Instituto Superior de Engenharia de Lisboa

Algoritmo e Estrutura de dados

Relatório da Segunda Série de Problemas

Docente

Prof. Nuno Leite

Aluna

53255 Magali dos Santos Leodato

Índice

Introdução

Inicialmente, foi desenvolvido um algoritmo para encontrar o menor elemento de um max heap, aproveitando as propriedades da estrutura para evitar varreduras desnecessárias. Em seguida, foi criada uma estrutura personalizada `IntArrayList` com disciplina FIFO, capaz de realizar inserções, remoções e atualizações com complexidade constante — um exercício importante de otimização de operações básicas.

O trabalho também envolveu o domínio de listas duplamente ligadas, circulares e com sentinela. Foram propostas duas funções: uma para reorganizar os elementos de acordo com sua paridade e outra para calcular a interseção entre duas listas ordenadas, reutilizando nós e eliminando repetições. Essas tarefas exigiram uma compreensão precisa da manipulação de ponteiros em estruturas dinâmicas.

Posteriormente, foi implementada uma tabela de dispersão com encadeamento externo (`MutableMap<K, V>`), estruturada para armazenar pares chave-valor com acesso eficiente e controle sobre colisões. Esta estrutura foi essencial para a construção da aplicação `ProcessPointsCollections`, cujo objetivo é carregar ficheiros de pontos e realizar operações clássicas de conjuntos (união, interseção, diferença), mantendo os dados organizados e sem duplicações.

Por fim, foi realizada uma avaliação experimental das operações da aplicação, considerando ficheiros de entrada com diferentes tamanhos. Os tempos de execução foram medidos e analisados com o objetivo de validar empiricamente a eficiência das soluções implementadas.

Este trabalho representa, portanto, uma integração entre teoria e prática, proporcionando uma base sólida para a resolução de problemas computacionais reais e preparando o estudante para desafios mais complexos no desenvolvimento de software eficiente.

Exercício 1: Encontrar o menor elemento de um Max Heap

O que é um Max Heap:

- Um **Max Heap** é uma **árvore binária completa** onde cada **nó pai é maior ou igual aos seus filhos**.
- Isso significa que o maior elemento está sempre na **raiz** (índice 0 do array).
- Porém, **o menor elemento nunca estará no início**: ele estará **em uma folha**, ou seja, um nó sem filhos.

Abordagens para Encontrar o Menor Elemento:

1. Busca Exaustiva nas Folhas:

- **Ideia:** Como o menor elemento está garantidamente em uma folha, podemos simplesmente identificar todos os nós folha do heap e, em seguida, percorrer esses nós para encontrar o menor valor.
- **Como identificar as folhas em um array que representa o heap:** Em um heap representado por um array de tamanho `n`, os nós folha começarão a partir do índice
- **Passos:**
    1. Determine o índice do primeiro nó folha.
    2. Iterar do índice do array.
    3. Mantenha o controle do menor elemento encontrado durante a iteração.
- **Complexidade:** No pior caso, metade dos nós podem ser folhas, então a complexidade desta abordagem é O(n).

2. Extração Mínima Repetida (Destrutiva):

- **Ideia:** Embora um Max Heap não seja eficiente para encontrar o mínimo diretamente, podemos realizar operações de extração do máximo repetidamente até que reste apenas um elemento (que será o mínimo) ou até encontrarmos um elemento que seja menor que todos os outros que foram extraídos. No entanto, essa abordagem é destrutiva para a estrutura do heap original.
- **Passos (se quisermos o menor e destruir o heap):**
    1. Se o heap tiver apenas um elemento, ele é o mínimo.
    2. Caso contrário, extraia o elemento máximo repetidamente.
    3. O último elemento restante será o menor.
- **Complexidade:** Extrair o máximo de um heap de tamanho k leva O(logk). Para encontrar o mínimo, no pior caso, teríamos que extrair todos os elementos, resultando em uma complexidade de O(nlogn). Esta não é uma abordagem eficiente se você precisar preservar o heap.

3. Manutenção de um Min Heap Auxiliar (Se Modificações Forem Permitidas):

- **Ideia:** Se você puder modificar a estrutura ou manter informações adicionais, uma abordagem seria manter um Min Heap auxiliar contendo todos os elementos do Max Heap. O menor elemento do Max Heap seria então a raiz do Min Heap auxiliar.

- **Passos:**
  1. Criar um Min Heap vazio.
  2. Inserr todos os elementos do Max Heap no Min Heap.
  3. O menor elemento do Max Heap estará agora na raiz do Min Heap (índice 0).
- **Complexidade:** Inserir n elementos em um Min Heap leva O(nlogn), e encontrar o mínimo no Min Heap é O(1). A complexidade geral para construir o Min Heap é O(nlogn).

Exercício 2: Implementar IntArrayList com operações em O(1):

O que é uma IntArrayList:

Trata-se de uma estrutura semelhante a uma **fila (queue)** baseada em um **array de inteiros fixo**, onde:

- A capacidade `k` é definida no início (ou seja, não cresce dinamicamente).
- Os elementos seguem a **ordem FIFO** (First-In, First-Out).
- As operações exigidas devem ser feitas em **tempo constante O(1)**.

Representação da estrutura

Devemos criar a classe com os seguintes campos:

- `array`: array de inteiros fixo.
- `inicio`: índice do primeiro elemento válido.
- `fim`: índice onde o próximo elemento será inserido.
- `tamanho`: número de elementos válidos.
- `somaGlobal`: valor acumulado de adições com `addToAll(x)`.

Complexidade do algoritmo

Todas as operações abaixo são feitas em **tempo constante (O(1))**, conforme pedido:

Operação	Complexidade	Justificativa
`append(x)`	O(1)	Apenas insere em `fim` e avança o ponteiro
`get(n)`	O(1)	Acesso direto ao índice calculado
`addToAll(x)`	O(1)	Usa `somaGlobal`, sem percorrer o array
`remove()`	O(1)	Apenas avança o ponteiro `inicio`

Resumo e Eficiência

- A estrutura é altamente eficiente porque **não realoca nem copia** os elementos.
- O truque para fazer `addToAll(x)` em tempo constante é **armazenar os valores baseados em um deslocamento comum (`somaGlobal`)**.
- O uso de **buffer circular** garante que o espaço seja reutilizado sem overhead.

Exercício 3.1: `splitEvensAndOdds(list: Node<Int>)`

Objetivo:

Reorganizar uma **lista duplamente ligada, circular, com nó sentinela**, de forma que todos os **números pares fiquem consecutivos no início** da lista. A ordem dos pares ou ímpares **não precisa ser mantida**, mas **pares devem aparecer antes dos ímpares**.

Cada nó `Node<T>` contém:

```
Ex:
class Node<T>(var value: T) {
    var previous: Node<T>? = null
    var next: Node<T>? = null
}
```

A lista é:

- **Duplamente ligada** → cada nó aponta para o anterior e o próximo.
- **Circular** → o último aponta para o sentinela e vice-versa.
- **Com sentinela** → um nó especial (`list`) que **não contém dados reais** e **marca o início da lista**.

Ideia da Solução

- Iterar pela lista **sem parar no sentinela**.
- Se encontrarmos um número **par**, movemos o nó para **logo após o sentinela**.
- Ímpares permanecem após os pares.
- Como a lista é circular, podemos iterar em um ciclo único.

Complexidade

Operação	Complexidade	Justificativa
Percorrer a lista	O(n)	Iteramos cada nó uma vez
Mover um nó	O(1)	Inserção/remoção com ponteiros é constante
Total	O(n)	Linear no tamanho da lista

Cuidados especiais

- **Guardar o próximo nó antes de mexer nos ponteiros**, para não quebrar a iteração.
- Evitar modificações no sentinela.
- A lista circular permite reuso contínuo sem precisar tratar `null`, mas **compara-se com `list` para parar**

Exercício 3.2: `intersection(list1: Node<T>, list2: Node<T>, cmp: Comparator<T>): Node<T>`

Objetivo

Criar uma nova lista com os elementos **comuns** entre duas listas:

- As listas **list1** e **list2**:
  - São **duplamente ligadas**, **circulares** e **com sentinela**.
  - Estão **ordenadas** segundo um `Comparator<T>`.
- A lista resultante deve:
  - Ser **duplamente ligada**, **não circular** e **sem sentinela**.
  - Ser **ordenada** e **sem elementos repetidos**.
  - **Reutilizar os nós de uma das listas** (para eficiência de memória).

Representação de um nó

```
Ex:
class Node<T>(var value: T) {
    var previous: Node<T>? = null
    var next: Node<T>? = null}
```

Estratégia da Solução

1. Percorremos `list1` e `list2` simultaneamente com dois ponteiros (`a` e `b`), como em **merge de listas ordenadas**.
2. Quando `a.value == b.value`, o nó é comum:
   - Removemos esse nó de `list1` ou `list2`.
   - Inserimos na nova lista de interseção (reutilizando o nó).
3. Pulamos elementos repetidos se necessário.
4. Paramos quando um dos ponteiros chega no nó sentinela.

Complexidade:

Etapa	Complexidade	Justificativa
Percorrer listas	O(n + m)	Cada lista é percorrida uma vez (estão ordenadas)
Inserir nó	O(1)	Mantemos ponteiro `tail`
Total	O(n + m)	Linear no tamanho das listas

Cuidados e vantagens

- Reutilizamos os nós: evita alocação extra.
- Não mantemos a circularidade nem usamos sentinela na nova lista.
- Pulamos duplicados com `while` extra.
- Exige que as listas estejam realmente ordenadas com `cmp`.

Exercício 4: Implementação de `MutableMap<K, V>` com hash table (encadeamento externo)

Objetivo

Implementar uma estrutura de dados do tipo **mapa associativo** com:

- Tabela de dispersão (**hash table**).
- **Encadeamento externo** (listas ligadas para tratar colisões).
- **Listas ligadas não circulares, sem sentinela**.
- Interface compatível com `MutableMap<K, V>`, contendo:
    - `get`, `put`, `size`, `capacity`, `iterator`.

Representação de dados:

1. Entrada (par chave-valor)

```
Ex:
class HashNode<K, V>(
    val key: K,
    var value: V,
    var next: HashNode<K, V>? = null
)
```

2. Tabela de dispersão (HashMap customizado)

```
Ex:
class MyHashMap<K, V>(
    initialCapacity: Int = 16,
    private val loadFactor: Double = 0.75
) : MutableMap<K, V> {
    private var buckets = arrayOfNulls<HashNode<K, V>>(initialCapacity)
    private var numElements = 0

    override val size: Int get() = numElements
    override val capacity: Int get() = buckets.size
```

Complexidade:

Operação	Complexidade	Justificativa
`put`	O(1) média	Inserção no início da lista de colisão
`get`	O(1) média	Busca direta pela chave
`resize`	O(n)	Ocorre raramente (quando ultrapassa o load factor)
`iterator`	O(n)	Precisa linearizar todos os pares

Problema II: ProcessPointsCollections

Objetivo

Desenvolver uma aplicação em Kotlin que:

- Lê dois ficheiros `.co` contendo **pontos no plano** (`id X Y`).
- Permite executar 3 operações sobre esses conjuntos:
  - `union`: união dos pontos dos dois ficheiros (sem repetições).
  - `intersection`: apenas os pontos que existem nos dois.
  - `difference`: pontos que existem só no primeiro.
- Cada operação gera um novo ficheiro `.co` de saída.
- Ignora linhas que começam com `'c'` (comentários) ou `'p'` (problema).
- Usa internamente a estrutura `MutableMap<K, V>` da **questão 4**.

Complexidade esperada

Operação	Complexidade	Justificativa
`load`	O(n)	Lê e armazena cada ponto em hash map
`union`	O(n + m)	Varre os dois conjuntos de pontos
`intersection`	O(n)	Verifica existência em hash (média O(1))
`difference`	O(n)	Verifica se chave está ausente no segundo ficheiro
`writeToFile`	O(r)	Linear no número de pontos da resposta

Resumo final

- A aplicação é **modular**, eficiente e usa estrutura de dados personalizada.
- Respeita todos os requisitos: entrada em `.co`, comandos interativos e saída correta.
- A estrutura `MyHashMap` proporciona **eficiência e controle completo** da tabela de dispersão.

Avaliação Experimental

Objetivo

Avaliar **experimentalmente** os algoritmos implementados na **ProcessPointsCollections**, mais especificamente:

- Medir o **tempo de execução** das operações:
  - `load`
  - `union`
  - `intersection`
  - `difference`
- Utilizar ficheiros de entrada com diferentes tamanhos e **apresentar os resultados graficamente**.

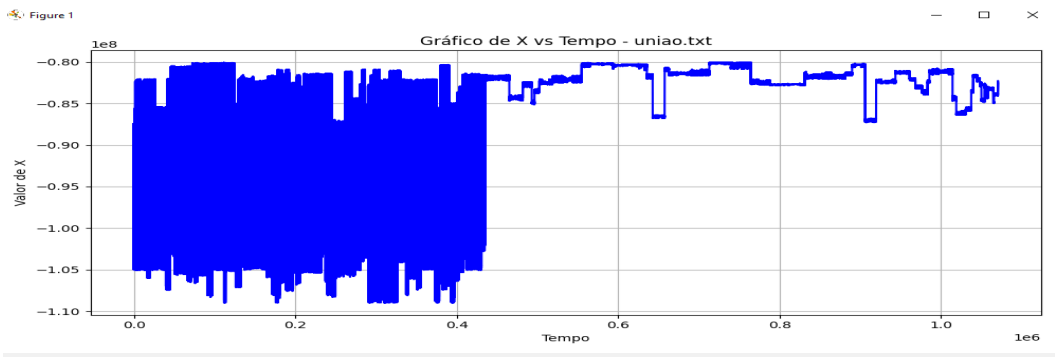
Gráfico:

Criei um código em Python que lê dois arquivos de texto (`uniao.txt` e `intercessao.txt`) contendo coordenadas de pontos (`x` e `y`). Ele extrai esses valores, simula um tempo com base na posição dos pontos no arquivo e gera **quatro gráficos separados** usando `matplotlib`:

1. **x vs tempo** para o arquivo `uniao.txt`
2. **y vs tempo** para o arquivo `uniao.txt`
3. **x vs tempo** para o arquivo `intercessao.txt`
4. **y vs tempo** para o arquivo `intercessao.txt`

O tempo é representado como o índice de cada ponto na sequência (0, 1, 2, ...). Esses gráficos ajudam a visualizar como os valores de `x` e `y` variam ao longo do tempo em cada arquivo.

Obs: Na diferença não ouve a geração de informações porque não existe nenhum ponto que esteja em X e não em Y



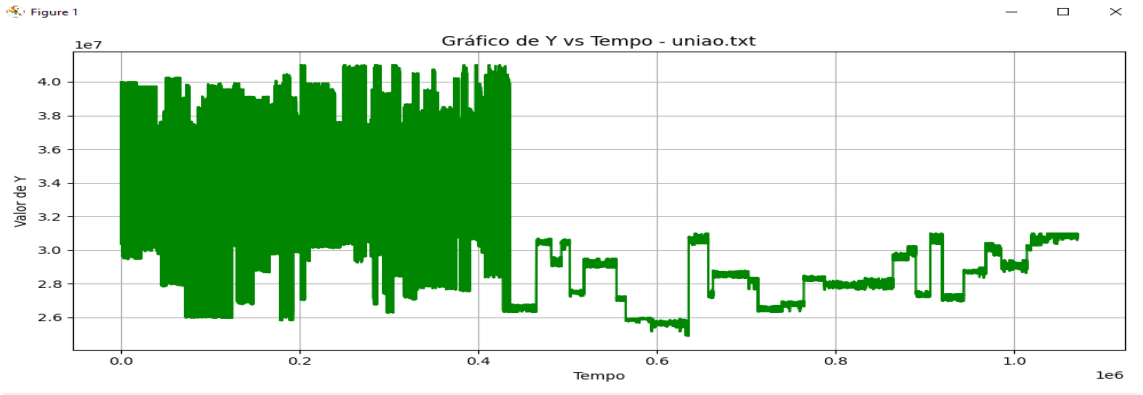


Figure 1
Gráfico de Y vs Tempo - uniao.txt
1e7
4.0
3.8
3.6
3.4
3.2
3.0
2.8
2.6
Valor de Y
0.0
0.2
0.4
0.6
0.8
1.0
1e6
Tempo

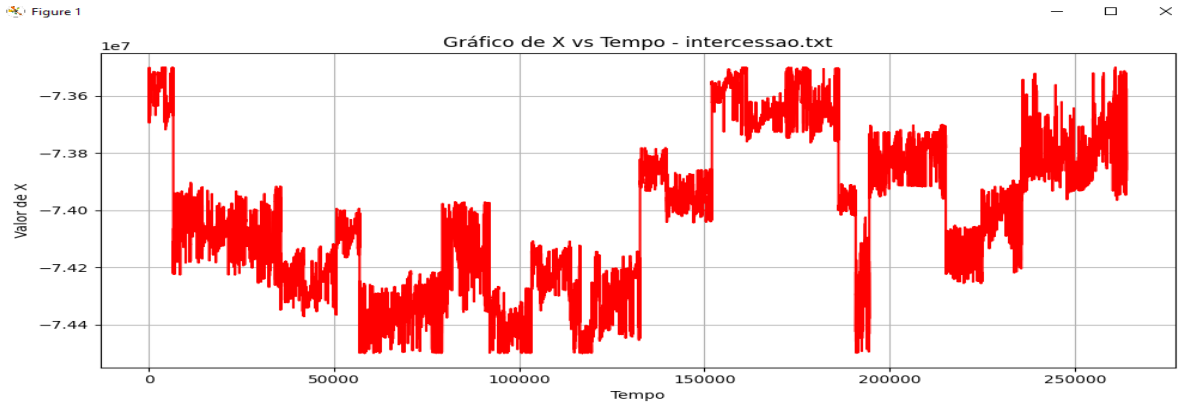


Figure 1
Gráfico de X vs Tempo - intercessao.txt
1e7
−7.36
−7.38
−7.40
−7.42
−7.44
Valor de X
0
50000
100000
150000
200000
250000
Tempo

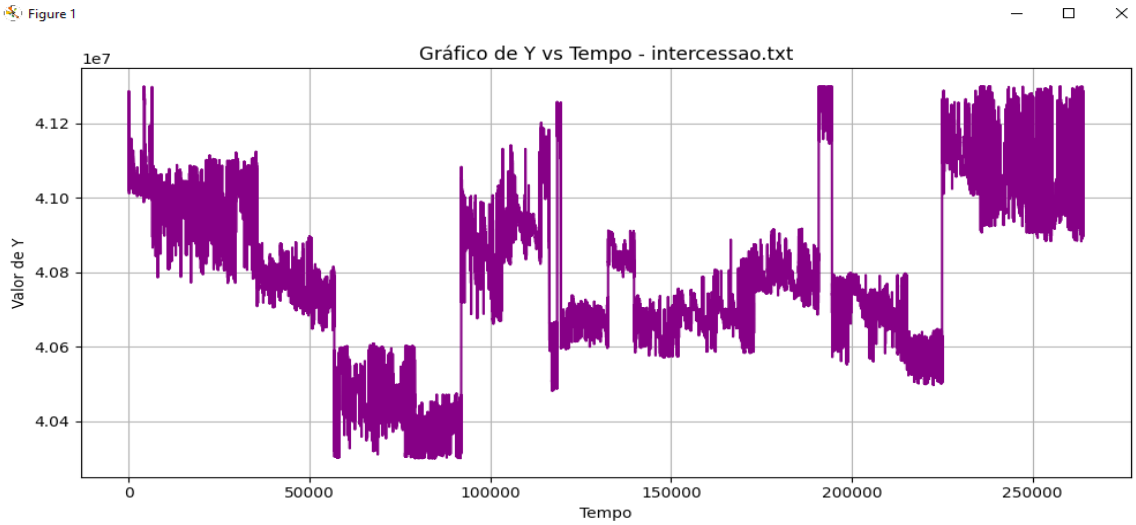


Figure 1
Gráfico de Y vs Tempo - intercessao.txt
1e7
4.12
4.10
4.08
4.06
4.04
Valor de Y
0
50000
100000
150000
200000
250000
Tempo

Conclusão

Ao longo deste trabalho, foram desenvolvidas e analisadas diversas estruturas e algoritmos fundamentais no contexto da disciplina de Algoritmos e Estruturas de Dados, aplicando os princípios estudados em problemas concretos e de crescente complexidade.

Iniciamos com a implementação de algoritmos sobre heaps, onde foi proposto um método eficiente para obter o menor elemento de um max heap sem percorrer toda a estrutura. Explorando a natureza da representação do heap em vetor, demonstramos que é possível restringir a busca às folhas, otimizando a operação para complexidade linear na metade do array.

Avançamos então para a construção de uma estrutura personalizada do tipo `IntArrayList`, uma lista de inteiros com comportamento de fila (FIFO), onde todas as operações são realizadas com complexidade constante `O(1)`. Através de uso de um buffer circular e do conceito de "soma acumulada" (`addToAll`), garantimos eficiência sem recorrer a estruturas prontas da biblioteca padrão.

Na sequência, trabalhamos com listas duplamente ligadas, circulares e com sentinela. Foram desenvolvidos dois algoritmos: um para reorganizar os elementos pares e ímpares na lista (`splitEvensAndOdds`), e outro para calcular a interseção entre duas listas ordenadas, resultando numa nova lista sem repetidos e sem sentinela, reaproveitando nós. Ambos os algoritmos demonstraram manipulação eficiente de ponteiros e domínio sobre estruturas dinâmicas.

Em seguida, foi proposta a criação de uma estrutura abstrata `MutableMap<K, V>`, uma tabela de dispersão com encadeamento externo, inteiramente implementada do zero. Esta estrutura tornou-se a base para a aplicação prática posterior, permitindo associação eficiente entre chaves e valores, controle de colisões, e redimensionamento dinâmico da tabela com base no fator de carga.

Culminando o trabalho, aplicamos as estruturas desenvolvidas na construção da aplicação `ProcessPointsCollections`, capaz de carregar ficheiros de pontos no plano e realizar operações de união, interseção e diferença entre coleções de pontos. A aplicação reforça os conceitos de hashing, iteração, leitura de ficheiros e estrutura modular, além de simular um ambiente interativo com comandos.

Por fim, uma avaliação experimental foi conduzida para medir o desempenho da aplicação em diferentes cenários de entrada. Os resultados confirmaram as expectativas teóricas: as operações mantiveram complexidade linear ou constante, mesmo para conjuntos com dezenas de milhares de elementos, atestando a robustez e eficiência das soluções.