

# Fintech

Anderson R. P. Sprenger

Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul  
Av. Ipiranga, 6681 – 90.619-900 – Porto Alegre – RS – Brasil

anderson.sprenger@edu.pucrs.br

**Resumo:** Este artigo apresenta uma solução para auxiliar uma empresa fictícia do ramo financeiro encontrar a melhor combinação de ordens de compra e venda de ações de uma corporação, com a finalidade de ter o maior lucro possível intermediando tais transações. Ele expõe uma descrição da problemática do caso, sua análise, e uma solução proposta pelo autor deste artigo, junto com os pseudocódigos utilizados para a implementação de uma estrutura de dados *heap* para a resolução. Destarte, foram testados 8 casos fornecidos, e analisados os resultados obtidos. Este artigo é referente ao trabalho 1 da disciplina de Algoritmos e Estrutura de Dados 2 da Pontifícia Universidade Católica do Rio Grande do Sul.

## Problemática

De acordo com o enunciado[1], uma empresa fictícia do ramo financeiro, denominada por *fintech*, necessita de uma solução para encontrar a melhor combinação de ordens de compra e venda de modo a obter o maior lucro possível intermediando estas transações. Para cada ordem de compra, o cliente deseja comprar um número de ações por um valor menor ou igual ao informado, e para cada ordem de venda, o cliente deseja vender por um valor maior ou igual ao informado.

A *fintech* lucra através da diferença entre os valores de compra e venda de ações. Por conseguinte, a fim de obter a maior margem de lucro, é necessário que os valores entre operações de compra e venda tenham a maior diferença possível. Para a *fintech* não obter prejuízo, as ações não podem ser negociadas quando o preço de compra for menor que o de venda. Após efetuada uma transação de compra ou venda, no caso de a quantidade do pedido de compra ser diferente do pedido de venda, as ordens não realizadas continuam disponíveis a novas transações.

Por exemplo, a ordem de compra mais alta é de 15 ações com preço 12, e a ordem de venda mais baixa é de 10 ações com preço 6, decorre uma diferença de 5 ações restantes na ordem venda que não foram negociadas e estarão disponíveis para a próxima negociação. Neste exemplo, foram negociadas 10 ações, e a empresa lucrou 6 em cada ação negociada, totalizando um lucro de 60. Para tal, foram fornecidos 7 casos com tamanhos diferentes, 10, 100, 1.000, 10.000, 100.000, 1.000.000 e 10.000.000, adicionalmente foi disponibilizado 1 caso de teste no enunciado com 30 operações totalizando 8 casos.

Cada arquivo possui em sua primeira linha o número de ordens, nas linhas seguintes estão ordens de compra e de venda, com cada linha tendo uma ordem, seu tipo de operação, com a letra C para compra e V para venda, a quantidade de ações do pedido e o preço, e a última linha do arquivo possui a letra q. Os formatos das ordens de compra e venda, respectivamente, nas linhas dos arquivos de caso são:

C <quantidade> <preço>  
V <quantidade> <preço>

Por fim, as transações devem começar a serem efetuadas durante o tempo de leitura do arquivo e devem ser tão performáticas quanto possível.

### Análise do Problema

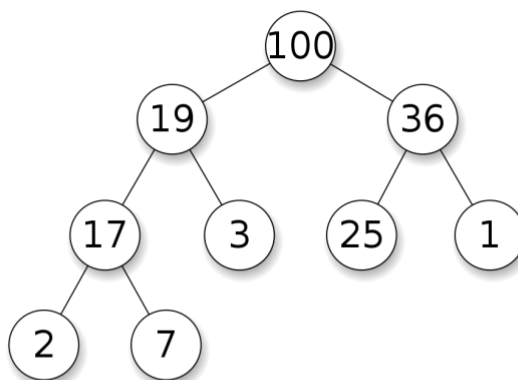
À medida que o arquivo é lido, nota-se que gradualmente acumulam-se ordens de compra e venda de ações. Tendo em vista que é necessário que haja a maior diferença possível entre os preços das ordens de compra e de venda, deverão sempre ser negociadas as ordens de compra com maior valor e as de venda com menor valor.

É denotado, portanto, a formação de duas filas de prioridade, uma de ordens de compra, onde as ordens com maior valor terão prioridade, e outra de venda, onde as ordens de menor valor serão priorizadas. Tendo em vista a grande quantidade de ordens a ser realizadas, a complexidade do método empregado para encontrar a ordem com prioridade é de grande impacto para o desempenho da aplicação. Soluções mais modestas como o emprego de lista para a fila de prioridade aumentarão consideravelmente o tempo.

Uma das implementações mais eficientes de filas de prioridade é o *heap* binário. Esta é uma estrutura de dados baseada em árvore onde cada nível é completado da esquerda para direita e precisa estar completo antes de começar a adicionar itens no próximo nível[2].

Além disso, o *heap* possui uma propriedade onde para cada nodo P do *heap*, se P for pai de um nodo filho F, então, no caso de um *max heap*, P deve ser maior que F, e no caso de um *min heap*, P deve ser menor que F. Esta propriedade garante que a raiz de um *heap* sempre possua o nodo com maior prioridade, ou seja, o maior nodo no caso do *max heap*, ou o menor no caso do *min heap*.

### Tree representation



### Array representation

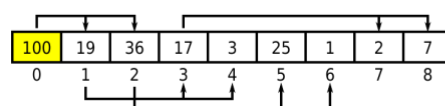


Figura 1 - Representações de um heap em árvore e em um vetor.

Ademais, os *heaps* binários são implementados com vetores. Para tal, assumindo que *posicaoPai* e *posicaoFilho* são, respectivamente, as posições dos nodos pai e filho no vetor:

- o nodo filho a esquerda está na posição do vetor:  $2 * posicaoPai + 1$
- o nodo filho a direita está na posição:  $2 * posicaoPai + 2$
- o nodo pai está na posição:  $(posFilho - 1) / 2$ , sendo desconsideradas as casas decimais.

Considerando tais regras, temos a raiz do *heap* na posição 0 do vetor, o que assegura a que a raiz possa ser consultada com complexidade  $\Theta(1)$ , pois não é necessária uma busca previa para encontrar o elemento na posição 0 de um vetor. Ademais, como uma fila de prioridades, a estrutura garante só possa ser removida do *heap* o nodo com a maior prioridade.

Contudo, inserir e remover elementos do *heap* exige que a estrutura de dados seja reordenada para continuar atendendo as suas propriedades. Para adicionar um elemento ao *heap*, basta colocá-lo na primeira posição do vetor que não possui um nodo do *heap* e executar a função *sift\_up* com esta posição como parâmetro. Nela é verificado se o nodo do pai é menor que o nodo do filho, caso isso ocorra, os nodos têm sua posição trocada pelo algoritmo a seguir, cujo é a implementação para o *max heap*. No caso do *min heap* é verificado se o nodo do pai é maior que o nodo do filho.

```
sift_up(int posicao) {
    int posicaoPai = (posicao - 1) / 2;
    if (vetor[posicaoPai] < vetor[posicao]) {
        swap(posicaoPai, posicao);
        sift_up(posicaoPai)
    }
}
```

Para remover um elemento ao *heap*, basta armazenar o elemento da raiz em uma variável a ser retornada, tirar o elemento da posição do vetor que possui um nodo do *heap*, colocá-lo na raiz, e executar a função *sift\_down* com a posição da raiz como parâmetro. Para o caso do *max heap*, é verificado qual dos filhos do nodo, caso exista, é o maior, e verificado se este nodo filho é maior que o cuja posição esta no parâmetro, caso positivo, ele tem sua posição trocada com o maior dos filhos. Para o *min heap*, é verificado se o menor dos filhos é menor que o pai, ocorrendo a troca neste caso. O algoritmo abaixo é a implementação do *sift\_down* do *max heap*, nele considere que *proximaPosUsadaVetor* é a primeira posição no vetor cuja não possui um nodo do *heap*.

```
sift_down(int posicao) {
    int posicaoFilhoEsquerda = 2 * posicao + 1;
    int posicaoFilhoDireita = 2 * posicao + 2;
    int posicaoMaior = posicao;

    if (
        posicaoFilhoEsquerda < proximaPosUsadaVetor &&
        v[posicaoFilhoEsquerda] > v[posicaoMaior])) {
        posicaoMaior = posicaoFilhoEsquerda;
    }
}
```

```

        if (
            posicaoFilhoDireita < proximaPosUsadaVetor &&
            v[posicaoFilhoDireita] > v[posicaoMaior]) {
                posicaoMaior = posicaoFilhoDireita;
            }

            if (posicao != posicaoMaior) {
                swap(posicao, posicaoMaior);

                sift_down(posicaoMaior);
            }
        }
    }
}

```

Segundo Dasgupta, ao analisar a complexidade das funções `sift_up`, e `sift_down`, temos que as inserções tem complexidade  $O(\log^2 n)$  e as remoções  $O(n)$ .

## Solução

A solução proposta para solucionar o problema da *fintech* envolve o uso de um *max heap*, para a inserção das ordens de compra, e de um *min heap*, para inserir as ordens de venda. Tendo isso em vista, utilizando a linguagem Java, são criadas duas classes, *Compra* e *Venda*. Elas implementam a interface *Operation* de modo a garantir que as classes sejam comparáveis entre si e que possuam os métodos *getters* para ler as quantidades das ordens e o preço. Logo, são customizadas as estruturas de dados *max heap* e *min heap* de forma a funcionarem com as classes *Compra* e *Venda*.

Durante a leitura do arquivo, se a linha começar com C ou V, é criado, respectivamente, uma ordem de compra ou venda, com os valores de quantidade e preço correspondente aos dois números esperados após a letra indicadora do tipo de operação. Depois, a ordem é adicionada ao seu respectivo *heap*, para então ser feita uma tentativa de negociar as ações pelo uso do algoritmo a seguir. Considere a variável *compras* como o *max heap* de compras, e *vendas* como o *min heap* de vendas.

```

trade() {
    if (compras.size() == 0 || vendas.size() == 0 ) {
        return;
    }

    if (compras.peek().getPreco() >= vendas.peek().getPreco()){

        Compra compra = (Compra) compras.get();
        Venda venda = (Venda) vendas.get();

        if (compra.getQuantidade() > venda.getQuantidade()) {
            compras.put(new Compra(compra.getQuantidade() -
                venda.getQuantidade(), compra.getPreco()));
        }
    }
}

```

```

        lucro += (compra.getPreco() - venda.getPreco())
        * venda.getQuantidade();

        acoesNegociadas += venda.getQuantidade();
    }

    else if (
        venda.getQuantidade() > compra.getQuantidade()
    ) {
        vendas.put(new Venda(venda.getQuantidade() -
            compra.getQuantidade(), venda.getPreco()));

        lucro += (compra.getPreco() - venda.getPreco())
        * compra.getQuantidade();
        acoesNegociadas += compra.getQuantidade();
    }

    else {
        lucro += (compra.getPreco() - venda.getPreco())
        * compra.getQuantidade();
        acoesNegociadas += compra.getQuantidade();
    }

    if (compras.peek().getPreco() >=
        vendas.peek().getPreco()){
        trade();
    }
}
}

```

Neste algoritmo, primeiro é verificado se os *heaps* tem uma raiz, e depois se o preço da compra na raiz de *compras* é maior que o preço da venda na raiz de *vendas*. Caso isto ocorra, as raízes dos *heaps* de *compras* e de *vendas* são retiradas e salvas, respectivamente nas variáveis *compra* e *venda*, em sequência, é verificado se há mais ordens de compra ou de venda.

Se houver mais ordens de compra, é criada uma ordem de compra com a diferença entre o número de ordens de compra e vendas como quantidade, o preço da variável *compra* como preço, e depois a ordem é colocada no heap de *compras*. Então é calculado o lucro da transação, com o produto do número de vendas pela diferença entre o preço de compra e o de venda. Consequentemente o lucro da transação é adicionado a uma variável de classe chamada *lucro*, e a quantidade de vendas é adicionada a uma variável de classe com nome *acoesNegociadas*.

Se houver mais ordens de venda, é criada uma ordem de venda com a diferença entre o número de ordens de venda e de compras como quantidade, o preço da variável *venda* como se preço, e depois a ordem é colocada no heap de *vendas*. Então é calculado o lucro da transação, com o produto do número de *compras* pela diferença entre o preço de compra e o de venda. Consequentemente o lucro da transação é adicionado a uma variável de classe

chamada *lucro*, e a quantidade de compras é adicionada a uma variável de classe com nome *acoesNegociadas*.

No caso de haver a mesma quantidade de compras e vendas, é calculado o lucro da transação, com o produto do número de *compras* pela diferença entre o preço de compra e o de venda. Consequentemente o lucro da transação é adicionado a uma variável de classe chamada *lucro*, e a quantidade de compras é adicionada a uma variável de classe com nome *acoesNegociadas*.

Por fim, é verificado novamente se preço da compra na raiz de *compras* é maior que o preço da venda na raiz de *vendas*, neste caso o método *trade* é chamado recursivamente até realizar todos os negócios possíveis.

## Resultados

A aplicação da solução foi executada com cada caso de teste disponibilizado, onde foram obtidos os valores de lucro, a quantidade de ações negociadas, de compras pendentes, de vendas pendentes, e do tempo de execução da solução. O método para obtenção do tempo de execução foi a diferença entre o retorno do método `System.currentTimeMillis()` executado no fim da execução (variável *t1*) pelo valor do retorno deste método no início (variável *t0*) da execução.

```
t0 = System.currentTimeMillis();
fintech.run(nome_do_arquivo_de_teste);
t1 = System.currentTimeMillis();
System.out.println("Tempo de execução: " + (t1 - t0) + "ms");
```

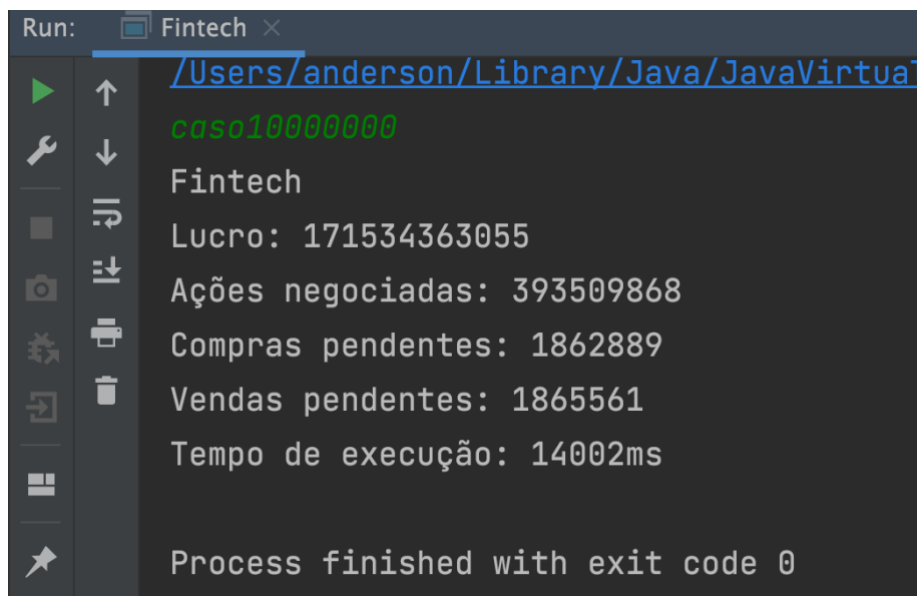
The image shows a screenshot of a Java IDE's Run console. The console window has a title bar that says 'Run: Fintech x'. The output text is as follows:  
/Users/anderson/Library/Java/JavaVirtual  
caso10000000  
Fintech  
Lucro: 171534363055  
Ações negociadas: 393509868  
Compras pendentes: 1862889  
Vendas pendentes: 1865561  
Tempo de execução: 14002ms  
Process finished with exit code 0  
The console also features a vertical toolbar on the left with various icons for running, debugging, and other IDE functions.

Figura 2 - Captura de tela da exibição dos resultados para o arquivo *caso10000000.txt*.

Casos	Lucro	Ações Negociadas	Compras Pendentes	Vendas Pendentes	Tempo de Execução
10	166932	364	1	3	10ms
30	45538	1228	3	7	10ms
100	1801040	3875	16	26	16ms
1000	18010606	39505	185	186	47ms
10000	171324569	398295	1779	1857	152ms
100000	1709357694	3922710	18504	18830	315ms
1000000	17192484984	39375260	186654	186168	1693ms
10000000	171534363055	393509868	1862889	1865561	14002ms

*Tabela 1 - Resultados obtidos na execução dos casos com a solução proposta.*

O caso mais complexo, com 10 milhões de operações, levou 14 segundos pra completar a execução. O caso com 30 operações apresentou resultados semelhantes aos descritos no enunciado do trabalho[1].<sup>i</sup>

## Conclusão

Ao analisar os resultados obtidos e expostos na seção acima, a implementação das filas de prioridade com *heaps* binários foi uma solução excelente para este caso. O algoritmo apresentou robustez suficiente para computar 1 milhão de ordens em 1,69 segundo e 10 milhões em 14 segundos. Isto responde a dúvida levantada no enunciado do trabalho, demonstrando que a solução não “engasgou” com casos maiores.

## Referencias

1. OLIVEIRA, João Batista Souza de. A Fintech. 02 de agosto de 2021.
2. Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Página 120. (2006). Algorithms. McGraw-Hill Professional. P.

---

<sup>i</sup> De modo a facilitar a reprodução do artigo, o código fonte do trabalho encontra-se disponível publicamente em: <https://github.com/andersprenger/fintech>