Trabalho 1: Simulador de Livro de Ordens de Alta Performance

Luca Weidner Bohnenberger
Disciplina de Algoritmos e Estrutura de Dados 2 – PUCRS

18 de setembro de 2025

Resumo

Este artigo apresenta a solução desenvolvida para o Trabalho 1 da disciplina de Algoritmos e Estrutura de Dados 2 da PUCRS. O documento descreve a concepção e implementação de um simulador de livro de ordens de alta performance para a HAL Corporation, uma empresa do setor financeiro. O problema central consiste em processar um fluxo contínuo de ordens de compra e venda de ações, executando transações que maximizem o lucro da fintech. É apresentada uma solução baseada no uso de duas filas de prioridade (heaps) para gerenciar as ordens e garantir a identificação ótima de pares para negociação. A eficiência da abordagem é validada pelos resultados obtidos, que demonstram a capacidade do sistema de processar um grande volume de transações em tempo real.

Introdução

No dinâmico mercado financeiro, a velocidade e a eficiência no processamento de transações são cruciais. Dentro do escopo do Trabalho 1 de Algoritmos e Estrutura de Dados 2, foi proposto o desenvolvimento de um sistema capaz de processar um livro de ordens em tempo real para a HAL Corporation, executando imediatamente todas as transações possíveis com o objetivo principal de maximizar o lucro da empresa.

O sistema deve lidar com dois tipos de ordens:

1. **Ordem de Compra (`C <quant> <preço>`)**: Uma instrução para comprar uma quantidade `<quant>` de ações a um preço **máximo** de `<preço>` por unidade.
2. **Ordem de Venda (`V <quant> <preço>`)**: Uma instrução para vender uma quantidade `<quant>` de ações a um preço **mínimo** de `<preço>` por unidade.

O lucro da fintech é gerado pela diferença (*spread*) entre o preço de uma ordem de compra e o de uma ordem de venda compatível. Por exemplo, ao casar uma ordem de compra a R$120 com uma ordem de venda a R$115, a empresa lucra R$5 por ação negociada. O desafio central é, portanto, projetar um algoritmo que não apenas processe as ordens rapidamente, mas que o faça seguindo uma estratégia que maximize o lucro total acumulado.

Solução Proposta

Para resolver o problema proposto, analisamos a estratégia de negócio e as estruturas de dados mais adequadas para implementá-la com alta performance.

Estratégia de Maximização de Lucro

Para garantir que cada transação individual gere o maior lucro possível, a estratégia adotada é sempre casar a melhor oferta de compra com a melhor oferta de venda disponíveis no mercado. Isso se traduz na seguinte regra:

> **Casar a ordem de compra com o preço mais alto disponível com a ordem de venda com o preço mais baixo disponível.**

Uma transação somente é executada se o preço da ordem de compra for maior ou igual ao preço da ordem de venda (`preço_compra >= preço_venda`). Ao seguir esta regra de forma consistente para cada nova ordem que chega, garantimos uma maximização local do lucro em cada passo. Como as ordens são processadas sequencialmente, essa abordagem leva à maximização do lucro total ao final da simulação.

Algoritmos e Estruturas de Dados

A implementação eficiente da estratégia descrita depende do acesso rápido às ”melhores”ordens a qualquer momento. Para isso, a solução foi implementada utilizando duas **Filas de Prioridade**, mais conhecidas como *Heaps*.

- **Max-Heap para Ordens de Compra**: Uma fila de prioridade que mantém a ordem de compra com o **maior preço** sempre no topo. Esta estrutura permite acesso em tempo constante ($O(1)$) à melhor oferta de compra, o que é fundamental para avaliar possíveis transações com novas ordens de venda.
- **Min-Heap para Ordens de Venda**: De forma análoga, uma fila de prioridade que mantém a ordem de venda com o **menor preço** no topo. Isso garante acesso em tempo constante ($O(1)$) à oferta de venda mais competitiva.

As operações de inserção e remoção em ambas as estruturas possuem complexidade de tempo logarítmica ($O(\log n)$), onde n é o número de ordens pendentes. Essa característica garante que o sistema se mantenha rápido e escalável.

Implementação e Funcionamento

O projeto foi estruturado em três componentes principais: a classe `Ordem` para representar os dados, as classes `MaxHeap` e `MinHeap` para a estrutura de dados, e a classe `OWB` como o orquestrador central da lógica de negócios.[1]

A Classe `Ordem`

Para que os heaps pudessem comparar e organizar os objetos, foi criada a classe `Ordem`, que encapsula a quantidade e o valor de uma ordem. Os métodos de comparação (`__lt__`, `__gt__`, etc.) foram implementados para que a fila de prioridade opere com base no atributo `valor`.

```
# classe.py
class Ordem:
    def __init__(self, quantidade, valor):
        self.quantidade = quantidade
        self.valor = valor

```

[1] O código-fonte completo da implementação está disponível no repositório GitHub: https://github.com/LucaWBohnenberger/T1_Alest2.git

```
    # Exemplo de metodo de comparacao para o heap
    def __lt__(self, other):
        return self.valor < other.valor
```

Heaps Mínimo e Máximo

A estrutura de dados `MaxHeap` foi implementada para gerenciar as ordens de compra, garantindo que a de maior valor esteja sempre no topo. A classe `MinHeap`, que gerencia as vendas, herda de `MaxHeap` e simplesmente inverte a lógica dos métodos `swim` e `sink` para manter o menor valor no topo. A diferença crucial está na condição de comparação, como visto no método `swim`:

```
# heaps.py - MaxHeap
def swim(self, index):
    while index > 1 and self.list[index] > self.list[index // 2]:
        # ... logica para subir na arvore

# heaps.py - MinHeap (herda de MaxHeap)
def swim(self, index):
    while index > 1 and self.list[index] < self.list[index // 2]:
        # ... logica para subir na arvore
```

Lógica Central de Negociação

A classe `OWB` contém a lógica de processamento. Ao receber uma nova ordem de compra, por exemplo, o método `_processar_compra` consulta o topo do heap de vendas (`self.vendas.peek()`) e, enquanto houver vendas compatíveis (com preço menor ou igual), ele executa as transações.

```
# operadorWB.py
def _processar_compra(self, quantidade, valor):
    while (quantidade > 0 and
           not self.vendas.is_empty() and
           self.vendas.peek().valor <= valor):

        melhor_venda = self.vendas.remove()
        qtd_negociada = min(quantidade, melhor_venda.quantidade)

        self.lucro_total += qtd_negociada * (valor - melhor_venda.valor)

        # ... logica para atualizar quantidades ...
```

Uma lógica análoga é aplicada no método `_processar_venda`, que interage com o heap de compras.

Resultados

O algoritmo foi implementado e executado com um arquivo de teste fornecido pela HAL Corporation. Os resultados obtidos pela simulação foram validados e correspondem exatamente aos valores de referência esperados. O tempo de execução para o volume de dados testado foi praticamente instantâneo, confirmando a eficiência da abordagem.

Os resultados finais da simulação foram:

- **Lucro da Empresa**: $45538
- **Quantidade de Ações Negociadas**: 1228
- **Ordens de Compra Pendentes**: 3
- **Ordens de Venda Pendentes**: 7

Conclusões

A solução desenvolvida atende de forma robusta aos requisitos de maximização de lucro e alta performance. A escolha de filas de prioridade (Heaps) se mostrou a mais acertada para o problema, permitindo acesso imediato às ordens mais competitivas e mantendo a complexidade das operações em um nível muito baixo.

A complexidade de tempo para processar uma única nova ordem que resulta em k transações é de $O(k \cdot \log n)$, onde n é o número de ordens pendentes no livro. Como as operações fundamentais (`inserir`, `remover` e `consultar o topo`) são logarítmicas, o sistema escala de forma excelente. Mesmo com milhões de ordens pendentes, o tempo de processamento permanece extremamente baixo, garantindo a capacidade do sistema de lidar com as "milhares de transações por segundo"exigidas pela fintech. Acreditamos ter desenvolvido uma solução eficiente e elegante para o problema proposto na disciplina.