção do tipo *OutOfMemoryError*
`Item dequeue()`
Remove e retorna o item no início da fila. Caso a fila esteja vazia deverá retornar *null*
`Item peek()`
Retorna o item no início da fila, mas não o remove. Retorna *null* caso a fila esteja vazia
`boolean isEmpty()`
Retorna *true* se a fila estiver vazia e *false* caso contrário
`int size()`
Devolve o tamanho (número de elementos) da fila
`QueueArray<Item> shallowCopy()`
Retorna uma cópia superficial da fila. Uma cópia superficial copia a estrutura da fila sem copiar cada item individualmente
`Iterator<Item> iterator()`
Retorna um iterador com estado para iterar sobre os elementos da fila. O iterador deve percorrer os elementos pela ordem natural da fila, ou seja em 1.º lugar o que está no início da fila, etc. Este iterador deverá implementar os métodos *hasNext()* e *next()* obrigatoriamente
`void main(String[] args)`
Método *main*, que deverá ser usado para testar os métodos acima.

Requisitos técnicos: Esta fila deverá ser implementada obrigatoriamente através de um *Array* para guardar os elementos. Para que a implementação seja o mais eficiente possível, deverá implementar esta classe **sem redimensionar** o *Array*. Não poderá usar na sua implementação nenhuma das colecções nativas do Java (*ArrayList*, *LinkedList*, *Vector*, etc), mas poderá usar *Arrays*.

Implemente a classe *QueueArray* no ficheiro *QueueArray.java*. A classe deverá estar definida na package *aed.collections*[1]. Outras classes auxiliares de que necessite deverão ser definidas no mesmo ficheiro.

Submeta **apenas o ficheiro QueueArray.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.

Problema B: Lista Sovina – uma implementação de uma lista duplamente ligada (10 valores)

Uma das desvantagens de uma lista duplamente ligada face a uma lista simplesmente ligada é que uma lista duplamente ligada gasta mais memória devido à necessidade de guardar dois ponteiros em cada nó.

No entanto, existe uma implementação alternativa de uma lista duplamente ligada que não tem esta desvantagem. A lista que iremos implementar no problema B, chamemos-lhe Lista Sovina, é uma lista duplamente ligada que usa nós com apenas um único ponteiro.

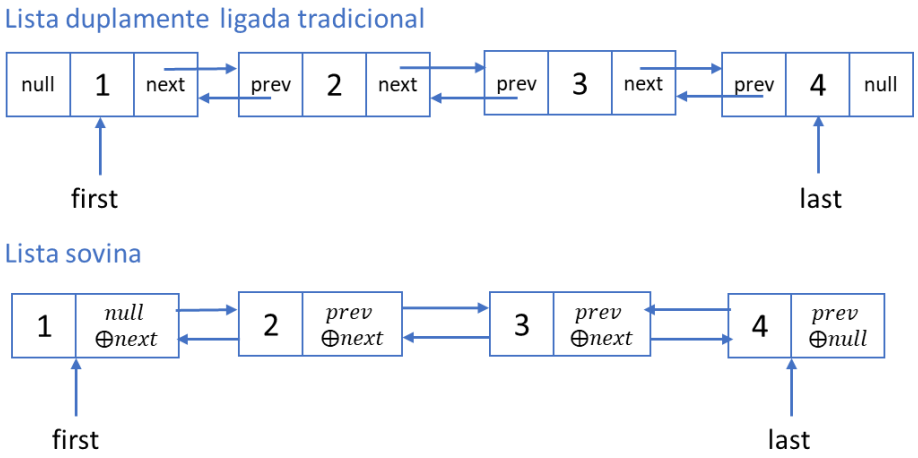


Figura 1 – Exemplo de como os ponteiros para os nós anterior e próximo são guardados numa lista sovina

Para o conseguir fazer, iremos guardar dentro do ponteiro o resultado da operação Ou Exclusivo (XOR ⊕) aplicada a ambos os ponteiros *previous* e *next*. A figura 1 ilustra como os membros *previous* e *next* que guardam respetivamente o ponteiro para o nó anterior e para o próximo nó são guardados numa lista sovina.

Pode parecer estranho, mas estamos a tirar partido das propriedades do operador XOR. Ao fazermos $p \oplus n$ obtemos um novo valor onde para cada bit, 0 representa que *p* e *n* são iguais

[1] Poderá fazê-lo usando a instrução *"package aed.collections;"* (sem as aspas) no ínicio do ficheiro. O ficheiro deverá estar guardado na pasta *"src/aed/collections."*

nesse bit, e 1 representa que são diferentes. Assim sendo, a partir de $p \oplus n$ podemos obter *n* se tivermos *p*, e podemos obter *p* se tivermos *n*.

Podemos demonstrar isto facilmente, considerando as propriedades do operador XOR:

1. Comutativa: $a \oplus b = b \oplus a$
2. Associativa: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$
3. Auto-inversa: $a \oplus a = 0$
4. Identidade: $a \oplus 0 = a$

Figura 2 – Propriedades do operador Ou Exclusivo

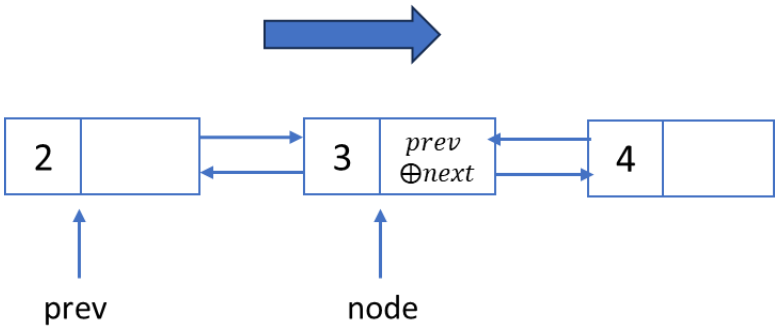
Se a partir de $p \oplus n$ aplicarmos novamente o operador XOR com *n*, iremos derivar a seguinte expressão:

$(p \oplus n) \oplus n = p \oplus (n \oplus n) = p \oplus 0 = p$ (pelas propriedades 2, 3 e 4).

Da mesma forma, se a $p \oplus n$ aplicarmos novamente o operador XOR com *p*, iremos obter:

$(p \oplus n) \oplus p = (n \oplus p) \oplus p = n \oplus (p \oplus p) = n \oplus 0 = n$ (pelas propriedades 1, 2, 3 e 4).

Por outras palavras, isto implica que ao guardarmos dentro de cada ponteiro o resultado de $prev \oplus next$, conseguimos obter facilmente *next* se tivermos *prev*, e vice-versa. Felizmente estas condições são fáceis de garantir quando estamos a percorrer uma lista duplamente ligada. Quando percorremos a lista da esquerda para a direita, é fácil guardar o último nó de onde viemos numa variável e este corresponde ao *previous*. Tendo o ponteiro *previous* de onde viemos, e o nó que estamos a analisar no momento, podemos obter facilmente o ponteiro para o próximo nó *next* como ilustrado na figura 3.



$$prev \oplus (prev \oplus next) = next$$

Figura 3 – Obtendo o ponteiro para o próximo nó a partir do nó n atual, e do ponteiro para o nó anterior

Quando percorremos a lista da direita para a esquerda a lógica é semelhante. É fácil guardar o último nó de onde viemos numa variável, mas agora este corresponde ao *next*. Tendo o ponteiro *next*, e o nó que estamos a analisar no momento podemos obter facilmente o ponteiro para o nó anterior *previous*, fazendo uma vez mais um XOR. Não vale a pena colocar uma imagem para ilustrar, mas sabemos que $next \oplus (prev \oplus next) = prev$.

A lógica aqui descrita é também válida quando o nó que estamos a aceder corresponde ao 1.º ou ao último nó da lista, mas nesse caso, o *previous* e/ou *next* têm o valor *NULL*. Podemos aplicar o operador XOR com *NULL*[2].

Implemente a classe StingyList de forma a que esta se comporte como uma lista sovina. A sua implementação deverá seguir a seguinte especificação:

StingyList<Item> – representa uma lista sovina de elementos do tipo *Item. Esta é uma lista duplamente ligada mas que gasta a quantidade de memória de uma lista ligada tradicional. A lista pode ser percorrida do início para o fim ou do fim para o início.* De forma a preservar a ordem de inserção, novos itens são colocados no fim da lista. A classe *StingyList<Item>* deve implementar a interface *Iterable<Item>*.
`StingyList()`
Construtor que cria uma lista vazia
`void add(Item item)`
Coloca um item no fim da lista. O item não pode ser *null*. Caso seja *null* deverá será lançada uma *IllegalArgumentException*
`Item remove()`
Remove o último elemento da lista e retorna-o. Caso a lista esteja vazia, deve ser lançada uma *IndexOutOfBoundsException*.
`Item get()`
Retorna o último elemento da lista. Caso a lista esteja vazia, deve ser lançada uma *IndexOutOfBoundsException*.
`Item get(int index)`
Retorna o elemento que está na posição index da lista. Se o index for inválido para o número de elementos atuais da lista é lançada uma *IndexOutOfBoundsException*.
`Item getSlow(int index)`
Retorna o elemento que está na posição index da lista. Se o index for inválido para o número de elementos atuais da lista é lançada uma *IndexOutOfBoundsException*. Este método não tira partido das ligações duplas da lista e percorre a lista sempre da esquerda para a direita.
`void addAt(int index, Item item)`
Adiciona um elemento a uma posição específica da lista, aumentando o número de elementos da lista. Caso já exista um elemento nessa posição, esse elemento continua a existir passando para a posição seguinte. Se index=0, corresponde a inserir o elemento no início da lista, se index= tamanho da lista, corresponde a inserir no fim. Se o index for inválido para o número de elementos atuais da lista é lançada uma *IndexOutOfBoundsException*.
`Item removeAt(int index)`
Remove o elemento que está na posição *index* da lista e retorna-o. Se o *index* for inválido para o número de elementos atuais da lista é lançada uma *IndexOutOfBoundsException*.
`void reverse()`
Altera a lista, de forma a reverter a ordem dos elementos na lista.
`StingyList reversed()`
Devolve uma nova lista, que partilha essencialmente os nós da lista original, mas pela ordem contrária. Alterações à esta lista irão afetar a lista original e vice-versa.
`void clear()`
Remove todos os items da lista, deixando-a vazia.
`boolean isEmpty()`
Retorna *true* se a lista estiver vazia e *false* caso contrário
`int size()`
Devolve o tamanho (número de elementos) da lista
`Object[] toArray()`
Este método[3] retorna um *array* com todos os objetos da lista, usando a ordem pela qual os elementos estão guardados na lista (por exemplo o 1.º item irá ser guardado no índice 0).

[2] *NULL* corresponde a um ponteiro com valor 0 em todos os bits

[3] Este método é usado apenas para testar os vossos métodos no *Mooshak*, e não precisam de se preocupar com a sua eficiência. Retorna um *array* de Objetos em vez de Items, porque os *arrays* não funcionam muito bem com tipos genéricos.

`Iterator<Item>  iterator()`
Retorna um iterador com estado para iterar sobre os elementos da lista, do início para o fim da lista. Este iterador deverá implementar os métodos *hasNext()* e *next()* obrigatoriamente.
`void  main(String[] args)`
Método *main*, que deverá ser usado para testar os métodos acima.

Requisitos técnicos: Esta lista deverá ser implementada de forma que cada nó da lista só tenha um ponteiro para outro nó, utilizando a técnica baseada no operador Ou Exclusivo.

Para conseguirem implementar esta técnica, deverão adicionar as classes disponibilizadas na tutoria (Ficheiros de Suporte ao Problema B) ao vosso projeto na package *aed.collections*. Utilizem os métodos definidos na classe UNode para criar e trabalhar com os nós da Lista Sovina.

Os métodos que recebem um índice como argumento (exceto o *getSlow*) deverão tirar partido das ligações duplas na lista para serem mais eficientes: se o *index* recebido estiver mais perto do início da lista, a lista é percorrida do início para o fim, se estiver mais perto do fim, a lista é percorrida do fim para o início.

Uma outra vantagem desta estrutura de dados, é que é possível reverter a lista em tempo constante. A sua implementação dos métodos *reverse* e *reversed* deverá ter uma complexidade temporal assimptótica de O(1).

Não poderá usar na sua implementação nenhuma das coleções nativas do Java (*ArrayList*, *LinkedList*, *Vector*, etc…).

Implemente a classe *StingyList* no ficheiro *StingyList.java*. A classe deverá estar definida na package *aed.collections*[4]. Outras classes auxiliares de que necessite deverão ser definidas no mesmo ficheiro.

Submeta **apenas o ficheiro StingyList.java** no Problema B.

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.

Relatório – Análise de complexidade temporal (3 valores)

Neste projeto, para além da implementação correta dos problemas A e B, pretende-se que os alunos aprendam a executar testes empíricos e ensaios de razão dobrada para estimar o tempo de execução médio e complexidade temporal para alguns métodos. Portanto, para além da submissão no Mooshak, deverá ser feita a entrega de um relatório em formato pdf com 2 a 4 páginas com a descrição dos testes efetuados, resultados obtidos, e uma análise dos resultados.

Testes empíricos a realizar para o problema A

Utilize testes empíricos, incluindo ensaios de razão dobrada, para determinar o tempo de execução médio e a ordem de crescimento temporal dos métodos *queue* e *dequeue* para a classe *QueueArray*. 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.

[4] Poderá fazê-lo usando a instrução *"package aed.collections;"* (sem as aspas) no ínicio do ficheiro. O ficheiro deverá estar guardado na pasta *"src/aed/collections."*

Testes empíricos a realizar para o problema B

Utilize testes empíricos, incluindo ensaios de razão dobrada, para determinar o tempo de execução médio e a ordem de crescimento temporal dos métodos *get(int index)*, *getSlow(int index)* e *reverse()* para a classe *StingyList*. 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.

Compare os resultados obtidos para os métodos *get(int index)* e *getSlow(int index)*. O que pode concluir acerca dos ganhos obtidos pela utilização de uma lista duplamente ligada face a uma lista ligada simples?

Condições de realização

O projeto deve ser realizado em grupos de no máximo 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 3 de Novembro.** 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 4 de Novembro**. As validações/discussões do projeto terão lugar na semana seguinte, de 6 a 10 de Novembro. 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*.