Trabalho Prático da disciplina de Estrutura de Dados

Leticia Ribeiro Miranda
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
leticia-ribeiro98@hotmail.com

1. Introdução

O trabalho tem como objetivo a implementação e otimização de um sistema de filas de um determinado hospital. Para isso, o programa deve importar os dados dos pacientes a partir de um arquivo em formato csv e gerar uma saída com as seguintes informações de cada paciente: identificador, data e horário de entrada, data e horário de saída, tempo de permanência total e mínimo no hospital e tempo de espera nas filas.

Com o propósito de resolver o problema citado, optou-se por criar sete classes, *'AjustarSaidaPaciente'*, *'Escalonador'*, *'Fila'*, *'Paciente'*, *'Procedimento'*, *'ProcedimentoComFilas'* e *'TempoOcioso'*, que fizeram todo gerenciamento dos eventos e das filas, além de armazenar informações importantes sobre os tempos de permanência dos pacientes e ajustar as saídas esperadas. Por fim, foi feita a análise experimental, que inclui medições de tempo de relógio e recurso e utilização de ferramentas do CacheGrind e do Valgrind.

2. Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema operacional: Ubuntu 24.04.1 LTS
- Compilador: gcc (Ubuntu 13.2.0-23 ubuntu 4) 13.2.0
- Processador: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
- RAM: 16Gb (utilizável: 13,9 GB)

2.1. Estrutura de Dados

Para resolver o problema proposto, a implementação foi estruturada em sete arquivos de cabeçalho: *'AjustarSaidaPaciente.hpp'*, *'Escalonador.hpp'*, *'Fila.hpp'*, *'Paciente.hpp'*, *'Procedimento.hpp'*, *'ProcedimentoComFilas.hpp'* e *'TempoOcioso.hpp'*, os quais contêm a definição das classes, incluindo seus atributos e métodos. Em cada um desses arquivos, a primeira parte é dedicada à definição dos atributos privados das classes, enquanto a segunda parte descreve os construtores e os métodos públicos. Além disso, o programa conta com sete arquivos de implementação: *'AjustarSaidaPaciente.cpp'*, *'Escalonador.cpp'*, *'Fila.cpp'*, *'Paciente.cpp'*, *'Procedimento.cpp'*, *'ProcedimentoComFilas.cpp'* e *'TempoOcioso.cpp'*, que contém o desenvolvimento do comportamento das funções e métodos definidos nas classes correspondentes.

A classe *'AjustaSaidaPaciente.hpp'* serve mais como uma ferramenta para realizar ajustes e cálculos relacionados ao tempo no hospital dos pacientes. Ela ajusta a data e hora de saída de um paciente com base no tempo total de permanência, considerando possíveis

transições de dias, meses e/ou anos. Além disso, para a impressão correta da saída, fornece os dias da semana correspondentes às datas de entrada e saída do paciente.

A classe '*Escalonador'* implementa um sistema de gerenciamento de eventos baseado em um minheap, onde os eventos são organizados por prioridade, considerando ano, mês, dia, hora e ID do paciente. Ela permite inserir novos eventos, remover o próximo evento prioritário e consultar qual data e horário que ele foi inserido, garantindo que o menor e mais antigo paciente esteja sempre no topo. Em caso de empate nos critérios de priorização, o desempate é feito pelo identificador do paciente, dando prioridade ao menor ID.

A classe *'Paciente'* é projetada para gerenciar informações sobre os pacientes. Dos métodos implementados, vale destaque para esses quatros:

- O método *'setEstado'* controla em que procedimento o paciente se encontra. Começando com o valor 0, indicando que o paciente está na triagem, e progredindo até o valor 7, quando o paciente recebe alta hospitalar.
- O método '*setPermanencia'* registra o tempo em que o paciente ficou nos procedimentos e nas filas de espera, adicionando esse valor na variável '*tempoTotalPermanencia'*.
- A variável '*tempoControleHoras'* é inicializada com a hora em que o paciente entrou no hospital e é incrementada durante a permanência dele no estabelecimento. Caso a variável ultrapasse o limite de 24 horas, significa que o paciente permaneceu no hospital por mais dias. Quando isso ocorre, a data de permanência do paciente é atualizada e, utilizando o método '*setControleHoras'*, o valor dessa variável é resetado, iniciando uma nova contagem a partir desse momento.
- O método '*calcularTempoMinimoHospital'*, calcula o tempo mínimo de permanência do paciente no hospital.

Na classe *'Fila'*, a implementação é feita através de uma estrutura circular para gerenciar a fila de pacientes. O método '*enfileira'* é responsável por adicionar o paciente na fila dos procedimentos, enquanto o *'desenfileira'* remove o paciente mais antigo dela.

A classe *'Procedimento'* tem a finalidade de gerenciar um procedimento médico, incluindo informações sobre seu nome, tempo médio de execução e a disponibilidade de unidades para realizá-lo. Ela oferece métodos para verificar se há unidades disponíveis, ocupá-las e liberá-las.

Já a classe *'ProcedimentoComFilas'* faz a gerência dos pacientes com diferentes graus de urgência. Ela é herdeira de *'Procedimento'* e utiliza os métodos da *'Fila'* por meio de composição. Além disso, possui três filas, organizadas por níveis de prioridade e, com base nesse critério, permite adicionar pacientes à fila correspondente. Por último, a classe obtém o próximo paciente para atendimento, priorizando as filas de maior urgência clínica.

Por fim, a classe *'TempoOcioso'* é responsável por calcular e formatar o tempo ocioso dos pacientes, utilizando datas e horários fornecidos. Ela converte ano, mês, dia e hora para um formato de tempo, o que facilita os cálculos. A classe possui métodos para configurar os horários de entrada e saída de uma fila de espera e calcula o tempo transcorrido entre esses dois pontos.

O programa principal, *‘main.cpp’*, utiliza as definições e funcionalidades contidas nas classes supracitadas. Inicialmente ele implementa a função *‘ajustarTempoParaNovoDia’*, que tem como objetivo gerenciar a transição de dias, meses e anos caso a hora do paciente no hospital (*‘getControleHoras()’*) ou o horário do hospital (*‘tempoConclusao’)* seja maior que 24 horas.

A função main começa lendo o arquivo de entrada e em seguida lê e armazena os parâmetros que fornecem informações sobre o tempo médio de execução dos procedimentos e a quantidade de pacientes. Posteriormente, o escalonador é inicializado para inserir os pacientes nos eventos.

Antes de gerenciar os eventos, os objetos de *‘Procedimento’* e *‘ProcedimentoComFilas’* são criados, representando diferentes filas que os pacientes irão passar. No caso da triagem, como inicialmente o paciente não tem grau de urgência, então será gerado apenas uma fila por ordem de chegada. Em seguida, cria um array de ponteiros de *‘Paciente’* e gera os pacientes a partir dos dados fornecidos no arquivo.

Finalmente, o programa chega na parte principal da implementação, o laço de controle de eventos que simula o fluxo de pacientes por diferentes estágios no processo hospitalar. Nele contém dois ‘*While’*: O primeiro, um loop externo, que controla a execução da simulação enquanto houver eventos a serem processados ou se qualquer uma das filas não estiver vazia. O segundo, um loop interno, que é usado para processar todos os eventos programados para o mesmo instante de tempo.

No loop interno, o código entra na fila de eventos do escalonador e, com base no estado atual do paciente, determina em que fase o paciente se encontra no processo hospitalar. O funcionamento de cada procedimento é determinado por três funções:

- Caso tenha unidades disponíveis, ele chama a função *‘atualizarEstadoPaciente’* que marca a unidade como ocupada, incrementa o tempo do procedimento no tempo total permanência do paciente, além de calcular o momento de conclusão do procedimento. Ela ajusta a data de conclusão, caso dure mais de um dia, insere o evento no escalonador e, por fim, atualiza o estado do paciente para o próximo procedimento.
- Quando um paciente entra na fila de espera, chama-se a função *‘processarEntradaNaFila’*. Ela registra o momento de entrada do paciente (para o cálculo do tempo ocioso) e o coloca na fila daquele procedimento.
- Quando um procedimento é finalizado, a função *‘liberarProcedimentoEProcessarProximo’* é chamada. Inicialmente, ela desocupa uma unidade e, se a fila do procedimento daquela unidade não estiver vazia, a função obtém o próximo paciente da fila, obedecendo o critério do grau de urgência. Além disso, a função ocupa a unidade novamente com este paciente, calcula o tempo de ociosidade e atualiza o tempo total de permanência do paciente com base nesse valor de espera, somado a duração do procedimento atual. Em seguida, ela ajusta a data de conclusão do procedimento, caso ultrapasse 24 horas, atualiza o estado do paciente para o próximo procedimento e insere um evento no escalonador com a nova data e hora de conclusão.

Ao final, o programa processa todas as informações dos pacientes, imprime os dados formatados e garante a limpeza da memória alocada.

3. Análise de Complexidade

A análise de complexidade foi feita individualmente em cada classe, e, em seguida, foi analisada a interação entre elas no programa principal, determinando o impacto combinado das suas operações.

- **AjustarSaidaPaciente**

A complexidade de tempo e espaço da classe é O(1), pois todas as operações são limitadas a um número fixo de iterações ou cálculos diretos e não há alocação dinâmica de memória já que todas as variáveis são de tamanho fixo.

- **Escalonador**

O construtor e o método *'inicializa()'* apenas inicializam o tamanho como zero, ambos têm complexidade de tempo O(1). A estrutura do heap já está alocada e ocupa espaço O(n), onde n = 1000.

Para inserir um evento ou retirar o próximo evento, o escalonador simula uma árvore como um vetor, isso garante que a profundidade das folhas irá diferir em no máximo 1. Logo, a árvore estará sempre balanceada e a complexidade de tempo será O(log n), sendo *'n'* o número de eventos inseridos no momento. Além disso, nenhuma alocação adicional é feita além de variáveis temporárias, então o espaço é O(1).

Portanto, a complexidade de tempo para a classe *'Escalonador'* é O(log n) e a complexidade de espaço é igual a O(n), devido à estrutura do heap.

- **Fila**

O construtor, o método *'inicializa()'* e *'finaliza'* apenas inicializam o tamanho da fila como zero e libera a memória alocada, logo ambos têm complexidade de tempo O(1). Para o construtor e o *'finaliza'*, a complexidade de espaço é O(1). Já o método *'inicializa()'* aloca espaço proporcional a quantidade *'n'* de pacientes, então a complexidade de espaço dele é O(n).

Para enfileirar e desenfileirar, o método adiciona um paciente ao final da fila ou remove o paciente no início da fila e nenhum espaço adicional é alocado. Logo, a complexidade de tempo e espaço é O(1).

Portanto, a classe *'Fila'* possui complexidade de tempo O(1) e complexidade de espaço igual a O(n), uma vez que depende do tamanho máximo da capacidade da fila.

- **Paciente**

A classe *'Paciente'* possui complexidade de tempo e espaço constante, O(1), pois as operações realizadas são limitadas a um número fixo de cálculos e não dependem do tamanho dos dados de entrada.

Construtor padrão apenas inicializa as variáveis membros com valores padrão. A complexidade de tempo é O(1). Além disso, o construtor parametrizado apenas configura as

variáveis membros com os valores fornecidos como argumentos, logo a complexidade de tempo é O(1) também.

A maioria das variáveis membros da classe são do tipo int ou float, que tem complexidade de espaço igual a O(1) cada. Elas incluem dados como *'id'*, *'ano'*, *'mes'*, *'dia'*, *'hora'*, *'grauUrgencia'*, entre outros. Membros de dados adicionais como '*tempoMinimoPermanencia'*, *'tempoTotalPermanencia'*, etc., também são O(1) em termos de complexidade de espaço, pois são variáveis do tipo primitivo.

Todos os métodos de acesso (getters e setters) apenas retornam ou alteram os valores das variáveis membros, o que ocorre em tempo constante e não dependem de espaço adicional. Portanto, para cada getter ou setter a complexidade de tempo e espaço é O(1).

Por fim, *'setPermanencia'* e *'calcularTempoMinimoHospital'* tem complexidade de tempo e espaço O(1), pois a quantidade de operações é fixa e não depende de nenhuma entrada.

- **Procedimento**

A classe *'Procedimento'* apresenta uma complexidade de espaço constante, O(1), já que possui apenas três atributos ('*tempoMedioExecucao'*, *'unidadesDisponiveis'* e '*unidadesOcupadas'*), todos de tamanho fixo. Em termos de complexidade de tempo, todas as operações realizadas pelos métodos da classe também são O(1), incluindo o construtor, que apenas inicializa os atributos, métodos e os getters, que envolvem operações simples de comparação, incremento, decremento ou retorno de valores.

- **ProcedimentoComFilas**

O construtor inicia as três filas, onde cada alocação depende do tamanho *'n'* das filas, logo a complexidade de tempo e espaço é O(n). Já os demais métodos têm complexidade de tempo e espaço O(1), pois apenas adicionam, verificam e comparam os elementos da fila, sem a necessidade de espaço adicional.

Portanto, a complexidade de tempo e espaço da classe é O(n) pois depende da quantidade de pacientes que as filas suportam.

- **TempoOcioso**

Todos os métodos têm complexidade de tempo O(1), ou seja, eles executam em tempo constante, independentemente dos valores fornecidos como entrada. Além disso, a complexidade de espaço O(1), pois todas as operações utilizam um número fixo de variáveis locais e a alocação de memória não depende do tamanho da entrada.

- **Main**

O programa lê *'n'* linhas do arquivo (uma por paciente). As informações das linhas são processadas individualmente, logo a complexidade de tempo é O(n).

Durante a fase de inicialização, o valor do número de eventos no heap é sempre proporcional ao número de pacientes processados até o momento. Como o programa inicialmente insere todos os *'n'* pacientes, o tamanho do escalonador aumenta de 0 a n, portanto a complexidade de tempo nesta etapa é O(n log n).

No loop principal, cada paciente gera múltiplos eventos, um para cada procedimento, assim a inserção ou remoção de eventos tem complexidade O(m * n log (m * n)), sendo *'m'* a quantidade de procedimentos e *'n'* o número de pacientes. Como há 6 procedimentos distintos, a expressão pode ser simplificada para O(n log n).

A inserção ou remoção das filas tem complexidade de tempo O(1), pois foram implementadas como estruturas circulares simples. Cada paciente passa pelas filas de diferentes graus de urgência (vermelha, amarela e verde) durante os 6 procedimentos. Portanto, a complexidade de tempo é O(n), sendo *'n'* a quantidade de pacientes.

O ajuste de horários dos pacientes com *'ajustarTempoParaNovoDia'* tem uma complexidade de tempo de O(h), onde *'h'* é o valor de *'tempoConclusao'* ou *'controleHoras'*, pois o loop while executa até que esses valores sejam menores que 24.

Na etapa final, os horários de saída de cada paciente são configurados com *'ajustarSaidaPaciente'*. A complexidade de tempo da função é O(n), onde *'n'* é o número de pacientes. A razão para isso é que o código contém um laço que percorre todos os pacientes, realizando operações constantes para cada um deles.

Após a análise, pode-se concluir que a complexidade de tempo do programa é O(n log n).

A explicação é que, embora outras partes do código realizem operações com complexidade O(n), a maior contribuição para o tempo de execução vem do escalonador, que processa eventos com complexidade O(n log n). Isso ocorre porque, além do crescimento linear, o fator logarítmico adiciona um crescimento extra, tornando o termo dominante na análise assintótica.

Comparação do crescimento das funções *'nlogn'*, representada pela curva vermelha e *'n'*, representada pela curva roxa. Note que a primeira função cresce muito mais rápido.

Em relação a complexidade de espaço, cada paciente ocupa O(1) espaço ao armazenar seus dados, para *'n'* pacientes, o espaço total será O(n). Já o escalonador, utiliza um minheap que armazena eventos inseridos. Como há *'m * n'* eventos totais, sendo *'m'* o número de procedimentos que é igual a 6 e *'n'* o número de pacientes, então a complexidade de espaço será O(n).

Ao todo, são 16 filas, uma para triagem e 3 para cada procedimento, sendo 'n' a capacidade total das filas que é igual ao número de pacientes, então a complexidade de espaço será

O(n). As demais variáveis para leitura e processamento de dados tem complexidade de espaço O(1).

Portanto, após o estudo, conclui-se que a complexidade de espaço do algoritmo é de O(n).

4. Estratégias de Robustez

No desenvolvimento do código, foram implementadas estratégias de robustez para garantir que o programa lidasse com possíveis erros e condições inesperadas, são elas:

- Verificação de abertura de Arquivo: No main, o código verifica se o arquivo foi aberto corretamente, caso contrário, exibe uma mensagem de erro e interrompe a execução.
- Limitação de tamanho: No escalonador, quando é inserido um evento, verifica se o escalonador já está em sua capacidade máxima, caso afirmativo, exibe uma mensagem de erro e encerra o programa. Já, ao remover ou verificar a data e o tempo do próximo evento, se o escalonador estiver vazio, um '*std::runtime_error'* é lançado, indicando claramente que não há eventos a serem retirados ou datas e tempos disponíveis, evitando comportamentos indefinidos ou erros de acesso a dados inexistentes.
- Verificação de fila cheia ou vazia: No método enfileira(), antes de adicionar um paciente à fila, o código verifica se ela já está cheia, comparando o tamanho com a capacidade. Caso afirmativo, um *'std::overflow_error'* é lançado, evitando a inserção de um novo paciente e mantendo a integridade da fila. Já no método desenfileira(), é feita uma verificação para garantir que a fila não esteja vazia antes de tentar retirar um paciente. Se estiver, um '*std::underflow_error'* é lançado, o que impede a tentativa de retirar um paciente quando não há nenhum na fila.
- Verificação de pontos de tempo indefinidos: Na classe *'TempoOcioso'*, no método *'calcularTempoOcioso'* verifica se os valores de entrada e saída são *'std::chrono::system_clock::time_point()'* (valor padrão). Caso algum dos pontos não estejam definidos, ele lança uma exceção com uma mensagem.
- Falhas nas conversões: Na classe *'TempoOcioso'*, no método *'construirTimePoint'*, a função std::mktime converte uma estrutura std::tm para um valor de tipo std::time_t. Se essa conversão falhar, será lançada uma exceção com uma mensagem indicando que houve um erro.

No geral, essas práticas ajudaram a garantir que o código fosse mais coeso, livre de erros e eficiente.

5. Análise Experimental

Na etapa de análise experimental foram utilizadas três bibliotecas para medir o tempo de relógio e recursos do programa.

Para medir o tempo de relógio, foi usada a função '*high_resolution_clock::now()'* da biblioteca chrono, que registra o tempo inicial e o tempo final ao redor da execução da função alvo. A diferença entre esses dois instantes é calculada e armazenada em uma variável do tipo '*duration<double, std::milli>'*, que converte a duração para milissegundos.

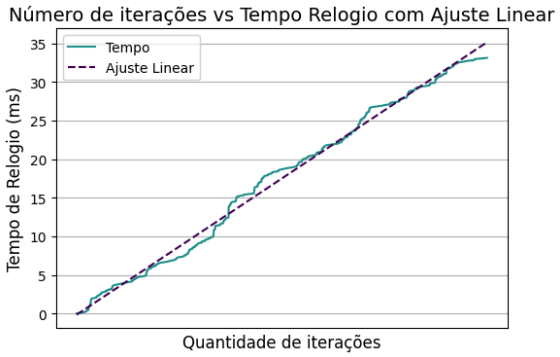
Já para os tempos de utilização de recursos (usuário e sistema), usou-se a função clock_gettime que pertence à biblioteca padrão do sistema no Linux, *'time.h'*. Calculou-se a diferença entre o início e fim da execução da função alvo para obter as durações.

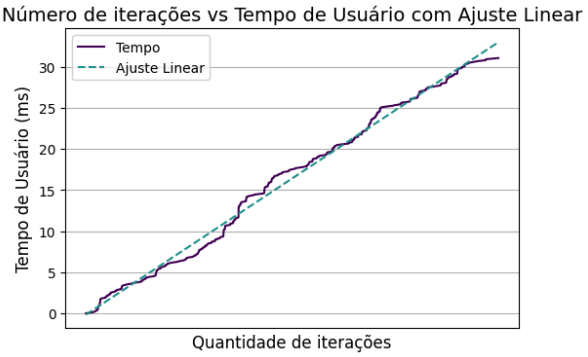
Além disso, foram utilizadas as ferramentas Cachegrind, que simula a hierarquia de memória e mede o desempenho de cache, e o Valgrind, que analisa as funções mais chamadas.

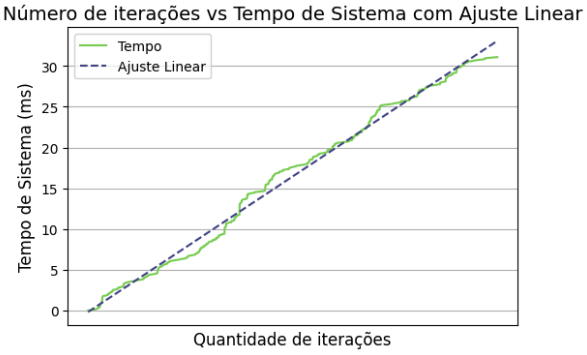
Por fim, em todas as análises, utilizaram-se uma base de dados com 1000 registros, totalizando 36.758 mil caracteres. Após esse limite, o programa apresentou um erro de segmentation fault, o que restringiu a quantidade de dados analisados.

Nos gráficos abaixo, observa-se uma tendência linear entre a quantidade de iterações e o tempo. À medida que o número de pacientes aumenta, consequentemente o número de iterações que o programa realiza também aumenta e o tempo de relógio, de usuário e de sistema cresce linearmente.

Com base na regressão linear, pode-se concluir que, para cada incremento de 1 unidade no eixo x, o tempo aumenta, em média, 0.0162 milissegundos (ms) para o de relógio e 0.0152 milissegundos (ms) para de usuário e de sistema. Isso sugere uma relação direta e previsível entre o número de pacientes e o tempo total registrado, embora o gráfico também revele pequenas flutuações ao redor da reta de ajuste.







Porém, essa análise não reflete a complexidade de tempo do programa, que é O(n log n), isso ocorre porque a curva logarítmica é muito semelhante à curva linear quando o número de dados é pequeno. Portanto, no gráfico gerado, o comportamento acaba se parecendo mais com uma curva linear.

Executando o código utilizando o Cachegrind, especificamente com a ferramenta cg_annotate, foram analisadas as principais funções que impactam no total de instruções executadas.

O programa executou o total de 156.481.766 instruções, sendo que 62.787.412 (40,1%) delas vêm do arquivo escalonador.cpp. O que já era esperado, visto que ele é o elemento central da simulação de eventos discretos e toda iteração é em torno dos eventos inseridos ou retirados do escalonador.

Dentro desta classe, o método *'heapifyDown(int)'* é o maior responsável pelo consumo de instruções, representando 26,6% (41.662.748) do total desse valor. Já o *'heapifyUp(int)'* vem em segundo lugar, consumindo 9,7% (15.243.297) de instruções.

A análise indica que as operações de manipulação do minheap ('*heapifyDown*' e '*heapifyUp*') são encarregadas por uma porção considerável do programa. Focar na otimização delas e examinar se há uma maneira de reduzir o número de chamadas ou melhorar sua eficiência pode proporcionar ganhos de desempenho.

Por fim, através do comando *'kcachegrind callgrind.out.<pid>'*, analisou-se as funções mais chamadas e quanto tempo de CPU foi consumido por cada uma delas.

Na figura a seguir, nota-se que os métodos *'retiraProximoEvento()'* e *'InsereEvento()'* consomem, respectivamente, 7,86% e 28,78% do tempo total de CPU, refletindo o impacto significativo das operações de inserção e remoção de eventos no heap.

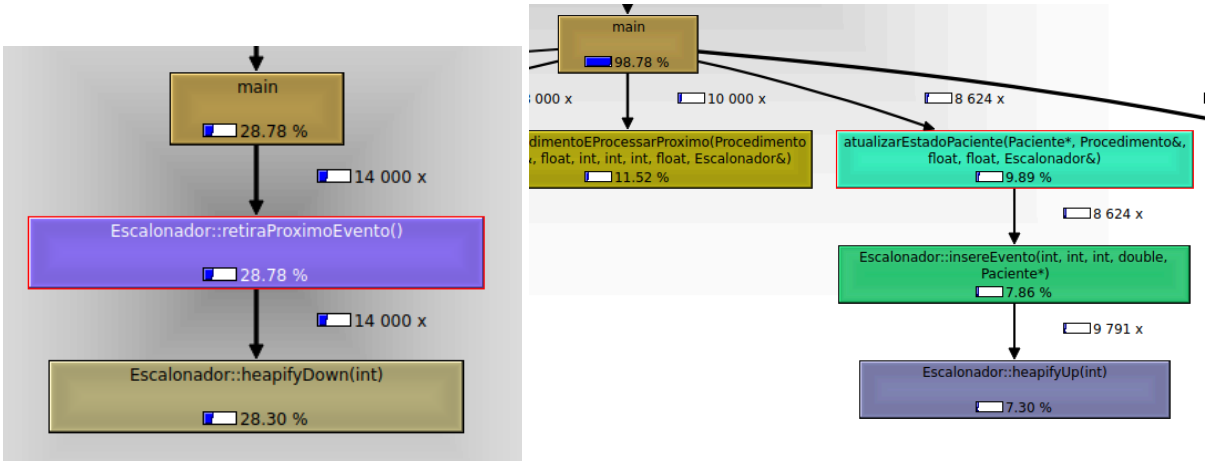


Figura que representa os tempos de execução, retirada do kcachegrind.

Esse resultado já era esperado, pois, como destacado anteriormente, o escalonador desempenha um papel central na simulação de um sistema de filas hospitalares e é o principal responsável pela complexidade de tempo do programa.

6. Conclusão

Este trabalho lidou com o problema de implementação e otimização de um sistema de filas, na qual a abordagem utilizada para a solução foi desenvolver um escalonador como minheap, que gerencia os eventos e recupera sempre o evento de menor data e hora, e criar uma estrutura circular de filas, que controla os pacientes que estão em espera do procedimento. Além disso, foram desenvolvidas classes para gerenciar a ocupação e

desocupação das unidades disponíveis. Por fim, os demais métodos deram suporte no controle das datas, dos horários e do tempo de permanência e ocioso do paciente.

Com a solução adotada, pode-se verificar que o sistema apresentou maior eficiência na organização e gerenciamento das filas. O escalonador baseado em minheap permitiu que os eventos fossem processados de forma eficiente, priorizando os pacientes com maior urgência hospitalar e/ou com mais tempo de espera, garantindo um fluxo contínuo e organizado. As filas foram separadas em três níveis, dependendo do grau de urgência, o que permitiu uma gestão mais precisa das prioridades. A divisão do programa em várias classes possibilitou uma estrutura de código organizada, facilitando a implementação de melhorias e permitindo a identificação e correção de problemas rapidamente.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados à TADs, implementação de filas e árvores binárias. Durante a implementação da solução houveram importantes desafios a serem superados, a principal delas era no controle das datas e horários para atualizar corretamente o tempo de espera e o tempo de permanência total do paciente, já que existia a possibilidade dele ficar mais dias no hospital.

Este estudo reforça a importância de estruturar um código eficiente e bem organizado que garanta o funcionamento otimizado e confiável do sistema de filas.

7. Bibliografia

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.