Algoritmos e Estruturas de Dados

2ª Série

Operações entre coleções de pontos no plano

Nome Luís Mota

Nome Jõao Gonçalves

Licenciatura em Engenharia Informática e de Computadores

Semestre de Verão 2024/2025

18/5/2025

Índice

1. Introdução

O presente trabalho tem como objetivo aplicar os conhecimentos adquiridos nas aulas, através da resolução de problemas que envolvem diferentes estruturas de dados. A abordagem seguida inclui a análise, implementação e avaliação de algoritmos eficientes, com vista a consolidar os princípios fundamentais da disciplina.

A primeira parte consiste na implementação de uma função que identifica o menor elemento num *Max-Heap*, tirando partido das suas propriedades para otimizar a pesquisa. Segue-se o desenvolvimento da estrutura *IntArrayList*, uma lista circular de inteiros com operações em tempo constante, segundo a política FIFO.

O trabalho aborda ainda a manipulação de listas duplamente ligadas, nomeadamente a reorganização de elementos pares e ímpares numa lista circular com sentinela, e a interseção de duas listas ordenadas, reutilizando nós e produzindo uma nova lista não circular e sem sentinela.

Por fim, é apresentada a implementação de uma estrutura genérica *MutableMap*<K, V>, baseada em tabela de dispersão com encadeamento externo, utilizando listas ligadas simples para uma gestão eficiente de pares chave–valor.

2. Análise de desempenho

Este conjunto de exercícios aborda a implementação e análise de estruturas de dados fundamentais e suas operações eficientes. Primeiramente, explora-se a função *minimum*, que identifica o menor elemento em um max heap, aproveitando a organização interna do heap representado como array para garantir complexidade linear. Em seguida, define-se o tipo de dados *IntArrayList*, uma estrutura baseada em fila circular que suporta operações como *append*, *get*, *addToAll* e *remove* com complexidade O(1), utilizando um mecanismo de adição virtual para otimizar o desempenho. Também são implementadas as funções *splitEvensAndOdds*, para reorganização de listas duplamente ligadas circulares, e *intersection*, que produz a interseção ordenada de duas listas duplamente ligadas circulares reutilizando seus nós. Finalmente, é apresentada a implementação de um *MutableMap* utilizando tabela de dispersão com encadeamento externo, visando operações eficientes de inserção, consulta e remoção, bem como o redimensionamento dinâmico da tabela para manter desempenho consistente.

2.1. Função *minimum*

```
fun minimum(maxHeap: Array<Int>, heapSize: Int): Int
```

Figura 1 – $fun\ minimum$

Organização Estrutural de um Heap Binário

Neste exercício, o objetivo principal foi compreender e explorar a organização interna de um *heap* binário representado como um *array*, bem como a sua aplicação na identificação eficiente do menor elemento entre os nós folhas. Esta representação baseia-se no facto de o *heap* binário ser uma árvore binária quase completa, o que permite uma disposição sequencial e otimizada dos elementos num *array*.

A estrutura do *heap* divide-se em duas zonas distintas. A zona de nós internos corresponde aos elementos localizados entre os índices 0 e $\lfloor heapSize/2 \rfloor - 1$,

representando os nós que possuem, pelo menos, um filho. Por sua vez, a zona de nós folhas inclui os elementos situados entre os índices $\lfloor heapSize/2 \rfloor$ e $heapSize - 1$, e é composta por nós que não possuem descendência. Esta separação reflete a forma sistemática como os elementos são inseridos num *heap* binário, preenchendo os níveis da árvore da esquerda para a direita antes de avançar para o nível seguinte.

Relativamente às propriedades específicas de um *max heap*, destaca-se que o valor máximo está sempre localizado na raiz da árvore, ou seja, na posição 0 do *array*. Além disso, cada nó interno possui um valor superior ou igual ao dos seus dois filhos, garantindo uma hierarquia de valores em ordem decrescente. Em virtude desta propriedade, é possível concluir que o menor valor de um *max heap* só poderá encontrar-se entre os nós folhas, dado que não pode existir nenhum valor inferior na hierarquia descendente.

Para encontrar o menor elemento presente num *max heap*, foi desenvolvida uma função organizada em três fases principais, de forma a garantir uma complexidade temporal linear, proporcional ao número de folhas. Na primeira fase, é determinado o índice correspondente ao primeiro nó folha, utilizando a expressão $firstLeafIndex = heapSize \div 2$. Este cálculo aproveita a regularidade estrutural da árvore para localizar de forma eficiente o início da zona de folhas. Em seguida, realiza-se um percurso seletivo, centrado exclusivamente nos elementos entre os índices *firstLeafIndex* e $heapSize - 1$, evitando a análise dos nós internos, cuja participação no resultado seria redundante. Durante esse percurso, é mantida uma variável auxiliar min, que vai sendo atualizada sempre que se encontra um valor inferior ao atual mínimo.

A análise de complexidade revela que, em termos temporais, a função apresenta complexidade O(n), mesmo considerando que apenas metade do *array* é percorrida (os nós folhas), uma vez que a notação Big-O ignora constantes multiplicativas. Em termos espaciais, a complexidade é O(1), dado que o algoritmo utiliza apenas um número fixo de variáveis auxiliares e não requer estruturas adicionais para a sua execução.

Passamos agora a explicação do exercício seguinte.

2.2 Definição do tipo de dados IntArrayList

Neste exercício tínhamos como objetivo definir um tipo de dados que armazenasse uma lista de k inteiros, onde k é um valor previamente conhecido, e que garanta que as operações que executa tenham uma complexidade O(1). Para além disso, esta estrutura de dados deve seguir a disciplina FIFO (First-In, First-Out) no seu tratamento de dados, o que significa que os elementos são removidos pela ordem de inserção. As operações a desenvolver neste exercício foram:

• *append*(x: *Int*): *Boolean* – Adiciona um inteiro x ao final da lista e retorna *true* se a operação for bem-sucedida e false caso a lista esteja cheia;

• *get*(n: *Int*): *Int*? – Retorna o enésimo elemento da lista ou *null* caso o índice seja inválido;

• *addToAll*(x: *Int*) – Adiciona x a todos os inteiros presentes na lista;

• *remove*(): *Boolean* – Remove o primeiro elemento da lista e retorna *true* se a operação for bem-sucedida e false caso a lista esteja vazia.

Para a realização deste exercício foi usada uma estrutura semelhante a uma fila circular juntamente com algumas *flags* que tornam a complexidade O(1) possível para todas as operações pedidas. A estrutura de tipo dados *IntArrayList*, ao ser baseada numa fila circular, é constituída por um *array* e possui variáveis *head*, *tail* e *size* para que se possa saber e localizar o início, fim e tamanho da fila respetivamente. Apresenta também quatro outras variáveis usadas para uma adição "virtual" a todos os valores da fila para que esta operação tenha uma complexidade O(1), sendo estas as variáveis *change*, *safeChange*, *oldSafeChange* e *newChange*. Esta adição "virtual" é usada ao invés de uma adição individual a todos os elementos da fila pois a adição individual implica uma complexidade linear pois aumenta juntamente com o número de elementos da fila. O seu conceito é fundamenta-se na necessidade de uma complexidade O(1) e para tal, a adição é feita somente quando a operação *get*(n: *Int*) é executada, somando o valor pretendido ao valor requisitado pelo utilizador.
Quando a operação *addToAll*(x: Int*)* é realizada, as variáveis change, *safeChange*, *oldSafeChange* e *newChange* são atualizadas para que somente os valores anteriores à

concretização desta operação sejam afetados por ela. As variáveis change e *newChange* representam os valores a adicionar aos elementos da fila sendo estes o total das adições a executar e a última adição a executar, respetivamente. As variáveis *safeChange* e *oldSafeChange* representam os limites inferiores da fila onde não há adições a realizar e onde somente a última adição é realizada, respetivamente.

2.3 Realização das funções $fun\ splitEvensAndOdds$ e $fun\ <T>\ intersection$

2.3.1 $fun\ splitEvensAndOdds$

```
fun splitEvensAndOdds(list: Node<Int>)
```

Figura 2 - fun splitEvensAndOdds(list: Node < Int >)

Nesta função, é pedido para que para uma determinada lista duplamente ligada com sentinela e circular referenciada em *list*, esta seja reorganizada de maneira que todos os números pares fiquem consecutivos no início da lista. Como a lista é reorganizada, a estrutura de dados utilizada para a resolução desta função foi uma lista circular duplamente ligada com sentinela, o que consiste numa lista onde cada elemento é composto por um atributo *key*, e dois atributos *next* e *previous* que servem como referências para o sucessor e antecessor do elemento, respetivamente. Quanto à composição desta função, a reorganização dos elementos é feita individualmente, revelando uma complexidade O(n) onde n é o tamanho da lista a reorganizar. A reorganização consiste em redirecionar o nó anterior ao nó com um valor par para o nó posterior a este e também ao redireccionamento do nó com um valor par para o início da lista, alterando também o atributo *next* do último elemento da lista circular para que este passe a ser direcionado ao novo começo.

2.3.2 $fun < T > intersection$

```
fun <T> intersection(list1: Node<T>, list2: Node<T>, cmp: Comparator<T>): Node<T>?
```

Figura 3 - $fun < T > intersection$

Para esta função, foi pedido para que dadas duas listas duplamente ligadas, circulares e com sentinela referenciadas por list1 e list2, e ordenadas de modo crescente segundo o comparador *cmp*, seja retornada uma nova lista composta por elementos que pertençam simultaneamente a ambas as listas. Esta lista de retorno é declarada no enunciado que deve ser duplamente ligada, não circular e sem sentinela, ordenada de modo crescente e sem elementos repetidos e deve ainda reutilizar os nós de uma das listas usadas como parâmetros. A complexidade desta função é O(n + m) onde n é a dimensão de list1 e m a dimensão de list2 pois a duração de execução desta função depende de ambas as listas passadas como parâmetros.

Em relação à lógica usada para o retorno da lista pedida, esta foi baseada na reorganização da lista da função anterior. Retirámos de list1 e acrescentámos à lista de retorno os nós com o atributo *value* mutuamente encontrados em ambas as listas, desconectando estes de *list1* e também retirámos os elementos repetidos nestas listas para que a lista de retorno esteja isenta dos mesmos.

Para finalizar, como a lista a retornar não é circular, o atributo *previous* do primeiro nó e o atributo *next* do último nó foram alterados para *null* para alcançar este objetivo.

2.4 Implementação do tipo de dados abstratos MutableMap

Implementação de um *MutableMap* com Tabela de Dispersão e Encadeamento Externo

Neste exercício, foi proposta a implementação de uma estrutura de dados que segue o contrato da interface *MutableMap*<K, V>, utilizando uma tabela de dispersão com encadeamento externo como mecanismo de resolução de colisões. A estrutura é baseada em listas ligadas simples, não circulares e sem a utilização de nós sentinela. Para a função de dispersão, é adotado o método padrão *hashCode*() fornecido por cada chave.

A estrutura implementada visa oferecer desempenho eficiente nas operações típicas de mapas mutáveis, mantendo um bom equilíbrio entre tempo de acesso e consumo de memória.

Componentes Estruturais da Implementação

1. **Interface *MutableMap*<K, V>**
    - Define o conjunto de operações essenciais de um mapa mutável: *put*, *get*, *remove*, entre outras.
    - Contém também a interface aninhada *MutableEntry*<K, V>, utilizada para representar os pares chave-valor armazenados na estrutura.
2. **Classe *HashMap*<K, V>**
    - Implementa a interface *MutableMap*<K, V>.
    - Utiliza um *array* de listas ligadas (*table*) como estrutura interna.
    - Cada índice do *array* representa um *bucket*, que armazena os elementos que colidem para o mesmo valor de dispersão.
    - Mantém duas variáveis principais: *size* (número total de elementos) e *capacity* (tamanho da tabela).

Resolução de Colisões: Encadeamento Externo

Quando múltiplas chaves produzem o mesmo valor de *hashCode*() % *capacity*, os pares são armazenados sequencialmente numa lista ligada associada a esse índice do *array*. A ausência de nós sentinela obriga a um controlo rigoroso dos apontadores durante as operações de inserção e remoção.

Desempenho e Análise de Complexidade

- **Inserção (put)**

- **Caso médio**: O(1), assumindo distribuição uniforme das chaves.
- **Pior caso**: O(n), quando todas as chaves são mapeadas para o mesmo bucket (colisões extremas).

- **Consulta (get)**

- Possui complexidade semelhante à da inserção, dependendo da profundidade da lista ligada associada ao bucket.

- **Remoção (remove)**

- Também depende do comprimento da lista dentro do bucket, mantendo complexidade O(1) no caso médio.

- **Redimensionamento (expand)**

- Quando o número de elementos ultrapassa o limite do fator de carga, o array é duplicado e todos os elementos são reinseridos de acordo com a nova capacidade.
- Esta operação tem complexidade O(n), sendo desencadeada esporadicamente.

Fator de Carga

- Valor utilizado: 0.75.
- Determina o limite a partir do qual ocorre o redimensionamento automático da tabela.
- Este valor representa um compromisso entre uso eficiente de memória e prevenção de colisões excessivas.

Com esta etapa concluída, passamos a próxima fase.

3 Operações entre coleções de pontos no plano

Nesta secção descrevemos o desenvolvimento de uma aplicação destinada à realização de operações entre coleções de pontos no plano, representadas em ficheiros de texto com extensão .co. A aplicação, denominada ProcessPointsCollections, permite efetuar três operações principais entre os pontos descritos em dois ficheiros: união, interseção e diferença. Cada ponto é identificado por um nome único e pelas suas coordenadas X e Y.

O processo de execução é iniciado com o carregamento dos dois ficheiros de entrada através do comando load. Após este carregamento, as operações solicitadas são realizadas com base numa estrutura de dados que garante uma consulta eficiente, e os resultados são gravados num novo ficheiro, também no formato .co, respeitando o formato definido (ignorando linhas de comentário c e de problema p).

Foram implementadas duas versões distintas da aplicação:

- A primeira versão recorre a estruturas fornecidas pela Kotlin Standard Library, nomeadamente HashMap, para armazenar e operar sobre os dados de forma eficiente e legível.
- A segunda versão utiliza uma estrutura de dados personalizada — uma tabela **de** dispersão **(**hash map**)** desenvolvida manualmente, sem recurso a bibliotecas da linguagem, com o objetivo de consolidar o entendimento dos mecanismos internos associados à gestão de dados dispersos.

Análise do problema

Manipulação e Comparação de Coleções de Pontos

O problema abordado neste exercício consiste na manipulação e comparação de coleções de pontos no plano, em que cada ponto é representado por um identificador e por duas coordenadas cartesianas, X e Y. As coleções são armazenadas em ficheiros de texto com extensão `.co`, que, além dos dados relevantes, podem incluir linhas não pertinentes, como comentários ou especificações do problema, as quais devem ser devidamente ignoradas durante o processamento.

A aplicação desenvolvida tem como finalidade realizar três operações principais sobre estas coleções de pontos. A primeira operação corresponde à união das coleções, implicando a reunião de todos os pontos distintos presentes em pelo menos um dos ficheiros. A segunda operação corresponde à interseção, ou seja, à identificação dos pontos comuns a ambas as coleções. Por fim, a operação de diferença visa determinar os pontos que pertencem à primeira coleção, mas que não se encontram na segunda.

Para que estas operações possam ser executadas de forma eficaz, é necessário garantir uma leitura eficiente dos ficheiros de entrada, filtrando as linhas relevantes e ignorando adequadamente as que não contribuem para o processamento. Além disso, os pontos devem ser representados internamente através de estruturas de dados que permitem acesso rápido e comparações eficientes, assegurando assim a integridade e o desempenho das operações realizadas. Finalmente, os resultados de cada operação devem ser escritos num novo ficheiro, respeitando o formato de saída esperado, de modo a garantir a sua reutilização ou análise posterior com base nos mesmos critérios de estruturação.

3.4 Estruturas de Dados

Nesta secção iremos mostrar algumas dessas estruturas de dados usadas:

HashMap<K, V>: Armazenar e operar sobre pontos.

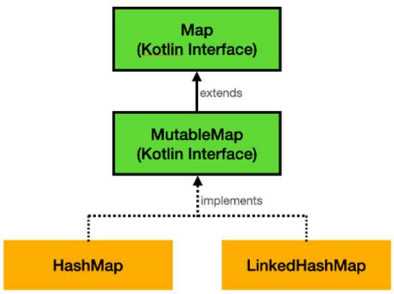

Figura 4 – HashMap

Array<T> : Base interna do HashMap

Figura 5 – *Array*

Linked List: Colisões dentro do Hashmap

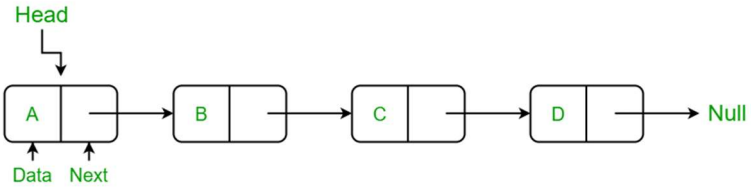

Figura 6 - *Linked List*

Com as ilustrações feitas, podemos passar ao próximo tópico.

Algoritmos e análise da complexidade

Neste tópico iremos apresentar os principais algoritmos e os seus graus de complexidade.

Função readPoints(filename: String)

- **Complexidade Temporal**: O(n), onde *n* é o número de linhas (pontos) no ficheiro.
- **Complexidade Espacial**: O(n), pois armazena todos os pontos num HashMap.

Função writePoints(points: HashMap<String, Point>, filename: String)

- **Complexidade Temporal**: O(n), com *n* igual ao número de pontos no mapa.
- **Complexidade Espacial**: O(n), para construir a string completa a escrever no ficheiro.

Função union(map1, map2)

- **Complexidade Temporal**: O(n + m), onde *n* e *m* são o número de pontos em map1 e map2.
- **Complexidade Espacial**: O(n + m), pois o mapa resultante pode conter todos os pontos dos dois mapas.

Função intersection(map1, map2)

- **Complexidade Temporal**: O(n), onde *n* é o número de elementos em map1.
- **Complexidade Espacial**: O(min(n, m)), pois no máximo podem coincidir todos os pontos.

Função difference(map1, map2)

- **Complexidade Temporal**: O(n), onde *n* é o número de elementos em map1.
- **Complexidade Espacial**: O(n), no pior caso (quando nenhum ponto de map1 está em map2).

Função forEach(action) da HashMap

- **Complexidade Temporal**: O(n), onde *n* é o número total de elementos no mapa.
- **Complexidade Espacial**: O(1), pois apenas são usadas variáveis temporárias para iteração.

4 Avaliação Experimental

Com o objetivo de complementar a análise teórica realizada anteriormente, foi conduzida uma **avaliação experimental** dos algoritmos desenvolvidos para a leitura, escrita e manipulação de coleções de pontos 2D.

Os testes foram realizados em ambiente controlado, utilizando uma máquina com as seguintes características:

- **Processador:** *Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz*
- **RAM:** 8 GB
- **Sistema Operativo:** *Windows 11 Home*

Foram implementados e testados os seguintes algoritmos: Load, Intersection, Union e Difference. Cada algoritmo foi executado para os diferentes volumes de dados mencionados, sendo os tempos de execução registados em milissegundos. Os resultados obtidos são apresentados sob forma de gráfico. A Figura **7** ilustra graficamente essas diferenças de desempenho.

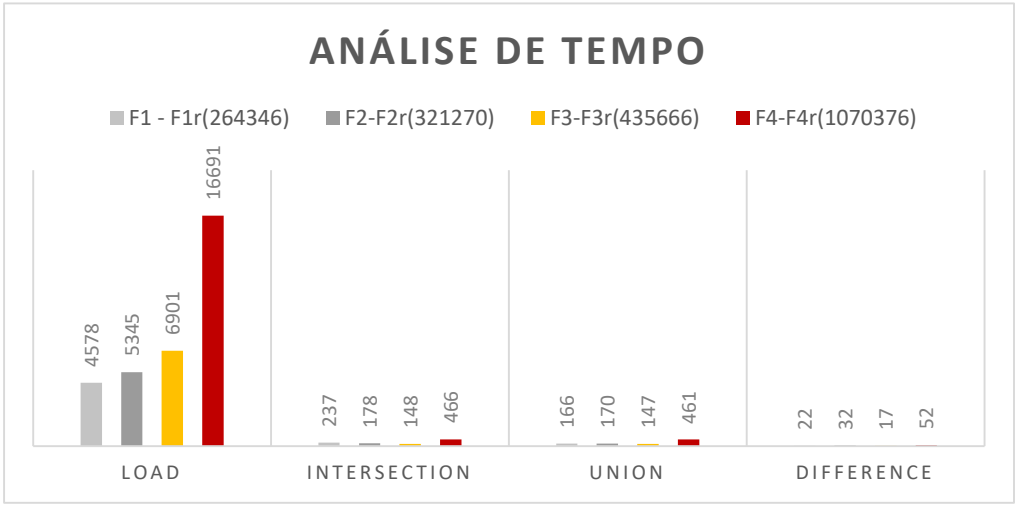

Figura 7 - Comparação dos tempos de execução dos vários algoritmos.

Com estes dados demonstramos, vamos abordar todos os resultados obtidos:

A operação *load* é a mais demorada e sensível à dimensão dos dados. Isto indica que o tempo de leitura e inserção dos dados na estrutura (como o *MutableMap*) escala linearmente ou pior, dependendo da eficiência da estrutura usada;

Os tempos de execução das operações *intersection*, *union* e *difference* são significativamente inferiores ao tempo da operação *load*, mesmo nos ficheiros de maior dimensão. verifica-se, aliás, que para ficheiros de tamanhos semelhantes, o tempo destas operações tende a diminuir à medida que o ficheiro cresce. este comportamento sugere que, uma vez carregados os dados para memória e devidamente organizados (por exemplo, através de tabelas de dispersão eficientes), as operações sobre os conjuntos são extremamente rápidas (geralmente com complexidade **O(n)** ou até melhor) beneficiando da eficácia da estrutura de dados utilizada.

5 Conclusões

O presente trabalho focou-se na resolução eficiente do problema da manipulação de coleções de pontos no plano, com ênfase nas operações de união, interseção e diferença. Para esse fim, foi concebida e implementada uma estrutura de dados baseada em *MutableMap*, utilizando uma combinação de tabelas de dispersão com encadeamento externo através de listas ligadas, com o objetivo de otimizar as operações de inserção, remoção e pesquisa.

A aplicação desenvolvida, *ProcessPointsCollections*, permitiu validar a implementação dos algoritmos propostos e avaliar a sua eficiência mediante testes experimentais. Os resultados obtidos demonstraram que a estrutura adotada apresenta um desempenho estável e eficaz, mesmo em contextos com grandes volumes de dados, evidenciando a sua adequação à natureza do problema.

Referências

[1]https://2425moodle.isel.pt/pluginfile.php/1271585/mod_folder/content/0/AED%20-%20parte6%20-%20Amontoados%20Bin%C3%A1rios%20%28Heaps%29.pdf?forcedownload=1

[2] https://2425moodle.isel.pt/pluginfile.php/1271585/mod_folder/content/0/AED%20-%20parte7%20-%20Filas%20Priorit%C3%A1rias%20%28Heaps%29.pdf?forcedownload=1

[3] https://2425moodle.isel.pt/pluginfile.php/1271585/mod_folder/content/0/AED%20-%20parte10%20-%20Tipo%20de%20Dados%20Abstratos.pdf?forcedownload=1

[4] https://2425moodle.isel.pt/pluginfile.php/1271585/mod_folder/content/0/AED%20-%20parte11%20-%20Listas%20Ligadas.pdf?forcedownload=1

[5] https://2425moodle.isel.pt/pluginfile.php/1271585/mod_folder/content/0/AED%20-%20parte13%20-%20Iteradores.pdf?forcedownload=1