Plano Estratégico

Ordenação de pedidos considerando múltiplos critérios

Problema

A VelozMart busca ordenar pedidos em tempo real para minimizar atrasos e custos logísticos.
O sistema deve considerar múltiplos critérios e se adaptar à chegada constante de novos pedidos.

Algoritmo/Estrutura

Priority Queue

Para a resolução do problema apresentado, o algoritmo definido foi o priority queue. Para o cenário de alta frequência de inserções e extrações. A estrutura mantém sempre o pedido mais prioritário no topo (A[1]) e permite operações logarítmicas. Diante do exposto ao trabalhar com múltiplos fatores de prioridade, a estrutura que traz uma melhor resolução seria esta, definindo uma função de comparação que combine os três critérios relevantes, sendo eles priorityScore (0-100), dispatchWindow (minutos restantes) e sizeCategory (P, M, G).

A proposta será a de ter um reprocessamento a cada 02 minutos (120s) para ter tempo hábil de entrada dos novos pedidos. será

considerado no priority queue na seguinte ordem: criticidade do pedido, sua importância para o negócio e a otimização do espaço, levando em consideração o tamanho do pacote. Utilizando a estratégia de peso composto, traz-se um entendimento facilitado do processo, tornando-o mais acessível a quem não tiver um conhecimento aprofundado na área.

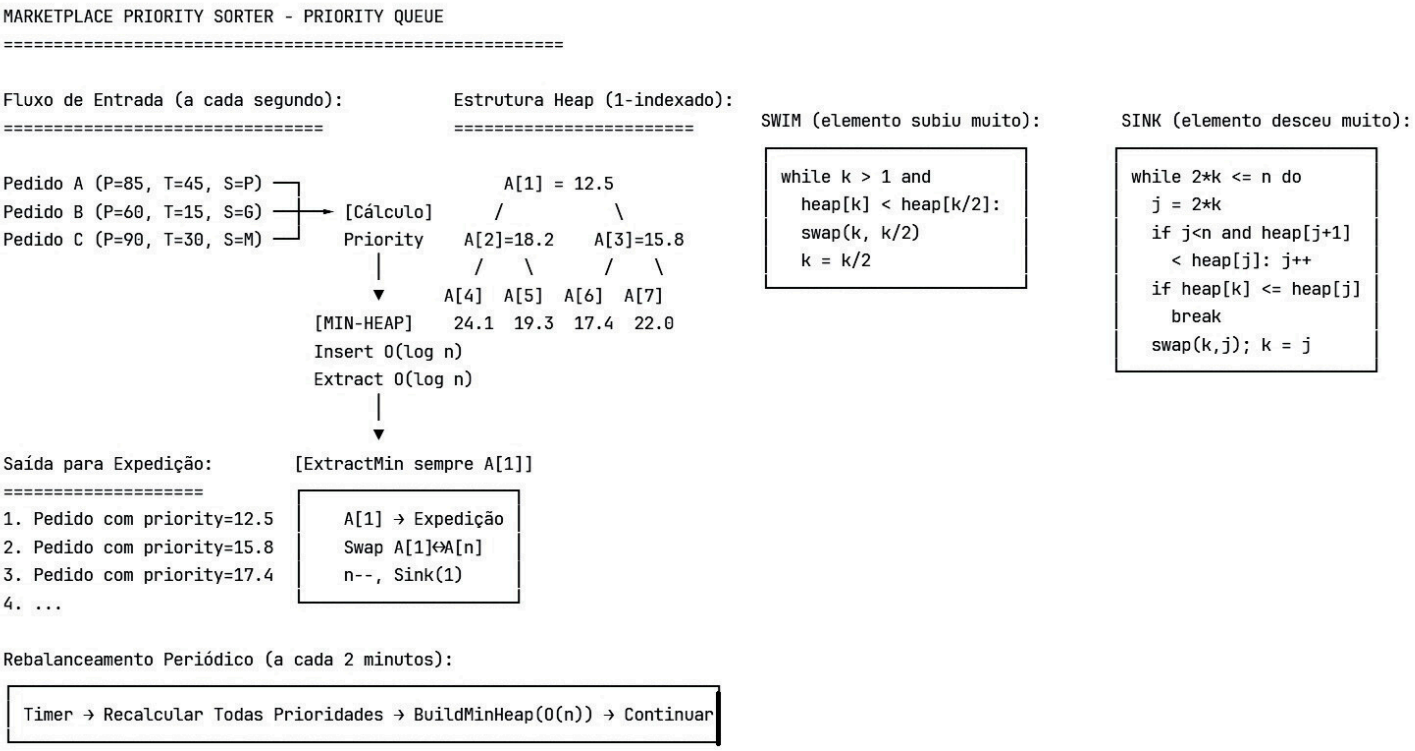
prioridade = (dispatchWindow * 2.0) - (priorityScore * 1.5) + pesoTamanho

Você pode controlar o peso de cada fator:
Mais urgente? Aumenta peso do dispatchWindow.
Mais valioso? Aumenta peso do priorityScore.
Mais difícil de manusear? Penaliza sizeCategory.

```
Função de Prioridade Composta:
priority = urgencyWeight × (maxWindow - dispatchWindow) +
           scoreWeight × (100 - priorityScore) +
           sizeWeight × sizeMultiplier

onde:
- urgencyWeight = 0.6 (60% - fator mais crítico)
- scoreWeight = 0.3 (30% - importância do negócio)
- sizeWeight = 0.1 (10% - otimização de espaço)
- sizeMultiplier = {P: 1, M: 2, G: 3}
Algoritmo Principal:
CLASSE OrderQueue:
  heap[1..capacity] // 1-indexado conforme material
  n = 0 // tamanho atual

  FUNÇÃO insert(pedido):
    SE n == capacity ENTÃO erro "fila cheia"
    priority = calcularPrioridade(pedido)
    heap[++n] = {pedido, priority}
    swim(n) // restaura propriedade heap

  FUNÇÃO extractNext():
    SE n == 0 ENTÃO erro "fila vazia"
    next = heap[1]
    swap heap[1] ↔ heap[n]
    n--
    SE n >= 1 ENTÃO sink(1)
    RETORNA next.pedido

  FUNÇÃO rebalance():
    PARA i = 1 ATÉ n:
      newPriority = calcularPrioridade(heap[i].pedido)
      heap[i].priority = newPriority
    buildMinHeap(heap, n) // O(n) reconstrução
```

Para gerar o pseudocódigo foi usado o Copilot, tornando o processo de apresentação de solução mais completo.

Diagrama



Para gerar o diagrama, foi usado o Gemini, com ele, organizamos de uma forma mais visual todos os processos, tornando a imagem auto explicativa.

Trade-offs

Prós:

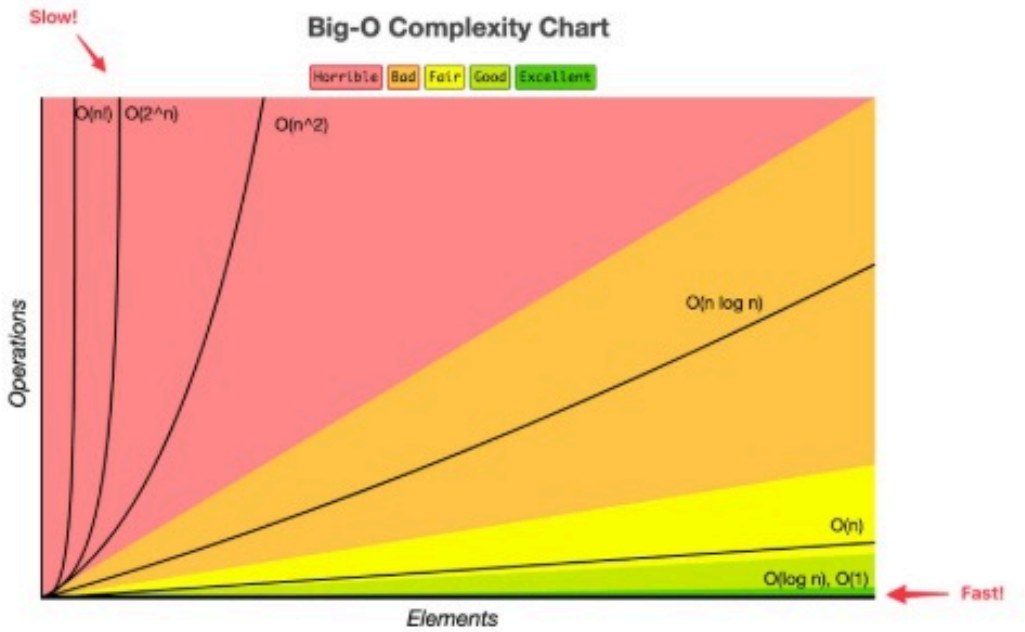
1. Eficiência Temporal Garantida: Inserção e extração em O(log n) conforme análise do material, mantendo performance mesmo com 10.000+ pedidos ativos.
2. Prioridade Sempre Correta: Garante que A[1] contém sempre o pedido mais prioritário, ideal para "melhor-pedido-primeiro" conforme requisito.
3. Estrutura Consolidada: Priority Queue é amplamente testada em schedulers de OS e feeds classificados, casos similares mencionados no material.

Contras:

1. Rebalanceamento Custoso: Reconstrução periódica do heap consome O(n) tempo, podendo impactar performance durante picos de demanda.
2. Busca Arbitrária Limitada: Conforme destacado no material, heap não oferece busca eficiente por pedidos específicos (O(n) para localizar pedido por ID).
3. Memória Adicional: Armazenamento de prioridades calculadas aumenta footprint comparado a ordenação simples por timestamp.

Análise Big O

A notação Big O geral para o algoritmo, considerando suas operações principais, pode ser considerada como O(log n) para adição e remoção de pedidos, e O(n) para exibição, com a possibilidade de O(n log n) para reavaliação, dependendo da abordagem que você escolher para essa parte do algoritmo.



Próximos passos

Aqui estão algumas sugestões de próximos passos e ideias futuras para aprimorar o sistema de otimização da expedição de pedidos no VelozMart, incluindo a incorporação de tecnologias avançadas como aprendizado de máquina, para isso, foi usado o ChatGPT 4.0 mini, de forma a organizar as ideias de maneira mais completas:

- **Avaliação do Sistema atual:**

Realizar testes abrangentes para avaliar o desempenho do sistema atual em cenários do mundo real. Coletar dados sobre tempos de espera, taxa de satisfação do cliente e eficiência do processamento de pedidos.

- **Desenho do Teste A/B:**

Teste A/B: Comparar o desempenho do priority queue com uma abordagem de fila simples baseada apenas em dispatchWindow. As métricas a serem observadas incluem atraso médio e throughput.

Grupo A (Controle): Implementação atual do sistema de priorização de pedidos. Grupo B (Variante): Uma nova abordagem, por exemplo: Teste A: Implementação de um algoritmo que garante que pedidos de baixa prioridade sejam atendidos em um tempo máximo. Teste B: Alteração na lógica de reavaliação de prioridades para incluir fatores como feedback em tempo real dos clientes.

Componentes	Bruno de Toledo Queiroz Elaine Fabíola Soares Thais Amorim de Oliveira Vagner Serafim da Silva
Data	10 de Julho de 2025

Todos os membros contribuíram com suas habilidades de comunicação para garantir o alinhamento nas metas. O projeto resultou em um documento norteador, onde foram expostas as razões as quais nos levaram a escolha do algoritmo e trouxe a importância da colaboração e do respeito às diferentes opiniões, o que certamente será aplicado em futuras experiências.