Licenciatura em Engenharia Informática e de Computadores
Licenciatura em Engenharia Informática, Redes e Telecomunicações

ALGORITMOS E ESTRUTURAS DE DADOS
(SEGUNDA SÉRIE DE PROBLEMAS)

Trabalho realizado pelo **Grupo 16**, constituído por:

(52718) Alexandre Miguel Pérides Gonçalves da Silva

(52599) Duarte Afonso Palma Reis Rodrigues

Docente: Maria Graça
Algoritmos e Estruturas de Dados 2024 / 2025 VERÃO
18 de maio de 2025

Conteúdo

1 Introdução

Este trabalho corresponde à **segunda série de problemas** da unidade curricular, com entrega a 18 de maio de 2025. O objetivo é desenvolver soluções eficientes para problemas com estruturas de dados como listas ligadas e tabelas de dispersão. Na primeira secção deste trabalho, são propostos vários exercícios com o intuito de consolidar conceitos fundamentais:

- Identificação do menor elemento em *max-heap*.
- Definição e implementação da estrutura de dados `IntArrayList`, respeitando a disciplina FIFO e garantindo operações em tempo constante.
- Manipulação de listas duplamente ligadas, nomeadamente:
  - Reorganização da lista para agrupar os elementos pares.
  - Cálculo da interseção entre duas listas ordenadas, reaproveitando nós e eliminando duplicados.
- Implementação de uma *hash table* com encadeamento externo, através da interface `MutableMap<K, V>`, incluindo operações de inserção, consulta, iteração e redimensionamento.

Na segunda secção, é proposta a construção de uma aplicação que permite realizar operações entre coleções de pontos no plano (x, y). As operações incluem:

- União de pontos dos ficheiros de entrada, eliminando duplicados.
- Interseção entre os conjuntos de pontos.
- Diferença entre os conjuntos, considerando apenas os pontos exclusivos do primeiro ficheiro.

Para isso, os dados são lidos de ficheiros no formato `.co`, e armazenados eficientemente com recurso a uma tabela de dispersão. A aplicação deve ser desenvolvida em duas versões: uma recorrendo às bibliotecas padrão da linguagem Kotlin, e outra utilizando a estrutura `MutableMap<K, V>` implementada anteriormente.

Finalmente, este relatório inclui também uma análise comparativa do desempenho das duas abordagens propostas, acompanhada de gráficos ilustrativos e tem todos os **códigos de resolução e funções auxiliares** presentes no **repositório do grupo**.

Este trabalho sucede à primeira série de problemas, onde foram abordados algoritmos elementares, criação de partições ordenadas a partir de ficheiros e análise assintótica de algoritmos. A sequência lógica entre os dois trabalhos visa consolidar e aplicar os conhecimentos adquiridos, através da implementação de soluções práticas e eficientes.

2 Parte I - Exercícios

Nesta secção irão ser apresentadas as respostas dos exercícios 1 a 4, tal como requisitadas na série.

2.1 Exercício 1 - minimum

Implemente a função:

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

Esta função retorna o menor elemento do max heap representado pelo parâmetro maxHeap. A dimensão do heap é indicada por heapSize. O algoritmo deve tirar partido das propriedades de um max heap na procura do menor elemento.

Descrição da Solução:
Num max heap, o maior elemento encontra-se no índice 0, ou raiz, enquanto o menor elemento está garantidamente numa das folhas, pois todos os nós internos são maiores que os seus descendentes.
Considerando que o heap está representado num array, os índices das folhas começam a partir de heapSize / 2 até heapSize - 1.
Assim, a forma mais eficiente de encontrar o menor elemento consiste em percorrer apenas os elementos neste intervalo, evitando visitar os nós internos que sabemos que são necessariamente maiores.

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

2.2 Exercício 2 - Estrutura de Dados IntArrayList

Objetivo:
Implementar uma estrutura com capacidade fixa k que armazena inteiros em política FIFO, garantindo $O(1)$ em todas as operações.

Solução:
Usa-se um **array circular** para inserções e remoções eficientes, e um campo `offset` para aplicar somas a todos os elementos sem percorrer o array.

- **append(x)** Insere no índice `end` (ajustado com `offset`) e avança circularmente. Sem deslocamentos: $O(1)$.
- **remove()** Avança `start` com módulo k e reduz o tamanho. O valor será sobrescrito: O(1).
- **get(n)** Calcula índice lógico com `(start + n) % k` e aplica `offset`. Leitura direta: $O(1)$.
- **addToAll(x)** Acumula x no `offset`, sem alterar o array. Efeito lógico imediato em $O(1)$.
- **iterator()** Percorre os elementos FIFO com `offset` aplicado. Iteração constante por elemento.

Conclusão:
A estrutura garante eficiência com array circular e `offset`, evitando cópias e percursos redundantes.

2.3 Exercício 3.1 - splitEvensAndOdds

Objetivo: Dada uma lista duplamente ligada circular com nó sentinela, reorganizar a lista de forma que todos os números pares fiquem consecutivos no início. A ordem dos pares e ímpares pode ser alterada, mas os pares devem ficar todos juntos no início.

Descrição da Solução: Percorremos a lista desde o primeiro nó após o sentinela até voltarmos ao próprio sentinela. Sempre que encontramos um número par, removemos o nó da sua posição atual e inserimo-lo imediatamente após o sentinela, agrupando assim todos os pares no início da lista. O procedimento respeita a estrutura circular e não cria novos nós.

Justificação:

- `removeNode(node)`: Remove um nó da lista, ajustando os ponteiros do nó anterior e do seguinte para manter a integridade da lista.

  Exemplo:
  [Sentinel $\leftrightarrow A \leftrightarrow B \leftrightarrow C \leftrightarrow$ Sentinel]
  Após remover o nó B:
  [Sentinel $\leftrightarrow A \leftrightarrow C \leftrightarrow$ Sentinel]

- `addAfter(reference, node)`: Insere um nó logo após o nó de referência (por exemplo, o sentinela), ajustando os ponteiros correspondentes.

  Exemplo:
  [Sentinel $\leftrightarrow A \leftrightarrow C \leftrightarrow$ Sentinel]
  Após adicionar o nó B a seguir ao sentinela:
  [Sentinel $\leftrightarrow B \leftrightarrow A \leftrightarrow C \leftrightarrow$ Sentinel]

- Durante a travessia da lista, o próximo nó (`next`) é sempre guardado antecipadamente antes de qualquer modificação. Isto evita erros causados por alterações na estrutura da lista enquanto esta está a ser percorrida.

Complexidade: $O(n)$, onde n é o número de nós (excluindo o sentinela). Cada nó é visitado e eventualmente movido uma única vez.

Link código:
fun splitEvensAndOdds (list : Node < Int >)

2.4 Exercício 3.2 - intersection

Objetivo: Dadas duas listas duplamente ligadas, circulares e com sentinela, ordenadas de forma crescente, determinar os elementos comuns a ambas. Os elementos devem ser removidos das listas originais e reutilizados para formar uma nova lista linear (não circular, sem sentinela), ordenada e sem repetições.

Descrição da Solução:

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

Na resolução, usamos dois ponteiros para percorrer as listas ao mesmo tempo, tal como na fusão de listas ordenadas. Quando encontramos elementos iguais:

- Removemos os nós correspondentes das duas listas.
- Reutilizamos um dos nós (de `list1`) e adicionamo-lo ao fim da nova lista.
- Saltamos quaisquer duplicados subsequentes nas listas.

Justificação:

- A travessia em paralelo e ordenada garante que os elementos comuns são encontrados em $O(n+m)$, onde n e m são os tamanhos das listas.
- A utilização de nós de `list1` permite cumprir o requisito de não criar novos nós.
- Ao evitar inserir duplicados consecutivos, garantimos que a nova lista não terá repetições.

Complexidade: $O(n+m)$ — cada lista é percorrida no máximo uma vez.

Foi também criado um main, que serve para testar as funções principais, utilizando funções auxiliares para facilitar a criação e visualização das listas. A randomIntList gera listas aleatórias, createCircularListWithSentinel transforma listas em listas duplamente ligadas circulares com sentinela, e printCircularList e printLinearList imprimem as listas no formato adequado. Estas funções tornam os testes no main mais simples e o código mais legível.

2.5 Exercício 4 - Estrutura de Dados HashMapCustom

Objetivo:
Implementar uma estrutura de dados `HashMapCustom<K, V>` baseada em uma tabela de dispersão com encadeamento externo (listas ligadas), suportando as principais operações do tipo de dados abstrato `MutableMap<K, V>`.
A estrutura deve redimensionar-se automaticamente conforme necessário e garantir uma iteração completa sobre os pares armazenados.

Descrição da Solução:
A tabela de dispersão foi implementada como um array de baldes (`buckets`), onde cada posição do array contém uma lista ligada de pares chave-valor, permitindo o tratamento de colisões por encadeamento.
O índice de cada chave é obtido através da função `hashCode()` da chave, seguido de um mapeamento modular ao tamanho atual da tabela.
O redimensionamento da tabela é disparado automaticamente sempre que o número de elementos ultrapassa o produto `capacidade × fator de carga`.
Neste processo, todos os elementos são re-hashados e distribuídos numa nova tabela com o dobro da capacidade anterior.

Componentes principais:

- **Tabela de buckets:** array de listas ligadas, cada uma contendo pares chave-valor;
- **Load Factor:** valor fixo, que determina a percentagem de ocupação permitida antes do redimensionamento;
- **Contador de elementos:** usado para determinar quando expandir a tabela;
- **Iterador:** percorre todos os baldes e, dentro de cada um, os nós da respetiva lista.

Operações Implementadas:

- `put(key, value)` Insere ou substitui o valor associado à chave. No caso de colisão, a chave é inserida ou substituída na lista ligada correspondente.
- `get(key)` Procura a chave na lista do balde correspondente e retorna o valor associado, se existir.
- `remove(key)` Remove a chave (caso exista) da lista correspondente, mantendo a integridade da lista ligada.
- `containsKey(key)`, `containsValue(value)` Verificam a presença de uma chave ou valor na estrutura.
- `clear()` Esvazia completamente a tabela, reinicializando os baldes e o contador de elementos.
- `iterator()` Fornece um iterador que percorre todos os pares chave-valor da estrutura, mesmo após operações de inserção, remoção ou expansão.

Gestão de Colisões:

Todas as colisões são resolvidas por encadeamento externo, usando listas ligadas simples.

Redimensionamento:

Quando a tabela atinge o seu limite de carga, é criada uma nova tabela com o dobro da capacidade.

Todos os elementos da tabela anterior são reinseridos na nova tabela, com base no novo cálculo de índices.

Esta operação é feita fazendo recurso às funções expand() e reinsertAll() e garante a manutenção de um bom desempenho médio nas operações de inserção e consulta.

Complexidade Temporal:

- `put`, `get`, `remove` – Tempo médio $O(1)$;
- `containsKey`, `containsValue`, `iterator` – $O(n)$;
- `expand` – $O(n)$, executado apenas em redimensionamentos e apenas uma vez para cada par Key-Value.

3 Parte II — Problema

3.1 Operações entre Conjuntos de Pontos 2D

3.1.1 Objetivo e Descrição da Solução

Objetivo:
O objetivo deste trabalho é desenvolver uma aplicação capaz de realizar operações de conjuntos — união, interseção e diferença — sobre dois ficheiros que contêm pontos no plano 2D. A aplicação deverá ler os ficheiros, armazenar os pontos de forma eficiente e escrever os resultados num ficheiro de saída no formato `.co`.

Descrição da Solução:
Para este problema, foram implementadas duas versões distintas que partilham a mesma lógica para o processamento e armazenamento dos dados, diferenciando-se apenas na estrutura de dados usada para representar o mapa associativo:

- Ambas as implementações utilizam um mapa associativo do tipo `Map<Point, Int>`, onde cada ponto, definido pela sua coordenada (x, y), é associado a um inteiro que indica a origem do ponto. Este inteiro funciona como uma flag binária, com o seguinte significado:
  - 1 (binário 01): ponto presente somente no primeiro ficheiro;
  - 2 (binário 10): ponto presente somente no segundo ficheiro;
  - 3 (binário 11): ponto presente em ambos os ficheiros.
- A primeira implementação (**Implementation1**) utiliza o `HashMap` da biblioteca padrão Kotlin para o mapa associativo.
- A segunda implementação (**Implementation2**) usa uma estrutura personalizada, denominada `HashMapCustom<Point, Int>`, implementada com encadeamento separado para controlo detalhado da dispersão.

Os ficheiros `.co` são lidos linha a linha e, para cada ponto encontrado, o mapa é atualizado, combinando as flags com a operação bitwise `OR` para indicar corretamente a origem do ponto.

3.1.2 Componentes e Operações Suportadas

Componentes Principais:

Os principais componentes da solução são:

- `Point`: estrutura que representa um ponto no plano 2D através das suas coordenadas x e y.
- `PointList`: estrutura auxiliar para manipulação de coleções de pontos.
- `PointUtils` e `PointUtils2`: módulos utilitários para leitura e escrita dos ficheiros, cada um adaptado à respetiva implementação.
- `HashMapCustom<K, V>`: implementação personalizada da tabela de dispersão com encadeamento separado.
- `Main.kt`: módulo principal que fornece a interface interativa para receber comandos do utilizador.

Operações Suportadas:

A aplicação suporta três operações principais entre os conjuntos de pontos:

- `union` (união): devolve todos os pontos que aparecem em pelo menos um dos ficheiros, correspondendo aos pontos com flags 1, 2 ou 3.
- `intersection` (interseção): devolve apenas os pontos comuns aos dois ficheiros, ou seja, com flag 3.
- `difference` (diferença): devolve os pontos exclusivos do primeiro ficheiro, com flag 1.

3.1.3 Lógica de Implementação e Complexidade

Lógica de Implementação:

Ao inserir cada ponto no mapa associativo, a flag que indica a origem é atualizada com a operação bitwise `OR`, permitindo indicar múltiplas origens para o mesmo ponto quando aplicável. Para executar cada operação entre conjuntos, a aplicação filtra os pontos cujas flags satisfazem as condições correspondentes.
Na primeira implementação, a estrutura usada é a `HashMap` da biblioteca padrão Kotlin, beneficiando-se das otimizações da JVM.
A segunda implementação oferece uma versão personalizada, útil para a análise detalhada e ensino da estrutura de dados subjacente.

Complexidade Temporal:

As operações `union`, `intersection` e `difference` têm complexidade temporal linear na soma do número de pontos dos dois ficheiros, ou seja, $O(n+m)$, onde n e m são os números de pontos nos ficheiros 1 e 2, respetivamente. A leitura e escrita dos ficheiros têm também complexidade $O(n)$ para cada ficheiro, proporcional ao número de pontos.

Exemplificação:
Para demonstração do resultado das operações, consideraram-se os seguintes ficheiros de exemplo:

file1.co

```
v a 1 1
v b 2 2
v c 3 3
```

file2.co

```
v d 2 2
v e 3 3
v f 4 4
```

Exemplificação:

No ficheiro `file1.co` encontram-se três pontos: `a` em (1,1), `b` em (2,2) e `c` em (3,3). O ficheiro `file2.co` contém os pontos `d` e `e` (presentes também no primeiro ficheiro) e o ponto `f` em (4,4).

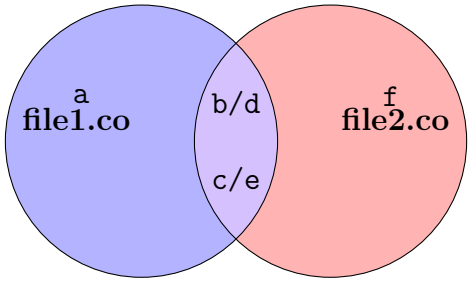

Diagrama de Venn exemplificativo

Cores:

- Blue → Chaves com value 1 (file1)
- Red → Chaves com value 2 (file2)
- Rosa → Chaves com value 3 (ambos os ficheiros)

Assim, os resultados esperados para as operações são:

- `union`: deve conter todos os pontos presentes em pelo menos um ficheiro, ou seja, `a`, `b`, `c`, `d`, `e` e `f`.
- `intersection`: deve conter apenas os pontos comuns a ambos os ficheiros, ou seja, `b/d` e `c/e`.
- `difference`: deve conter apenas os pontos exclusivos do primeiro ficheiro, ou seja, `a`.

Este exemplo simples permite confirmar o correto funcionamento da lógica das flags e da filtragem para as operações de conjuntos, garantindo a robustez da solução.

3.1.4 Análise Comparativa das Implementações

Análise Comparativa com Kotlin Standard Library HashMap:

Ambas as implementações apresentaram resultados corretos e idênticos, confirmando a validade da abordagem adotada. No entanto, existem diferenças relevantes entre a `HashMapCustom` e a `HashMap` da Kotlin Standard Library que influenciam a escolha da estrutura em diferentes contextos.

- **Performance:** A `HashMapCustom` mostrou um bom desempenho em operações de inserção e consulta, mesmo com grandes volumes de dados. Contudo, apresenta um desempenho inferior à `HashMap` da Kotlin, que beneficia de otimizações internas avançadas específicas da biblioteca padrão.
- **Consumo de Memória:** A implementação personalizada consome mais memória devido ao uso de nós explícitos em listas ligadas e à falta de otimizações de armazenamento presentes na biblioteca padrão Kotlin.
- **Funcionalidades:** A `HashMap` da Kotlin oferece funcionalidades avançadas, como iteração eficiente por entradas, chaves e valores, além de métodos otimizados para manipulação dos dados. A `HashMapCustom` é limitada às operações essenciais, focando no controlo detalhado e simplicidade.
- **Flexibilidade e Usabilidade:** A `HashMapCustom` permite maior controlo sobre a estrutura interna e é adequada para fins pedagógicos e de experimentação, permitindo uma melhor compreensão dos mecanismos de dispersão e gestão de colisões. Em contraste, a `HashMap` da Kotlin é mais eficiente e é uma melhor opção devido à sua robustez e desempenho.
- **Complexidade de Implementação:** A implementação customizada requer maior esforço de desenvolvimento e manutenção, enquanto a utilização da biblioteca padrão simplifica o desenvolvimento e reduz a probabilidade de erros.

Em resumo, para aplicações práticas que exigem máxima eficiência e simplicidade, a `HashMap` da Kotlin Standard Library é a escolha recomendada. Por outro lado, a `HashMapCustom` é uma ferramenta valiosa para aprendizagem, análise detalhada e personalização das estruturas de dados subjacentes.

4 Avaliação Experimental - Análise

Análise de Desempenho com Base em Gráficos:

Para avaliar empiricamente o desempenho das duas implementações — `Implementation1` com `HashMap` da Kotlin Standard Library e `Implementation2` com a `HashMapCustom` — foram realizados testes com múltiplos pares de ficheiros `.co` de diferentes dimensões, medindo o tempo total de execução das operações de `union`, `intersection` e `difference`.

Os resultados de dois desses testes foram registados e representados graficamente, permitindo observar tendências de desempenho à medida que o número de pontos aumenta.

Observações retiradas dos gráficos:

- **Diferença de desempenho crescente:** À medida que o tamanho dos ficheiros aumenta, a diferença entre os tempos de execução das duas implementações torna-se mais pronunciada. Isto evidencia as otimizações internas da `HashMap` padrão, como melhor gestão da dispersão e alocação de memória.
- **Operações mais exigentes:** A operação `union` é consistentemente a mais pesada em tempo de execução, por envolver todos os elementos dos dois ficheiros. Por contraste, a operação `difference` tende a ser mais rápida, pois depende apenas da análise de um subconjunto dos dados (os exclusivos do primeiro ficheiro).
- **Consistência dos resultados:** Em todos os testes, os resultados das operações foram equivalentes em ambas as implementações, reforçando a correção funcional da `HashMapCustom`, mesmo com desempenho inferior.
- **Overhead inicial da implementação personalizada:** Para ficheiros pequenos, o overhead da `HashMapCustom` é mais evidente, dado que o custo de criar e gerir as listas ligadas se torna comparativamente maior face ao tempo total de execução.

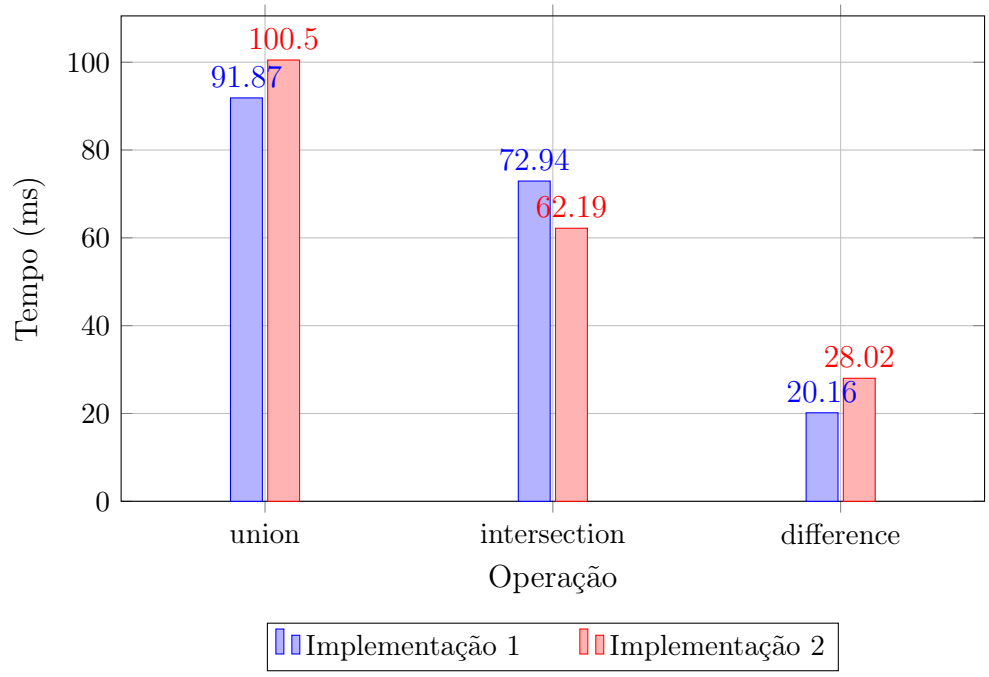

Figura 1: Comparação dos tempos (em ms) entre as duas implementações para cada operação. File 1 = 7 120 KB, File 2 = 70 606 KB

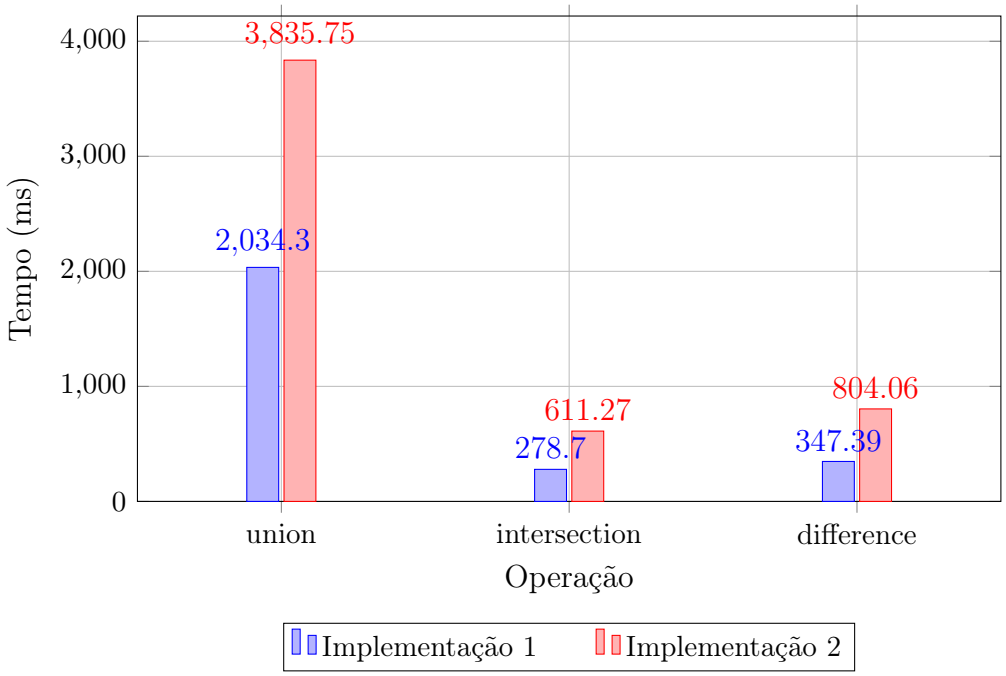

Figura 2: Comparação dos tempos (em ms) entre as duas implementações para cada operação. File 1 = 70 606 KB, File 2 = 1 198 540 KB

Conclusão:

A `HashMapCustom` é uma solução válida e interessante, especialmente para entender como funcionam as tabelas de dispersão com encadeamento separado.
No entanto, a `HashMap` da biblioteca padrão oferece vantagens claras em termos de desempenho e otimizações automáticas. Por isso, é mais indicada para aplicações reais que lidam com grandes volumes de dados.

Os testes mostraram que a implementação padrão aproveita melhorias importantes, como um tratamento eficiente de colisões, rehashing dinâmico e uma gestão otimizada da memória.
Essas otimizações são difíceis de implementar manualmente sem um grande esforço.

Por outro lado, a `HashMapCustom` é útil em contextos educativos para ilustrar conceitos básicos, ou em casos muito específicos onde é necessário controlar detalhadamente o comportamento da estrutura de dados.

Por fim, recomenda-se usar a biblioteca padrão em ambientes de produção que exigem alta performance e escalabilidade.
A implementação customizada, por sua vez, serve melhor como uma ferramenta de aprendizagem e experimentação.

5 Teste das soluções

Testes das Soluções:

Todas as implementações desenvolvidas foram rigorosamente testadas utilizando tanto os ficheiros de testes fornecidos pelos professores como um conjunto adicional de ficheiros criados por nós.

Os testes abrangeram diferentes dimensões e tipos de ficheiros `.co`, incluindo dois ficheiros criados especificamente para automatizar a verificação com outputs previstos.
Estes ficheiros foram utilizados nos testes automatizados presentes nos ficheiros `Implementation1Test` e `Implementation2Test`, permitindo validar rapidamente a correção funcional das operações de `union`, `intersection` e `difference`.

Conteúdo dos ficheiros de teste:

Test1.co	Test2.co
v 1 1 2	v 2 2 3
v 2 2 3	v 4 5 6
v 3 4 5	v 5 7 8

Outputs Esperados das Operações:

Operação	Resultado Esperado (coordenadas)
`union`	(1,2), (2,3), (4,5), (5,6), (7,8)
`intersection`	(2,3)
`difference`	(1,2), (4,5)

Todas as implementações passaram com sucesso a totalidade dos testes, demonstrando a fiabilidade e o bom funcionamento do projeto.