

Problema

O VelozMart lançou uma funcionalidade para exibir em tempo real os pedidos aguardando expedição. Cada pedido tem um PriorityScore (0–100), tempo restante até expiração do prazo (dispatchWindow) e tamanho (P, M, G). O desafio é ordenar dinamicamente os pedidos para reduzir atrasos e otimizar a operação logística.

Algoritmo / Estrutura

Em um ambiente logístico dinâmico como o do VelozMart, onde pedidos chegam e mudam a todo instante, é fundamental ter uma solução que mantenha a fila de expedição sempre priorizada de forma eficiente, adaptável e sustentável.

O modelo baseado em Min-Heap com chave composta foi escolhido por oferecer o equilíbrio ideal entre desempenho operacional, flexibilidade para ponderar múltiplos critérios estratégicos e simplicidade para evoluir e ser mantido pela equipe técnica no longo prazo.

pseudo

Copy Edit

```
// Novo pedido chega
função adicionarNovoPedido(novoPedido):
    novoPedido.chave = calcularPrioridade(novoPedido)
    heapPedidos.inserir(novoPedido, novoPedido.chave)

// Pedido existente é atualizado
função atualizarPedido(pedidoAtualizado):
    // Remove o pedido do heap
    heapPedidos.remove(pedidoAtualizado.id)

    // Recalcula a chave com os novos atributos
    pedidoAtualizado.chave = calcularPrioridade(pedidoAtualizado)

    // Reinsere no heap com a nova chave
    heapPedidos.inserir(pedidoAtualizado, pedidoAtualizado.chave)

// Quando for necessário despachar
função despacharProximoPedido():
    pedidoDespachado = heapPedidos.removeTopo()
    retornar pedidoDespachado
```

pseudo

Copy Edit

```
// Inicializa uma fila de prioridade (min-heap) vazia
heapPedidos = MinHeap()

// Função para calcular a prioridade composta
função calcularPrioridade(pedido):
    // Exemplo: ponderação simples
    pesoPriorityScore = 0.6
    pesoDispatchWindow = 0.4
    penalidadeTamanho = 0
    se pedido.sizeCategory == 'G':
        penalidadeTamanho = 10
    senão se pedido.sizeCategory == 'M':
        penalidadeTamanho = 5

    prioridade = (pedido.dispatchWindow * pesoDispatchWindow) /
                (pedido.priorityScore * pesoPriorityScore) +
                penalidadeTamanho
    retornar prioridade

// Para cada pedido recebido inicialmente
para cada pedido em listaPedidosIniciais:
    pedido.chave = calcularPrioridade(pedido)
    heapPedidos.inserir(pedido, pedido.chave)
```

Diagrama — Como ordenar pedidos considerando múltiplos critérios

Este diagrama mostra o fluxo básico de um pedido ao entrar no sistema:

Quando o pedido é recebido, uma chave composta é calculada considerando dispatchWindow, priorityScore e sizeCategory.

Essa chave representa a prioridade do pedido de forma quantitativa.

O pedido é então inserido no Min-Heap, que garante que a fila esteja sempre ordenada de forma eficiente, com o pedido mais prioritário no topo para despacho imediato.

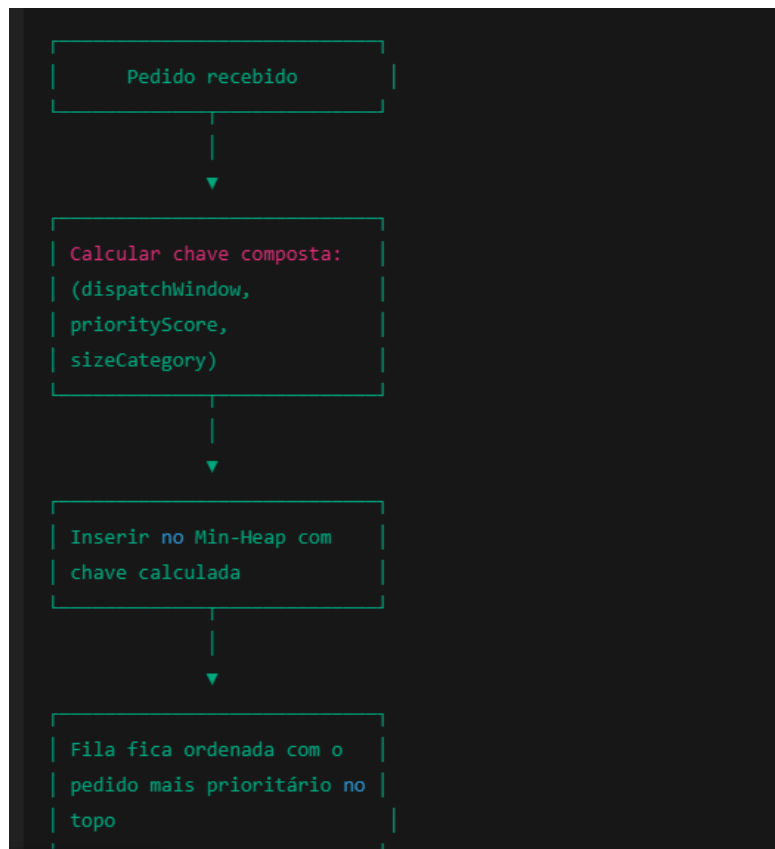


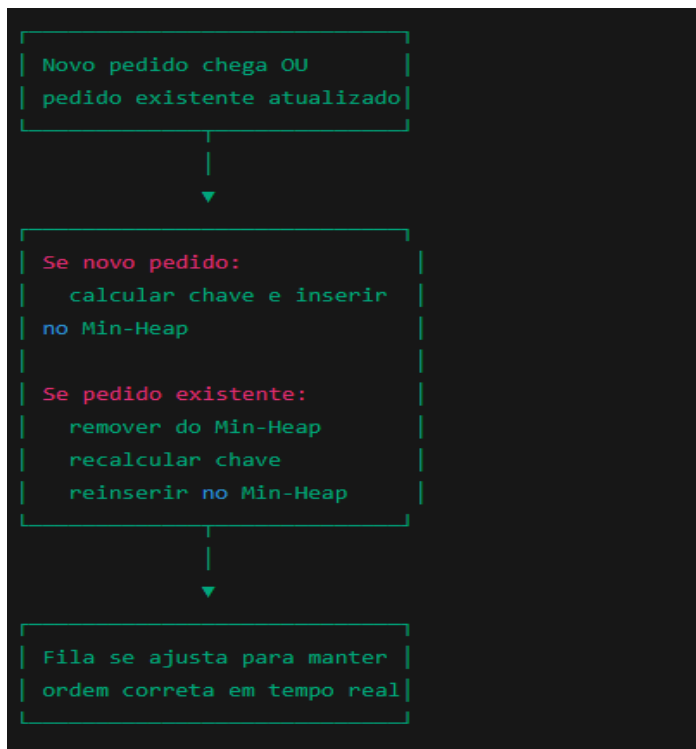
Diagrama — Como re-enfileirar pedidos à medida que novos chegam ou prazos mudam

Este diagrama complementa o anterior mostrando como o sistema lida com alterações dinâmicas:

Quando um novo pedido chega, o fluxo é idêntico ao anterior: calcular chave e inserir.

Quando um pedido existente tem atributos alterados (por exemplo, diminuiu o tempo restante), ele é removido do heap, sua chave é recalculada, e ele é reinserido.

Isso garante que a fila reflita a prioridade correta em tempo real, sem necessidade de reordenar a lista inteira.



Trade-offs




Prós:

- Suporta múltiplos critérios de forma flexível (basta ajustar os pesos).
- Permite inserção e remoção rápidas de pedidos em tempo real ($O(\log n)$).
- Escala bem para grandes volumes de pedidos sem perder desempenho.

Contras:

- Não é in-place: utiliza memória adicional para manter a estrutura do heap.
- Não é estável por padrão (empates podem alterar a ordem original).
- Requer calibrar bem os pesos para refletir corretamente as prioridades logísticas.

Mapa de Trade-off

Critério	Prós 	Contras 	
Simplicidade	<ul style="list-style-type: none"> - Estrutura bem conhecida (heap). - Algoritmo padrão com bibliotecas prontas. - Lógica clara de prioridade composta. 	<ul style="list-style-type: none"> - Exige definição inicial dos pesos para a chave composta. - Implementação mais complexa que um simples sort. 	
Desempenho	<ul style="list-style-type: none"> - Inserção e remoção rápidas: $O(\log n)$. - Sempre mantém fila ordenada sem necessidade de resorting completo. - Boa escalabilidade para centenas/milhares de pedidos. 	<ul style="list-style-type: none"> - Para filas gigantes (>1M), pode precisar de paralelismo. - Repriorização exige remover/reinserir. 	
Manutenibilidade	<ul style="list-style-type: none"> - Modular: fácil ajustar pesos na chave. - Estrutura separada da UI e regras de negócio. - Padrão comum para times técnicos manterem. 	<ul style="list-style-type: none"> - Mudanças nas regras de prioridade exigem alterar a função de cálculo. - Devs precisam conhecer bem a API do heap. 	

Justificativa: Min-Heap com chave composta

Eficiência operacional contínua:

Mantém a fila sempre priorizada em tempo real, com baixo custo computacional para inserções, remoções e reclassificações dinâmicas.

Modelagem flexível de prioridades:

Suporta múltiplos critérios ponderados (SLA, score, tamanho), permitindo calibrar a estratégia sem mudanças estruturais.

Escalabilidade com simplicidade:

Atende grandes volumes com desempenho previsível, utilizando uma estrutura padrão de fácil manutenção.

Como ordenar pedidos considerando múltiplos critérios?

Para ordenar pedidos levando em conta múltiplos critérios como **priorityScore**, **dispatchWindow** e **sizeCategory**, utilizamos um conceito chamado **Priority Key**, que transforma esses fatores em um único número de prioridade ponderada. Esse número é calculado atribuindo pesos aos critérios, por exemplo:

$$\text{priorityKey} = \alpha \times (100 - \text{priorityScore}) + \beta \times \text{dispatchWindow} + \gamma \times \text{sizeWeight}$$

Depois, os pedidos são gerenciados dinamicamente em uma **fila de prioridade implementada com Min-Heap**, garantindo que o pedido mais urgente esteja sempre no topo da fila para expedição.

Complexidade Assintótica (Big O)

- Inserção de pedido: $O(\log n)$
- Remoção do pedido mais prioritário: $O(\log n)$
- Atualização de prioridade: $O(\log n)$

In-Place?

- **Não é in-place**, pois usa uma estrutura de dados adicional (o heap).

Como re-enfileirar pedidos à medida que novos chegam ou prazos mudam?

A estrutura de dados **Min-Heap com Priority Key** facilita a re-enfileiração dinâmica de pedidos.

Sempre que um novo pedido chega ou um pedido existente tem algum de seus critérios atualizados (como `dispatchWindow` ou `priorityScore`), recalculamos sua **priorityKey** e o re-inserimos na fila com a nova prioridade.

Como funciona:

- Para **novo pedido**:
 - Calcula sua `priorityKey` com base nos critérios atuais.
 - Insere no Min-Heap em $O(\log n)$.
- Para **pedido com prioridade alterada**:
 - Remove o pedido do Min-Heap em $O(\log n)$.
 - Recalcula sua `priorityKey`.
 - Insere novamente no Min-Heap em $O(\log n)$.

Dessa forma, a fila é sempre mantida ordenada pela prioridade composta, refletindo o estado mais recente de cada pedido.

Pontos Positivos

Rápido e eficiente para atualizações frequentes.

Não é necessário reordenar a lista inteira.

Suporta alta taxa de chegadas e mudanças em tempo real.

Pontos Negativos

Requer atenção para evitar inconsistências (por exemplo, esquecer de remover antes de inserir novamente). Sobrecarga ligeiramente maior para sistemas muito pequenos. Complexidade de implementação maior que algoritmos simples.

Complexidade Assintótica (Big O)

- Atualização ou inserção de pedido: $O(\log n)$
- Remoção do mais prioritário: $O(\log n)$

Quais métricas comprovam que a estratégia é melhor que uma ordenação simples por tempo?

A ordenação por múltiplos critérios usando **Min-Heap com Priority Key** é superior à ordenação simples por **dispatchWindow** (tempo restante) porque considera **urgência real + impacto logístico + valor estratégico**.

Para comprovar essa superioridade, podemos analisar as seguintes métricas:

1. Atraso médio (Average Delay)

- Mede o tempo médio de atraso dos pedidos em relação ao SLA.
- Estratégia simples (ordenar só por tempo) pode priorizar um pedido barato e leve antes de outro mais importante.
- Com Priority Key, pedidos de maior valor logístico são despachados antes, **reduzindo o atraso médio dos mais relevantes**.

2. Throughput por hora (Pedidos/hora expedidos)

- Avalia quantos pedidos são processados por hora.
- Ao evitar o acúmulo de pedidos grandes e difíceis no final da fila, a ordenação inteligente distribui melhor os volumes, **aumentando o ritmo geral de expedição**.

3. Valor médio expedido por hora

- Soma do valor total dos pedidos expedidos dividido pelo tempo.
- Ao considerar o **priorityScore** (que inclui valor e reputação), a estratégia garante que **pedidos mais valiosos sejam priorizados**, elevando o retorno logístico.

4. Uso eficiente da doca (Balanceamento de tamanhos)

- Ordenar apenas por tempo pode concentrar muitos pedidos grandes em sequência.

- Com ponderação por **sizeCategory**, a estratégia prioriza de forma balanceada, **evitando gargalos físicos na doca de embalagem.**

Próximos passos

- **Calibrar pesos da chave composta:**
Realizar testes A/B para ajustar os pesos atribuídos a *priorityScore*, *dispatchWindow* e *sizeCategory*, maximizando a eficiência logística.
- **Monitorar métricas operacionais:**
Implementar dashboards para acompanhar indicadores como atraso médio, throughput/hora e % de pedidos dentro do SLA, validando o impacto do modelo.
- **Evoluir para aprendizado online:**
Explorar modelos de machine learning online para que os pesos da prioridade sejam ajustados automaticamente com base em dados históricos e mudanças no comportamento da demanda.
- **Integração com a UI e acessibilidade:**
Incorporar a lógica de priorização na interface de operação, com destaque visual para pedidos mais críticos e aderência a boas práticas de acessibilidade.