Documentação do Projeto de Programação 1

Simulador de Senhas

Alunos: Bruna Azambuja e Ricardo Santos
Matrículas: 18/0014153 e 18/0027263
Disciplina Estrutura de Dados – turma E (2/2018)
Professor George Teodoro

S u m á r i o

1. Proposta do Projeto

Uma instituição laboratorial, Nibas, está tendo problemas com o sistema de atendimento, e tem de enfrentar diversos clientes insatisfeitos com a demora e a falta de organização nas filas. Para resolver tal problema e amenizar as reações de seus clientes, foi contratado um programador para fazer um sistema que aumente a eficácia do atendimento.

Este projeto tem como proposta um simulador, que faz a priorização de atendimento e cria um sistema de senhas para organizar a distribuição nas filas de acordo com três parâmetros: a prioridade física, prioridade por idade e o tempo de chegada dos pacientes, a fim de tornar o sistema mais eficiente e rápido.

Sabendo que o laboratório possui cinco tipos diferentes de serviço para cada guichê de atendimento, e que por dia, o número mínimo de guichês que estarão funcionando é um por serviço, o programa irá receber uma configuração no início do dia com o número total de guichês funcionais e seus respectivos serviços, que chamaremos de agora em diante de Arquivo Configuração.

O grande desafio desse projeto é organizar os pacientes à medida que eles chegam ao laboratório, de forma que otimize o processo. Para isso, programa irá receber também, outro arquivo, que será chamado nesse projeto de Arquivo Carga Trabalho, que possui todos os pacientes com seu horário de chegada, idade, condição física e o serviço desejado.

Para que o atendimento seja otimizado de maneira justa, foi definido que pessoas com prioridade maior são inseridas na fila na frente de pessoas com menor prioridade, mesmo que tenham chegado depois. Para casos em que a prioridade dos pacientes é igual, a organização se dará pela ordem de chegada.

2. Como o algoritmo funciona ?

O alogoritmo contará com três parâmetros de entrada, como já foi comentado anteriormente. O primeiro será o Arquivo de Configuração, que contará com quantos guichês na primeira linha e seus respectivos serviços, na ordem em que aparecem, e terá o seguinte formato:

```
---------------------------------------------------------------------
      N – número total de guichês;
      X1 – função do guichê de número 1;
      X2 – função do guichê de número 2;
      ...
      Xn – função do guichê N;
---------------------------------------------------------------------
```

O segundo será o Arquivo Carga Trabalho, que contará com a configuração dos pacientes e suas características, e terá o seguinte formato:

```
---------------------------------------------------------------------
      Y1 Y2 Y3 Y4 – tempo de chegada, idade, serviço e condição física
do paciente 1;
      ...
      Y1 Y2 Y3 Y4 - tempo de chegada, idade, serviço e condição física
do paciente n;
---------------------------------------------------------------------
```

Sabendo que os pacientes são organizados por ordem de chegada no arquivo, e que todos os casos são equiprováveis – numa certa unidade de tempo ninguém chegar, mais de um paciente chegar ou apenas um chegar.

O terceiro parâmetro será o nome do arquivo em que será escrita a saída. Com esses três parâmetros, a execução do programa deverá se dar com a seguinte linha:

./sim_senhas arquivo_configuracao arquivo_carga_trabalho arquivo_saida

A lógica do programa se dará por um simulador que insere os pacientes nas filas à medida que eles chegam ao laboratório, por ordem de prioridade, e os retira da fila quando algum guichê que estiver vazio o chame. Cada guichê possuirá um countdown que é inicializado com o tempo que seu respectivo serviço demora, esse countdown passará a diminuir assim que o paciente for chamado pelo guichê, e, quando esse contador chegar em zero – ou seja, a pessoa terminou de ser atendida – o guichê passa a estar vazio novamente.

Os tempos de cada serviço e seus respectivos códigos são fornecidos pelo laboratório, e nesse caso são dados por:

- Hemograma simples – códiogo (0) – tempo: 5 unidades;
- Pré-natal – código (1) – tempo: 10 unidades;
- Vacinação – código (2) – tempo: 8 unidades;
- Hemograma completo – código (3) – tempo: 7 unidades;
- Entrega de resultados – código (4) – tempo: 2 unidades.

Após todas as pessoas atendidas, o programa fará um relatório do dia, gerando uma saída que contém o tempo de espera médio dos clientes na fila, a quantidade média de pacientes atendidos por unidade de tempo, e em seguida, um relatório de cada cliente atendido naquele dia.

A saída será impressa no arquivo de saída, e terá o seguinte formato:

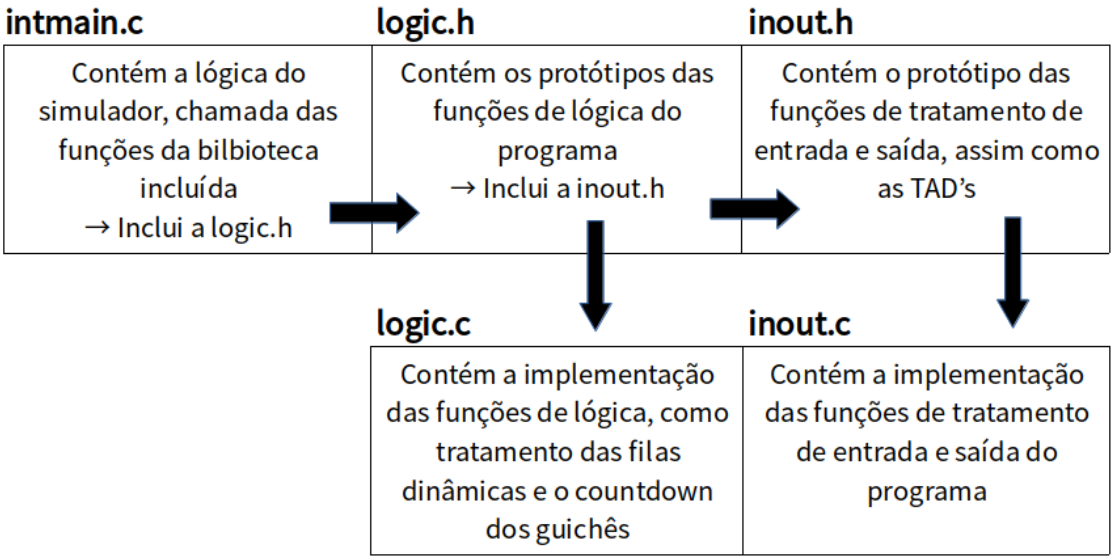
```
---------------------------------------------------------------------
    Z1 Z2 – tempo de espera médio dos clientes na fila e quantidade
média de clientes atendidos por unidade de tempo;
    W1 W2 W3 W4 – guichê em que foi atendido, prioridade, tempo
esperado e serviço do paciente 1;
    ...
    W1 W2 W3 W4 - guichê em que foi atendido, prioridade, tempo
esperado e serviço do paciente n;
---------------------------------------------------------------------
```

3. Como é dada a implementação ?

3.1. Separação dos módulos e sua inter-dependência

O programa foi separado em três módulos, sendo um para o programa principal (.c) – que será chamado de intmain, e dois (.c) que contém o protótipo das funções utilizadas na main, com suas respectivas bibliotecas (.h) – que serão chamados de inout e logic.

O diagrama das principais funções de cada módulo e a interdependência entre eles se encontra abaixo:



3.2. Descrição das Estruturas de Dados utilizadas

As TAD's utilizadas foram organizadas da seguinte forma:

- struct pessoa: {
  int chegada, condição física, prioridade, idade, serviço, guichê em que foi atendido, tempo de espera, número da pessoa; → *será usada para guardar as informações de um paciente.*
  }
- **elemento*** apontador; → *será usado como ponteiro de uma struct elemento.*
- struct elemento: {
  apontador próximo; → *apontará para a próxima estrutura elemento, que por sua vez terá o campo próximo e o campo dados .*
  pessoa dados; → *conterá os dados de uma struct do tipo pessoa .*
  }
- struct lista: {
  apontador primeiro; → *apontará para a primeira struct elemento da lista dinâmica.*
  apontador último; → *apontará para a última struct elemento da lista dinâmica.*
  }
- struct guichê {
  int número do guichê, contador, flag; → *guarda o número do guichê, o countdown e uma flag (0- vazio, 1- cheio).*

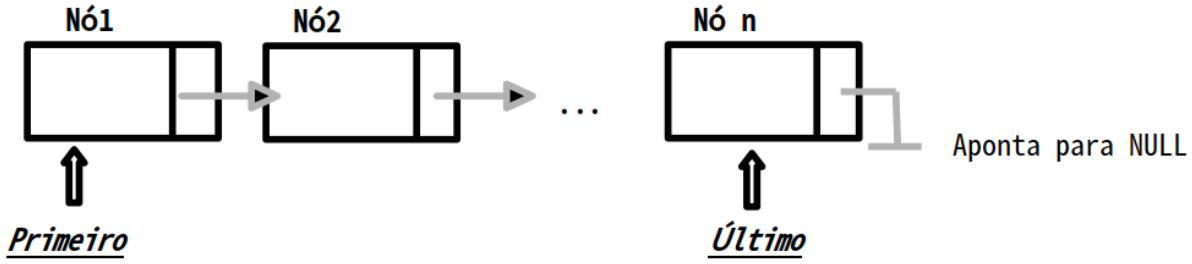
pessoa pessoa atual sendo atendida; → *conterá os dados da pessoa que está sendo atendida pelo guichê.*

}

3.3. Implementando as TAD's

As TAD's descritas no tópico anterior foram implementadas de diferentes formas, para diferentes funções. A struct pessoa foi utilizada apenas para armazenar os dados dos pacientes.

Já a struct elemento, possui o formato do que chamaremos de Nó de uma lista dinâmica. Sendo assim, esse Nó possui um campo de *dados*, que serão preenchidos, nesse caso, com uma struct pessoa, e o campo *próximo*, que armazena um ponteiro para outro Nó, com o mesmo formato do primeiro. Com a implementação dessa forma, conseguimos fazer uma lista dinâmica, ou seja, uma lista de Nós, onde cada um aponta para seu sucessor, sem tamanho fixo. Quando chegamos ao último da lista, seu apontador aponta para NULL.



A struct lista, por sua vez, armazena apenas duas informações; nesse caso, ponteiros para o primeiro e o último Nó da respetiva lista. Fazendo uma analogia, numa fila para um determinado atendimento, essa struct fila possuiria o "endereço" da primeira e da última pessoa da fila.

Por fim, a struct guichê possui os dados do guichê, como número, contador e a flag; e o campo de dados da pessoa que o guichê está atualmente atendendo. Caso o guichê esteja vazio, sua flag é zerada, assim como o campo de dados da pessoa atual.

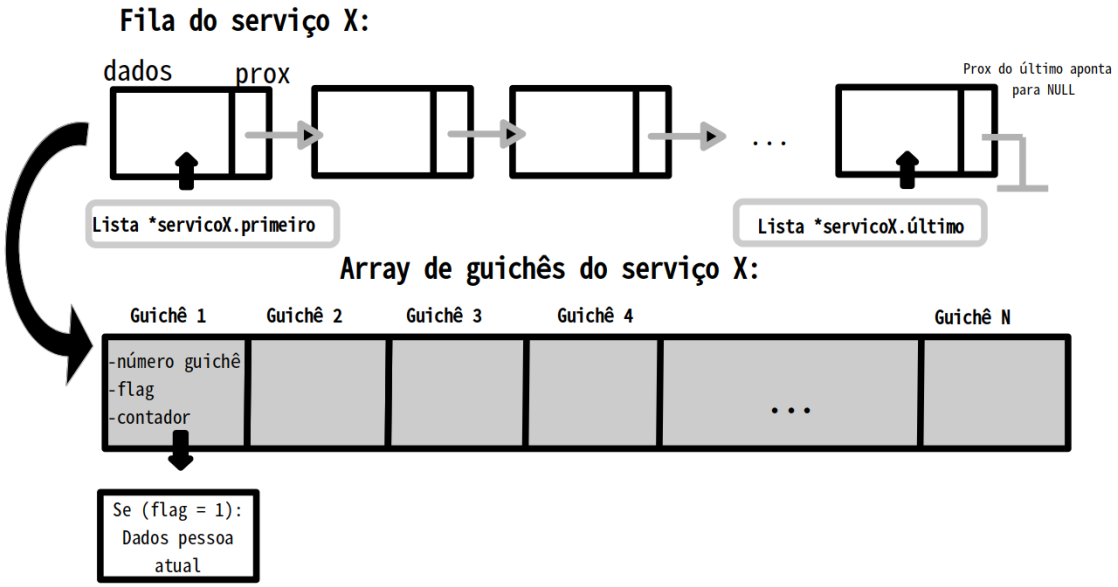
4. Listagem do programa fonte

Depois de explicado como cada Estrutura de Dados se comporta, será mostrado como essas estruturas foram utilizadas na aplicação da lógica do programa, passo a passo, descrevendo, função por função, o relevante do programa principal.

4.1. Como as TAD's foram utilizadas

As estruturas de dados foram organizadas da seguinte forma:

1. Uma lista auxiliar dinâmica que contém todas as pessoas do Arquivo Carga Trabalho;
2. Uma lista de resposta dinâmica que será utilizada para printar a saída;
3. Cinco filas dinâmicas, uma de cada serviço;
4. Cinco arrays, um de cada serviço, em que cada array contém [número de guichês daquele serviço] espaços.



4.2. Tratamento das entradas

Após a inicialização de todas as variáveis e estruturas que irão ser usadas, fez-se o tratamento de entradas do programa. Para a alocação dos cinco arrays de guichês, chamou-se a função *entradaguiches* , que retorna os valores dos tamanhos que cada vetor possuirá- ou seja, quantos guichês de cada serviço, e os usa para fazer as respectivas alocações dinamicamente.

Criados os arrays, chamou-se a função *define_numero_guiche_ordem_arquivo* – que define o número do guichê de acordo com a ordem de sua entrada do arquivo, e em seguida a *crialista* para cada lista que será usada nesse programa. Na lista auxiliar, já antes comentada, inserem-se todos os pacientes do Arquivo Carga Trabalho, que já está organizado por ordem de chegada. Ao inserir, calculamos a prioridade do paciente, e seu número (novamente de acordo com sua ordem no arquivo). Feito isso, tem-se uma lista dinâmica a qual contém as informações dos pacientes na mesma configuração do arquivo de entrada.

4.3. Tratamento da lógica

Inicia-se um loop infinito (*while(1))* com uma variável *unidade de tempo* inicializada com zero, em que serão feitas as seguintes operações, nessa ordem:

1. *le_pacientes* – percorre a lista auxiliar, verificando se alguém chegou na unidade de tempo atual. Se sim, insere essas pessoas nas filas dos respectivos serviços por ordem de prioridade e chegada;

2. retira da lista auxiliar todo mundo que chegou na unidade de tempo atual;

3. *manda_primeiro_da_fila_pros_guiches* – verifica se tem alguém na fila e se tem algum guichê vazio, se sim, tira a primeira pessoa da fila e manda ela para o guichê. Inicia-se o countdown do guichê com o tempo de atendimento de cada serviço. Assim, o guichê passa a estar cheio (flag = 1);

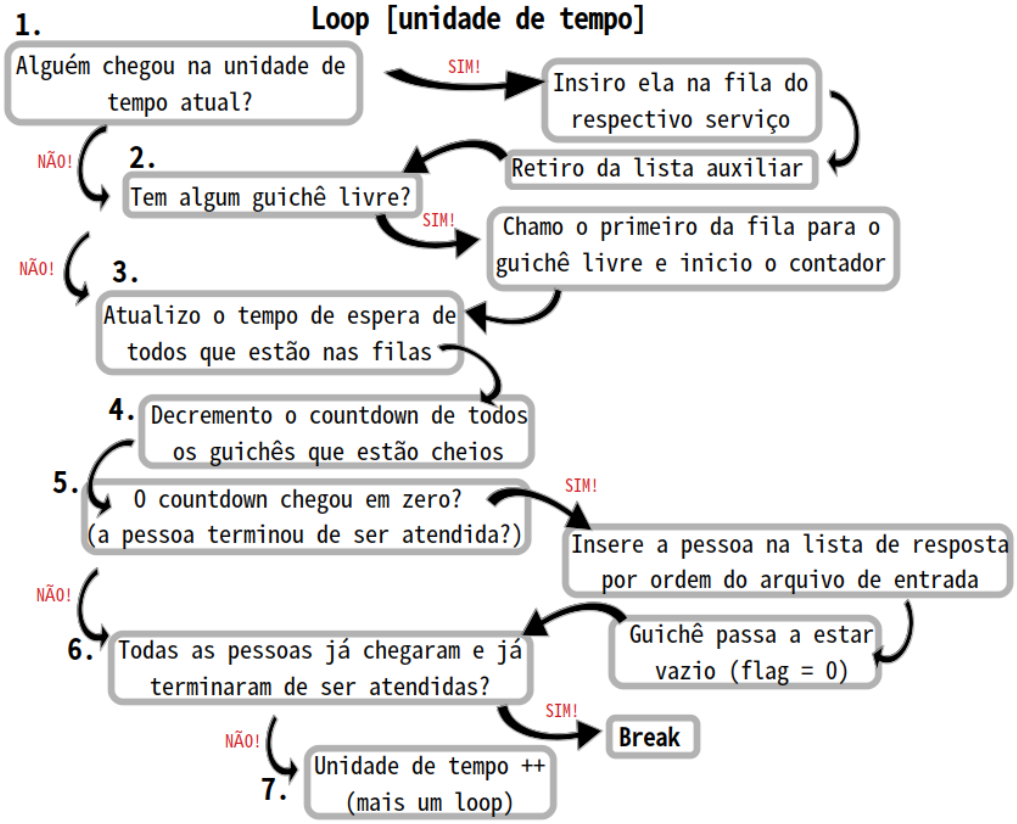
4. *atualiza_tempo_espera* – verifica se tem alguém na fila, se sim, percorre ela inteira atualizando o tempo de espera de cada paciente em +1;

5. *atualiza_count_down_insere_resp* – percorre todo o array de guichês, caso o guichê esteja cheio (flag = 1), decrementa o countdown em -1. Caso o countdown chegue em zero, *insere_lista_ordenada* – insere a pessoa que estava sendo atendida na lista de resposta, por ordem do número atribuído à ela (mesma ordem do arquivo de entrada);

6. *verifica_caso_parada* – verifica se todas as filas de todos os serviços estão vazias, bem como a lista auxiliar – ou seja, todos foram atendidos. Caso sim, sai do loop; caso contrário…

7. ...incrementa a unidade de tempo e roda mais um loop.

→ Desenho esquemático representando o processo de um loop:



4.4. Cálculo do relatório de saída

Os cálculos dos dados pedidos no relatório do funcionamento do simulador foram dados por:

1. **tempo de espera médio =** (Somatório de todos os tempos de espera) / (total de pacientes);
2. **quantidade média de pacientes atendidos por unidade de tempo =** (total de pacientes) / (unidade de tempo em que o loop parou).

Após o simulador completo e todos os dados calculados, desaloca-se o espaço usado no programa. Função *desaloca* – percorre toda a lista

auxiliar e a lista de resposta, dando free(nó por nó). Bem como nos arrays de guichês - que foram alocados dinamicamente.

5. Como é dada a compilação ?

Para a compilação, foi criado um arquivo Make File, que trata de todo o processo de compilação necessário para o funcionamento do programa. Nesse arquivo, incluiu-se a compilação tanto da inout.c quanto da logic.c, bem como a do programa principal – intmain.c.

Esse método de compilação se dá muito eficiente pois, além de reduzir o número de comandos necessários para compilar o programa inteiro, ele otimiza o processo, verificando a data de compilação de cada módulo. Ou seja, cada .c só será compilada novamente se alguma modificação foi feita desde a última vez.

Para utilizar o Make File basta digitar o comando “make” no terminal, e a saída será a compilação dos objetos que foram modificados.

6. Estudo da complexidade

Para termos uma noção melhor da eficiência do simulador, foi feita uma análise da complexidade de cada função das duas bibliotecas incluídas. Essa análise será comentada abaixo, bem como uma breve descrição do que a função associada faz, os parâmetros recebidos por ela e seu tipo.

Função para tratamento de arquivos:

1. `FILE* abrearquivo(char *nome, char *funcao);`

Recebe o nome do arquivo bem como o modo que este arquivo deverá ser aberto, caso não exista, cria um arquivo com esse nome. Complexidade O(1).

Funções para tratamento das TAD’s:

2. `void crialista(Lista *li);`

Recebe o ponteiro de uma struct tipo lista, aloca o espaço de um Nó (struct elemento) e força o ponteiro do primeiro e do último apontarem para esse mesmo Nó, bem como o ponteiro do próximo apontando para NULL – lista vazia. Complexidade: é uma função com 3 ações, traz uma complexidade +3 cada vez que é chamada - O(1).

3. `void insere_lista_final(Lista *li, pessoa paciente);`

Recebe o ponteiro da lista e a struct pessoa que queremos inserir. Alocamos o espaço de um novo Nó e fazemos o ponteiro do último apontar para ele, lincando o antigo último com o novo último (ultimo→prox = nó, ultimo = ultimo→prox), por fim preenchemos os dados do novo nó com a struct paciente. Complexidade: é uma função com 4 ações, acarreta num aumento de +4 cada vez que é chamada. O(1).

4. `void insere_lista_auxiliar(FILE *fp, Lista *li);`

Recebe o ponteiro do Arquivo Carga Trabalho e da lista que iremos inserir. Lê o arquivo inteiro, calcula a prioridade e o número do paciente, e o insere no final da lista. Complexidade: a função ocorrerá n vezes, sendo n o número de pacientes no arquivo de entrada. O(n).

5. `void insere_lista_ordenada(Lista* li, pessoa paciente, int parametro);`

Recebe o ponteiro da lista, o paciente que será inserido e um parâmetro. Caso o parâmetro seja 0, inserimos pela ordem do arquivo de entrada, caso seja 1, inserimos por ordem de prioridade. Percorremos a lista inteira, procurando o lugar onde devemos inserir, uma vez achada a posição, o ponteiro do anterior→prox passa a apontar para o novo Nó, e o Nó→prox aponta para o elemento atual. Complexidade: A função é muito maleável e pode apresentar comportamentos diferentes dependendo do parâmetro em que está sendo executada assim como o estado das filas. Isso faz com que um pior caso seja muito difícil de ser encontrado. Porém de uma forma geral, a complexidade da função irá variar de acordo com o tamanho da lista de pessoas que já chegaram e aguardam para ser atendidas. Apresentando, portanto, complexidade O(n) – no pior caso, todos os pacientes estarão nessa lista.

Funções para tratamento dos dados:

7. `int calcula_prioridade(pessoa paciente);`

Recebe um paciente, e de acordo com as normas já ditas anteriormente, calcula sua prioridade. Complexidade: no pior caso, faz 5 comparações, portanto apresenta O(1).

8. `void entradaguiches(FILE *fp, int *t1, int *t2, int *t3, int *t4, int *t5);`

Recebe o ponteiro do Arquivo Configuração e o endereço das variáveis respectivas aos tamanhos de cada array de guichês. Lê o arquivo inteiro e atualiza o valor dos tamanhos. Complexidade: a função ocorre q + 1 vezes, sendo q a quantidade de guichês. Portanto uma complexidade O(q).

9. `void define_numero_guiche_por_ordem_no_arquivo(FILE* fp, guiche *p1, guiche *p2, guiche *p3, guiche *p4, guiche *p5);`

Recebe o ponteiro do Arquivo Configuração, e o endereço dos arrays de guichês. Lê o arquivo inteiro, e vai preenchendo os campos de número do guichê por ordem no arquivo, assim como inicializando a flag com zero. Complexidade O(q).

10. `void le_pacientes(Lista *li_aux, Lista *li, int parametro, int unidadetempo);`

Recebe o ponteiro da lista auxiliar, e da fila individual do serviço [parâmetro], e a unidade de tempo atual. Percorre a lista auxiliar, e vai inserindo todos os pacientes que chegaram na unidade de tempo atual, por ordem de prioridade, na fila. Complexidade: a função utiliza a função insere_lista_ordenada dentro dela, sabendo também que ela irá rodar n vezes, então sua complexidade fica com $O(n^2)$.

Funções para printar e acompanhar o processo:

11. `void printa_resposta_arquivo(FILE *fp, Lista *li);`

Recebe o ponteiro do arquivo de saída, assim como o ponteiro da lista de resposta. Percorre a lista inteira, printando cada paciente no arquivo. Complexidade: O(n).

12. `void printa_lista(Lista *li, int parametro);`

Recebe o ponteiro da lista e um parâmetro para identificá-la. Percorre a lista inteira e vai printando no terminal os dados de cada paciente. Complexidade: A função printa lista terá uma complexidade variável com o tamanho. Nos primeiros casos, no qual o parâmetro vai de 0 a 4, ordenando todos eles em um só bloco têm-se uma complexidade igual a O(q). No parâmetro “100”, a complexidade será dado por O(n). No parâmetro “500”, a complexidade será dada por j (número de pessoas ja atendidas).

Funções com respeito à lógica do programa:

13. `void manda_primeiro_da_fila_pros_guiches(guiche *p, int tam, Lista *li, int temposervico);`

Recebe o ponteiro do array de guichês e da lista do serviço X, assim como o tamanho do array e o tempo de atendimento desse serviço. Percorre o array inteiro, caso tenha um guichê vazio, tiramos a pessoa da fila, botamos ela no guichê, inicializamos o contador com o tempo de serviço e mudamos a flag para 1. Complexidade: a função possui 7 ações no pior caso e irá percorrer por todos os guichês de dado serviço, portanto O(q).

14. `void atualiza_tempo_espera(Lista *li);`

Recebe o ponteiro da fila do serviço X. Percorre a lista até o final, incrementando o tempo esperado de cada paciente. Complexidade: O pior caso dessa função irá ocorrer quando todos os pacientes estiverem na lista de espera,

ou seja, cada um chegou e está esperando pra ser atendido para um único serviço. A complexidade é portanto igual a n - tam, ou seja, o número total de pacientes menos a quantidade de guichês para o serviço, O(n).

15. `void atualiza_count_down_insere_resp(guiche *p, int tam, Lista *resp);`

Recebe o ponteiro do array de guichês, seu tamanho, e o ponteiro da lista de resposta. Percorre todos os guichês decrementando em 1 o contador. Caso chegue em zero (pessoa já foi atendida), inserimos a pessoa na lista de resposta por ordem do Arquivo Carga Trabalho, e mudamos a flag para 0. Complexidade: O(q).

16. `int verifica_caso_parada(Lista *li, guiche *p, int tam);`

Recebe o ponteiro da fila do serviço X, assim como do array de guichês e seu tamanho. Verifica se o primeiro e o último apontam para o mesmo Nó (fila está vazia), e se todos os guichês têm flag = 0 (todos estão vazios). Complexidade: a função tem uma ação e um loop que rodará q vezes, portanto O(q).

17. `float calcula_tempo_medio_total(Lista *li, int *cont);`

Recebe o ponteiro da lista de resposta e o endereço de um contador. Percorre a lista inteira incrementando esse contador em um a cada nó (armazenará, então, o número de pacientes), e somando numa variável local os tempos de espera. Divide o tempo de espera total pelo número de pacientes e retorna o tempo médio. Complexidade: a função percorre por todos os pacientes, portanto O(n).

18. `void desaloca(apontador p);`

Recebe o ponteiro que aponta para o início da lista, desalocando Nó a Nó. Complexidade: a função desaloca o espaço usado para os pacientes, portanto O(n).

Funções com respeito à lógica do programa:

19. Funções clock:

A função é chamada 2 vezes e possui complexidade O(1);

20. Função while(1) - Simulador:

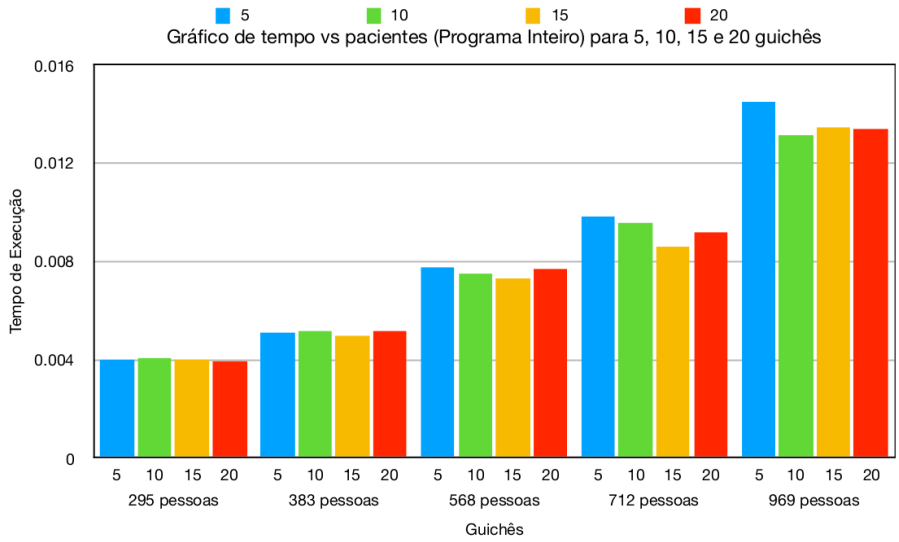
A complexiadade do simulador é muito maleável, pois depende muito de quantas vezes o loop irá rodar - [unidade de tempo] vezes. A pior complexidade encontrada dentre as funções chamadas no do loop, será da função insere_lista_ordenada, a qual ocorre $O(n^2)$.

7. Estudo do tempo de execução

Outra importante análise para determinar a funcionalidade de um programa, é a análise do tempo de execução. De nada adianta um simulador que nunca termina de rodar, ou que demora um tempo inacessível. “Um programa modesto que funciona é mais útil que um super-ambicioso que nunca funciona.”

Para essa análise em específico, os dados foram coletados experimentalmente. Fixou-se um número de guichês, variando o número de pessoas, e o tempo de execução foi calculado. Num outro momento, fixou-se o número de pessoas, e variou-se o número de guichês – mantendo sempre o mesmo número de guichês para cada serviço, e o tempo de execução foi novamente calculado. Com os dados em mãos, foi possível perceber qual seria o comportamento do simulador para cada variável.

Os dados foram coletados quatro vezes para cada alteração, e a média entre esses valores foi tirada, para que se possa ter um valor médio ótimo. Para construir o gráfico das tabelas com os dados acima, usou-se o valor médio, e as duas variações foram colocadas num mesmo gráfico para que se possa observar a diferença de comportamento.



Analisando o gráfico, conseguimos perceber o natural – o simulador demora mais para um conjunto de pacientes maior. Porém já de acordo com a variação no número de guichês, podemos perceber que não necessariamente um número maior de guichês faz com que o simulador rode mais rápido. Muito pelo contrário, o tempo permanece parcialmente constante para números de guichês diferentes, apenas para 5 guichês a função demora mais para conjuntos maiores de pacientes.

Assim, pode-se definir um número ótimo de guichês entre 10 e 15. Não é necessário mais que esse valor, e menor, como já vimos, se torna eficiente para conjuntos muito grandes.

8. Programa Gerador de Entradas

Uma parte importante do trabalho foi testar o simulador com arquivos criados aleatoriamente. Para a criação desses arquivos de entrada para o programa principal, foi feito um programa auxiliar capaz de gerar os dados dos guichês e dos pacientes.

O programa foi feito da seguinte forma: utilizou-se das funções rand e sarna para a criação de números aleatórios.

A função 'srand' é importante para que a seed utilizada pela função 'rand' seja diferente a cada vez que ela é chamada, isso faz com que o programa sempre gere números diferentes dos anteriores. Caso não seja feito isso, o programa sempre que executado irá apresentar apenas uma vez os valores aleatórios, e repetí-los nas próximas execuções. A linha *srand((unsigned) time(&t))*, faz com que isso não aconteça.

Para que os números gerados aleatoriamente estejam dentro de um intervalo utilizou-se da operação %. Quando se utiliza o conjunto rand % X, o número gerado estará dentro do intervalo de 0 a X. Pode-se também implementar da seguinte forma: rand % X + 1; dessa forma o valor gerado estará entre 1 e X + 1.

Essa ideia foi utilizada na função Random_Numbers que gera um valor entre 1 e o valor requerido.

A função main então ocorre da seguinte forma: primeiro são abertos dois arquivos onde serão escritos os valores “arquivo_configuracao" e “arquivo_carga_trabalho”, com a função “w”, para que sempre sejam sobrescritos os últimos valores pelos novos.

Em seguida, é gerado um número aleatório de guichês entre 5 e 25. Esse valor é escrito no arquivo de configuração, seguido de uma quantidade de guichês linhas, com valores entre 0 e 4, que representarão os serviços dos guichês.

Seguidamente, é gerado um valor que representará a quantidade de pacientes atendidos naquele dia; um valor entre 1 e 1000. Esse número é utilizado para alocar dinamicamente espaço para um vetor de structs que guardará as informações desses pacientes, que também serão gerados aleatoriamente.

Após geradas as informações sobre os pacientes e guardados no vetor, o mesmo é organizado, utilizando-se o insertion sort, de acordo com a hora de chegada do paciente.

Após organizado, são escritas as informações no segundo arquivo da carga do trabalho.

Por fim, os arquivos são fechados e o vetor de desalocado.

9. Testes executados

Para que se possa ter noção do funcionamento do simulador, foram realizados diversos casos de teste. Alguns deles serão listados abaixo:

```
Número de Guichês Gerados: 5
Número de Pacientes Gerados: 10
tempo médio de espera dos clientes na fila: 2.00
unidade de tempo: 31
quantidade média de clientes por unidade de tempo: 0.32

Número de Guichês Gerados: 22
Número de Pacientes Gerados: 158
tempo médio de espera dos clientes na fila: 0.00
unidade de tempo: 730
quantidade média de clientes por unidade de tempo: 0.22

Número de Guichês Gerados: 20
Número de Pacientes Gerados: 467
tempo médio de espera dos clientes na fila: 16.49
unidade de tempo: 788
quantidade média de clientes por unidade de tempo: 0.59
```

Assim, com esses dados em mãos, podemos perceber o já esperado: com uma quantidade grande de pacientes, o tempo de espera destes na fila, é maior; bem como a quantidade média de clientes por unidade de tempo. Ou seja, não só mais clientes são atendidos, mas também os clientes esperam mais para serem atendidos.