UNIVERSIDADE FEDERAL DE MINAS GERAIS
Faculdade de Engenharia de Produção

Trabalho Prático 2 - Sistema de Escalonamento Hospitalar

Tasso A. T. Pimmenta
2021072198

20 de janeiro de 2025

Sumário

Resumo

Esse trabalho trata de um software para simulação de eventos discretos de um sistema hospitalar com filas e niveis de urgencia, por meio da obtenção de eventos a acorrer a partir da menor data. A estrutura de dados implementada é a MinHeap, feita de forma generica para suportar o escalonador, as filas, a ordenação de servidores e a geração de estatisticas, pela sua caracteristica de sempre manter o menor 'sortparam" no topo, podendo ser a data ou id nos casos. Também implementado uma classe para armazenar as caracteristica dos pacientes e do funcionamento dos procedimentos.

1 Introdução

O sistema de atendimento de um Hospital começa pela adimção do paciente ao hospital, seguido pela triagem onde é determinado o nivel de urgencia do paciente, então ele é atendido. No atendimento é definido se ele já pode ter alta ou se deve seguir pelos outros quatro procedimentos, em internação antes da alta.

Com esse problema o objetivo do trabalho é fazer uma simulação do tempo em que cada paciente ficou no hospital, levando em conta que existe um certo numero de servidores disponiveis para aatendimento, e que caso esteja ocupado se deve esperar em fila. porém dado o nivel de urgencia definido pela triagem, existe uma prioridade no atendimento dos casos mais graves em detrimento dos mais leves.

O Hospital oferece quatro procedimentos:

1. Medidas hospitalares (p.ex., medir pressão e temperatura corpórea).
2. Testes de laboratório
3. Exames de imagem
4. Uso de instrumentos e aplicação de medicamentos

que de inicio é tradado como sequencia, ou seja um paciente só pode ir para o seguinte caso não precise ou já tenha passado no procedimento anterior. Porém é possivel otimizar esse sistema de forma que o paciente sempre entre no lugar que vai ter atendimento disponivel mais proximo, potencialmente diminuindo o tempo em espera.

A implementação desse trabalho usa da simulação de eventos discretos que salta de tempos em tempos para o proximo evento a ocrrer, de modo em que os tempos sempre sejam o horario em que o paciente está disponivel, seja a hora que chegou ou a hora que saiu, dessa forma garantindo que sempre será checado o horario de disponiblidade de um servidor. Os estados são definidos de forma linear, de modo em que entra na fila, depoois vai para o serviço correspondente, depois entra em outra fila, até que não tenha mais procedimentos ou tenha recebido alta.

2 Método

A função **main** é responsável por inicializar a simulação do sistema hospitalar. Ela começa lendo os parâmetros de entrada do arquivo especificado, que incluem os tempos e quantidades de cada procedimento, além

dos dados dos pacientes. Em seguida, as estruturas de dados necessárias são inicializadas, como as filas de prioridade (**MinHeap**) e as filas de serviço (**Queue**).

A simulação é realizada em um loop principal que continua até que todas as filas estejam vazias e não haja mais eventos a serem processados. Dentro do loop, o próximo paciente é extraído da fila de eventos (**Escalonador**) e processado de acordo com seu estado atual. Dependendo do estado, o paciente pode ser inserido em uma fila de triagem, atendimento, ou outros procedimentos, ou pode ser diretamente processado se uma unidade estiver disponível.

A cada iteração, a data atual é atualizada para o próximo evento, e o paciente é movido através dos diferentes estados até que seja liberado do hospital. As estatísticas dos pacientes são coletadas e impressas ao final da simulação.

A estrutura de dados principal é a MinHeap, implementada de forma generica para garantir seu funcionamento em diversas aréas do codigo. A MinHeap é a base para o Escalonador, onde o topo sempre será o proximo evento, ou seja com a menor data. A Heap também foi implementada para a fila, a fila tem uma estrutura onde se utiliza três heaps, uma para cada grau de urgencia, para garantir que o topo sempre será por alguém de urgencia maior e por menor tempo de entrada na fila, para evitar com que alguém que chegou primeiro seja atendido primeiro no mesmo grau de urgencia. Por fim ela também foi implementada para os servidores, onde a menor data de disponibilidade fica no topo para recuperar o proximo servidor disponivel e para as estatisticas para poder plotar por ordem de menor id as estatisticas, e salvar ordenadamente, pois nem sempre o menor id saia primeiro do hospital, assim evitando uma estrutura de ordenação nova.

2.1 MinHeap

Estrutura padrão de MinHeap, com uma diferença no parametro de ordenação, onde se pode escolher qual parametro especifico se usaria, como se fosse ordenação por meio de diferentes chaves.

- **heapifyUp**: Método auxiliar para manter a propriedade da heap após a inserção de um novo elemento. Ele compara o elemento no índice fornecido com seu pai e os troca se necessário, repetindo o processo até que a propriedade da heap seja restaurada.

- **heapifyDown**: Método auxiliar para manter a propriedade da heap após a remoção do elemento mínimo. Ele compara o elemento no índice

fornecido com seus filhos e os troca se necessário, repetindo o processo até que a propriedade da heap seja restaurada.

- **swap**: Método auxiliar para trocar dois ponteiros de elementos na heap.
- **insert**: Método público para inserir um novo elemento na heap. Ele adiciona o elemento no final da heap e chama o método *heapifyUp* para restaurar a propriedade da heap.
- **extractMin**: Método público para remover e retornar o elemento mínimo da heap. Ele troca o elemento mínimo com o último elemento, remove o último elemento e chama o método *heapifyDown* para restaurar a propriedade da heap.
- **viewMin**: Método público para visualizar o elemento mínimo da heap sem removê-lo.
- **isEmpty**: Método público para verificar se a heap está vazia.
- **getSize**: Método público para obter o número de elementos na heap.

2.2 Queue

A classe **Queue** é responsável por gerenciar as filas de pacientes com diferentes níveis de urgência. Ela utiliza três **MinHeaps** para organizar os pacientes nas filas vermelha, amarela e verde, de acordo com o nível de urgência.

2.3 Patient

A classe **Patient** representa um paciente no sistema hospitalar. Ela armazena informações como o ID do paciente, datas de admissão e alta, nível de urgência, medidas hospitalares, exames de laboratório, exames de imagem, medicamentos e estado atual. A classe fornece métodos para acessar e modificar esses atributos, além de métodos para atualizar o estado do paciente, consumir recursos hospitalares e calcular tempos de espera e atendimento.

A classe também possui um método para imprimir as informações do paciente de forma formatada.

2.4 Procedure

A classe **Procedure** é responsável por gerenciar os diferentes procedimentos hospitalares pelos quais um paciente pode passar. Ela utiliza várias **MinHeaps** para organizar as unidades de triagem, atendimento, medidas hospitalares, testes de laboratório, exames de imagem e instrumentos, de acordo com a disponibilidade de cada unidade.

- **Procedure**: Construtor da classe Procedure que inicializa as heaps de unidades para cada tipo de procedimento e define os tempos de duração de cada procedimento.
- **getNextServAvail**: Método que retorna a próxima data disponível para um determinado tipo de serviço.
- **occupyUnit**: Método auxiliar que ocupa uma unidade de um determinado tipo de procedimento, atualizando a data de disponibilidade da unidade.
- **updatePatientTime**: Método auxiliar que atualiza o tempo de atendimento de um paciente.
- **occupyUnitProcedure**: Métodos que ocupam unidades para cada tipo de procedimento hospitalar (triagem, atendimento, medidas hospitalares, testes de laboratório, exames de imagem e instrumentos), atualizando a data de disponibilidade da unidade e o tempo de atendimento do paciente.
- **processPatient**: Método que processa um paciente, ocupando a unidade correspondente ao estado atual do paciente.

A classe **Procedure** garante que os pacientes sejam atendidos de forma eficiente, ocupando as unidades disponíveis e atualizando os tempos de atendimento de acordo com os procedimentos realizados.

2.5 Date

A classe **Date** serve para manipular datas e horários sem depender de bibliotecas externas. Ela permite armazenar e formatar datas, além de calcular o dia da semana, mês e verificar anos bissextos. A classe também suporta operações de diferença em horas entre duas datas e operadores de comparação para facilitar seu uso em estruturas como MinHeap.

3 Análise de Complexidade

A análise de complexidade do código **main** considera tanto o tempo quanto o espaço necessários para a simulação do sistema hospitalar.

3.1 Complexidade de Tempo

A complexidade de tempo do código **main** pode ser dividida em várias partes:

- **Leitura do Arquivo de Entrada**: A leitura dos parâmetros e dos dados dos pacientes do arquivo de entrada tem complexidade $O(n)$, onde n é o número de pacientes.
- **Inserção na MinHeap**: A inserção de cada paciente na **MinHeap** (Escalonador) tem complexidade $O(\log n)$. Como há n pacientes, a complexidade total para a inserção é $O(n \log n)$.
- **Loop Principal da Simulação**: O loop principal processa cada paciente e realiza operações de inserção e extração na **MinHeap**, cada uma com complexidade $O(\log n)$. No pior caso, cada paciente pode passar por várias filas e procedimentos, mas o número de operações é limitado pelo número de pacientes e eventos, resultando em uma complexidade total de $O(n \log n)$.

Portanto, a complexidade de tempo total do código **main** é $O(n \log n)$.

3.2 Complexidade de Espaço

A complexidade de espaço do código **main** é determinada pelo armazenamento dos pacientes e das estruturas de dados utilizadas:

- **Armazenamento dos Pacientes**: Cada paciente é armazenado na **MinHeap** (Escalonador) e possivelmente em várias filas (**Queue**). No pior caso, todos os pacientes podem estar em filas ao mesmo tempo, resultando em uma complexidade de espaço de $O(n)$. As Estatisticas armazenam os pacientes que sairam o escalonador, servindo de complemento, não aumentando a ordem de complexidade.
- **Estruturas de Dados Auxiliares**: As estruturas de dados auxiliares, como as **MinHeaps** para triagem, atendimento e outros procedimentos, também contribuem para a complexidade de espaço. No entanto, o número de unidades de serviço é constante e não depende do número de pacientes, resultando em uma complexidade de espaço adicional constante $O(1)$.

Portanto, a complexidade de espaço total do código **main** é $O(n)$.

4 Estratégias de Robustez

Nesta seção, são descritas as estratégias de robustez implementadas no sistema para garantir a programação defensiva e a tolerância a falhas.

4.1 Programação Defensiva

A programação defensiva foi aplicada em várias partes do código para prevenir erros e comportamentos inesperados:

- **Verificação de Argumentos**: No início da função **main**, é verificado se o arquivo de entrada foi fornecido como argumento. Caso contrário, uma mensagem de erro é exibida e o programa é encerrado.
- **Verificação de Abertura de Arquivo**: Após tentar abrir o arquivo de entrada, é verificado se a operação foi bem-sucedida. Se o arquivo não puder ser aberto, uma mensagem de erro é exibida e o programa é encerrado.
- **Validação de Dados de Entrada**: Durante a leitura dos dados do arquivo de entrada, são realizadas verificações para garantir que os valores lidos são válidos e estão no formato esperado.
- **Tratamento de Exceções**: Em várias partes do código, são lançadas exceções para lidar com situações inesperadas, como níveis de urgência inválidos ou tentativas de extração de elementos de filas vazias.

4.2 Tolerância a Falhas

Para aumentar a tolerância a falhas, foram implementados mecanismos que permitem ao sistema continuar operando mesmo em caso de erros:

- **Estruturas de Dados Resilientes**: As estruturas de dados utilizadas, como **MinHeap** e **Queue**, foram projetadas para lidar com inserções e extrações de elementos de forma robusta, garantindo que a propriedade da heap seja mantida e que as filas estejam sempre em um estado consistente.

- **Reinicialização de Pacientes**: Caso um paciente não possa ser processado em um determinado estado devido à indisponibilidade de unidades de serviço, ele é reinserido na fila de eventos (**Escalonador**) para ser processado em uma iteração futura.

Essas estratégias garantem que o sistema seja robusto e capaz de lidar com diferentes tipos de falhas, mantendo a integridade dos dados e a continuidade da simulação.

5 Análise Experimental

Para testar o codigo foram disponibilizados algumas datas com informações de quantidade de servidores, para cada serviço, o tempo medio de serviço. Além da lista de pacientes com id, data, se teve alta após o atendimento e a quantidade de procedimentos realizados de cada serviço.

Os resultados para datas menores foi perfeito, o comportamento foi exatamente o esperadoo. O mesmo não posso dizer para datas maiores, existe uma certa incerteza sobre os comportamentos da fila, que não foram bem explicados e entendidos. Logo por mais que tenha dado um resltado bom e proximo ao fornecido como certo, porém o resultado não foi exato.

Uma explicação possivel é a ordem dos itens no escalonador, como ele define o item que vai ser atendido primeiro, pode acontecer de ocupar um servidor e deixar outro em fila que não poderia, mas não é possivel determinar com precisão como ele deve funcionar, se deve priorizar o que acabou de chegar, se foi o que chegou primeiro. O comportamento escolhido é que o menor tempo fica no topo e se for igual quem tiver o menor id vence. Não é necessario colocar uma priorização de estado, pois os mesmo estados só disputam entre si, eu tentei implementar outro metodo de ordenação na heap para considerar o grau de urgencia, caso a data seja igual coloque em cima quem tem a maior urgencia, se for igual o id, porém o resultado não mudou.

6 Conclusões

Neste trabalho, foi desemvolvido um software para simulação de eventos discretos em um sistema hospitalar, utilizando estruturas de dados como **MinHeap** e **Queue** para gerenciar filas de pacientes e servidores. A simulação foi projetada para otimizar o tempo de atendimento dos pacientes, levando em consideração diferentes níveis de urgência e a disponibilidade de unidades de serviço.

A implementação do sistema demonstrou ser eficiente para conjuntos de dados menores, apresentando resultados precisos e comportamentos esperados. No entanto, para conjuntos de dados maiores, observamos algumas incertezas no comportamento das filas, possivelmente devido à ordem de atendimento no escalonador. A escolha de priorizar o menor tempo e, em caso de empate, o menor ID, pode não ser a abordagem ideal em todos os cenários.

Aprendemos que a definição clara dos critérios de priorização é crucial para garantir a consistência dos resultados. Além disso, a implementação de estratégias de robustez, como a programação defensiva e a tolerância a falhas, foi fundamental para assegurar a integridade dos dados e a continuidade da simulação.

Em resumo, o trabalho alcançou seu objetivo de simular o tempo de permanência dos pacientes no hospital, considerando a disponibilidade de servidores e a prioridade dos casos mais urgentes. No entanto, melhorias podem ser feitas na definição dos critérios de priorização para obter resultados ainda mais precisos em cenários complexos.