Trabalho Prático 2
Servidor de emails otimizado

Guilherme Mota Bromonschenkel Lima - 2019027571

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

guilhermebkel@ufmg.br

1. Introdução

O problema proposto foi implementar um simulador de servidor de emails, o qual deve possuir um gerenciamento adequado de memória além de um foco na otimização da pesquisa por usuários e mensagens. Nesse sistema podemos executar as ações de entrega, consulta e remoção de mensagens.

Ao executar o programa, ele faz a leitura de um arquivo contendo o tamanho da HashTable que será utilizada, além das ações que serão executadas dentro do servidor.

Ao fim da execução do programa, é criado um arquivo de saída que contém a descrição de cada ação efetuada, como a identificação da mensagem e do usuário, além de informações de erro quando alguma ação não tem um retorno esperado.

Durante a execução do programa, fizemos uso de um módulo chamado **memlog** para fazer diversos registros de acesso de memória. Além disso, para realizar a análise experimental foi necessário utilizar a biblioteca **analisamem**, que é responsável por pegar os dados gerados pela biblioteca **memlog** e tratá-los para uma melhor visualização.

Por fim, fazemos uso também de uma ferramenta chamada **gprof** para verificar métricas de desempenho computacional, como a duração de chamadas de funções do programa.

2. Implementação

O programa foi desenvolvido na linguagem C, compilada pelo compilador GCC da GNU Compiler Collection.

Os testes do programa foram realizados no sistema operacional Ubuntu (versão 18.04).

2.1. Estrutura de Dados

Para que o programa funcionasse da melhor forma possível, fizemos uso de quatro Tipos Abstratos de Dados (TADs) principais, que serão descritos logo abaixo.

Nesse sentido, fizemos uso de dois modelos principais, que foram o Mailer (responsável por executar as ações de consulta, entrega e remoção de mensagens) e o Message (responsável por armazenar as informações de identificação e conteúdo da mensagem).

Além disso, a estrutura de gerenciamento das mensagens ficou por conta de uma combinação de uma HashTable (segue a ideia de ser o servidor de email) com uma Árvore Binária (segue a ideia de ser a caixa de entrada das mensagens).

2.2. Formato dos dados

Durante a execução do programa é feita a leitura de um arquivo de entrada para obter os dados que serão utilizados mais tarde.

No início do arquivo temos a informação do tamanho da HashTable que será utilizada no servidor.

Logo em seguida, temos as ações que serão executadas dentro do servidor, que seguem o formato abaixo:

1. ENTREGA “U” “E” “N” “MSG” *(Entregar uma mensagem)*
    - U: Identificador do Usuário
    - E: Identificador do E-mail
    - N: Quantidade de palavras da mensagem
    - MSG: Conteúdo da mensagem
2. CONSULTA “U” “E” *(Consultar uma mensagem)*
    - U: Identificador do Usuário
    - E: Identificador do E-mail
3. APAGA “U” “E” *(Remover uma mensagem)*
    - U: Identificador do Usuário
    - E: Identificador do E-mail

2.4. Modularização

Para garantir uma melhor qualidade de código e facilidade de manutenção, temos o programa como um todo dividido em 10 módulos com obrigações diferentes:

O módulo **mailer** (é a classe responsável por realizar as regras de negócio para gerar os resultados das ações efetuadas no servidor de e-mail).

O módulo **shared-util** (é o contexto responsável por fornecer métodos utilitários compartilhados entre todos os módulos existentes).

O módulo **hash-table** (é a estrutura de dados de lista utilizada pelo **mailer** para gerenciar as informações do servidor de e-mail).

O módulo **binary-tree** (é a estrutura de dados de lista utilizada pelo **hash-table** para gerenciar as mensagens da caixa de entrada do servidor de e-mail).

O módulo **binary-tree-validation** (é a estrutura de dados de lista utilizada pelo **binary-tree** para validar informações).

O módulo **memlog** (realiza a gravação de acessos de memória durante a execução do programa)

O módulo **app** (funciona como um orquestrador do programa, fazendo a leitura dos dados de entrada e utilizando o **mailer** para processar esses dados e gerar um arquivo de saída).

O módulo **app-validation** (é o contexto responsável por fornecer métodos que validem os dados que são utilizados durante a execução do **app**).

O módulo **app-util** (é o contexto responsável por fornecer os métodos utilitários que são utilizados durante a execução do **app**).

O módulo **message** (é a estrutura de dados utilizada como modelo para a mensagem da caixa de entrada do servidor de e-mail).

3. Análise de Complexidade

A análise de complexidade do programa foi realizada utilizando como base o módulo **app**, que é o ponto de entrada de execução de todas as regras de negócio.

3.1. Tempo

Dentro do módulo do **app**, ocorre a execução de um loop de leitura e dos seguintes métodos (que podemos considerar como os mais impactantes):

- **Mailer.read;**
- **Mailer.send;**
- **Mailer.remove**.

Ou seja, vamos iniciar analisando a complexidade de cada um desses métodos para depois analisar a complexidade do módulo **app** como um todo.

No método **Mailer.read**, nós realizamos uma busca dentro da HashTable $O(1)$, em seguida fazemos uma busca dentro da BinaryTree $O(log\ n)$ / $O(n)$. Ou seja, nesse método temos a seguinte complexidade:

Melhor caso = $O(log\ n)$

Pior Caso = $O(n)$

No método **Mailer.send**, nós realizamos uma busca dentro da HashTable $O(1)$, em seguida fazemos uma inserção dentro da BinaryTree $O(log\ n)$ / $O(n)$. Ou seja, nesse método temos a seguinte complexidade:

Ou seja, teremos a seguinte complexidade para este método:

Melhor caso = $O(log\ n)$

Pior Caso = $O(n)$

No método **Mailer.remove**, nós realizamos uma busca dentro da HashTable $O(1)$, em seguida fazemos uma remoção dentro da BinaryTree $O(log\ n)$ / $O(n)$. Ou seja, nesse método temos a seguinte complexidade:

Melhor caso = $O(log\ n)$

Pior Caso = $O(n)$

Agora que analisamos cada um dos métodos, podemos voltar para a análise do módulo **app**.

Temos um loop principal de n-repetições com as seguintes ações: verificar qual o tipo de ação $O(1)$, Mailer.read $O(log\ n)$ / $O(n)$ e Mailer.remove $O(log\ n)$ / $O(n)$. Ou seja, nessa etapa teremos a seguinte complexidade:

Melhor caso = $O(n\ log\ n)$

Pior Caso = $O(n^2)$

Quando a ação envolve entregar uma mensagem, temos um loop de n-repetições para ler todas as palavras da mensagem $O(n)$ e por fim, usar o método Mailer.send $O(log\ n)$ / $O(n)$ para realizar o envio da mensagem. Ou seja, nesse método temos a seguinte complexidade:

Melhor caso = Pior Caso = $O(n^2)$

Dessa forma, podemos concluir que a nossa complexidade temporal para o módulo do **app** como um todo pode ser dada por:

1. **Caso nenhuma mensagem for entregue:**

$$\text{Melhor caso} = O(n \log n)$$

$$\text{Pior Caso} = O(n^2)$$

2. **Caso alguma mensagem for entregue:**

$$\text{Melhor caso} = \text{Pior Caso} = O(n^2)$$

3.2. Espaço

Nesse programa, devido a necessidade de armazenar as mensagens na caixa de entrada, podemos dizer que teremos a seguinte complexidade de espaço:

$$Complexidade\ Espacial\ = O(n)$$

4. Estratégia de Robustez

Quando estamos lidando com uma HashTable, temos um problema muito comum que é o de colisão. Nesse sentido, existe uma validação extra durante a leitura de uma mensagem para garantir que a mensagem pesquisada é realmente a mensagem do usuário especificado pelos argumentos do método.

Além disso, foram utilizadas validações para garantir que o comando correto seja executado de acordo com as entradas fornecidas pelo arquivo inicial do programa.

Por fim, foram realizadas validações durante a execução principal do programa para garantir que o arquivo de entrada existe e também que o arquivo de saída foi gerado com sucesso.

5. Testes

Foi realizada uma série de testes para garantir que todas as operações estão sendo executadas corretamente. Logo abaixo é possível ver uma entrada que foi utilizada em um dos testes:

- Entrada:

23
ENTREGA 5 103 6 Bom dia, meu amigo! Tudo bom?
ENTREGA 6 104 8 Boa tarde, minha amiga! Vou bem e você?
ENTREGA 5 105 18 Também! Não estou encontrando meu casaco, será que o deixei na sua casa no fim de semana passado?
ENTREGA 6 106 8 É um casaco longo e vermelho com bolsos?
ENTREGA 5 107 3 Sim, este mesmo.
ENTREGA 6 108 6 Está aqui, vou guardá-lo para você.
ENTREGA 5 109 8 Obrigada! Vou buscá-lo no próximo sábado. Até lá!
ENTREGA 6 110 1 Até!
CONSULTA 5 103
CONSULTA 6 104
CONSULTA 5 107
APAGA 5 103
CONSULTA 5 103
CONSULTA 6 110

Com a entrada acima, fomos capazes de obter a saída esperada.

- Saída

OK: MENSAGEM 103 PARA 5 ARMAZENADA EM 5
OK: MENSAGEM 104 PARA 6 ARMAZENADA EM 6
OK: MENSAGEM 105 PARA 5 ARMAZENADA EM 5
OK: MENSAGEM 106 PARA 6 ARMAZENADA EM 6
OK: MENSAGEM 107 PARA 5 ARMAZENADA EM 5
OK: MENSAGEM 108 PARA 6 ARMAZENADA EM 6
OK: MENSAGEM 109 PARA 5 ARMAZENADA EM 5
OK: MENSAGEM 110 PARA 6 ARMAZENADA EM 6
CONSULTA 5 103: Bom dia, meu amigo! Tudo bom?
CONSULTA 6 104: Boa tarde, minha amiga! Vou bem e você?
CONSULTA 5 107: Sim, este mesmo.
OK: MENSAGEM APAGADA
CONSULTA 5 103: MENSAGEM INEXISTENTE
CONSULTA 6 110: Até!

6. Análise Experimental

Agora, podemos efetuar uma análise de desempenho computacional e eficiência de acesso à memória.

6.1. Desempenho Computacional

Para esse experimento, vamos analisar algumas dimensões existentes nesse programa, usando as seguintes entradas:

- Número de usuários: vamos testar algumas quantidades de usuários variados (1000 usuários, 2000 usuários, 3000 usuários, 4000 usuários, 5000 usuários).
- Número de mensagens: vamos testar algumas quantidades de mensagens variadas em um único usuário (1000 mensagens, 2000 mensagens, 3000 mensagens, 4000 mensagens, 5000).
- Tamanho das mensagens: vamos testar a criação de menos de 10 mensagens com algumas quantidades de caracteres variados (1000 caracteres, 2000 caracteres, 3000 caracteres, 4000 caracteres, 5000 caracteres).
- Distribuição de frequência de operações: vamos testar algumas quantidades de operações variadas para um único usuário com 1000 mensagens na caixa de entrada (1000 operações de entrega, 1000 operações de consulta e 1000 operações de exclusão).

Vamos usar a métrica de tempo de execução de cada operação como métrica relevante para a avaliação do nosso programa, visto que é uma métrica que nos possibilita entender o custo de cada execução na prática e fazer uma comparação com a análise de complexidade que fizemos anteriormente. Sendo assim, o tempo de execução (em segundos) para cada quantidade de rodadas pode ser visto na tabela abaixo:

Quantidade de Usuários	**Tempo de execução** *(segundos)*
1000	0.013455684
2000	0.075629526
3000	0.116616108
4000	0.185244765
5000	0.236262561

No caso acima, podemos ver que como estamos apenas criando novas caixas de entrada (pois estamos literalmente enviando uma mensagem para cada usuário novo), o tempo de execução do programa aumenta minimamente de acordo com a quantidade de usuários, visto que estamos apenas inserindo uma informação dentro da HashTable.

Quantidade de Mensagens	**Tempo de execução** *(segundos)*
1000	0.174789440
2000	0.638972838
3000	1.902164597
4000	3.619866849
5000	5.068959025

No caso acima, podemos notar que como estamos gerando uma quantidade de mensagens apenas para um usuário, estamos tendo que lidar com a operação de inserção apenas em uma única árvore binária. Logo, a quantidade de nós dessa árvore binária corresponde à quantidade de mensagens dos testes acima. Dessa forma, estamos sujeitos à ordem de complexidade de uma BinaryTree.

Quantidade de Caracteres	**Tempo de execução** *(segundos)*
1000	0.001692932
2000	0.005742246
3000	0.007341212
4000	0.008319351
5000	0.012740971

No caso acima, como estamos apenas alterando o tamanho das mensagens inseridas na caixa de entrada e mantendo a mesma quantidade de mensagens sempre, é possível perceber que o tempo de execução aumenta em intervalos extremamente pequenos. Isso se deve ao fato de que temos uma complexidade linear para pegar cada palavra e criar o texto que será utilizado na mensagem da caixa de entrada.

Quantidade de Operações	**Tempo de execução** *(segundos)*
1000 entregas	0.149939006
1000 consultas	0.197060670
1000 exclusões	0.192058726

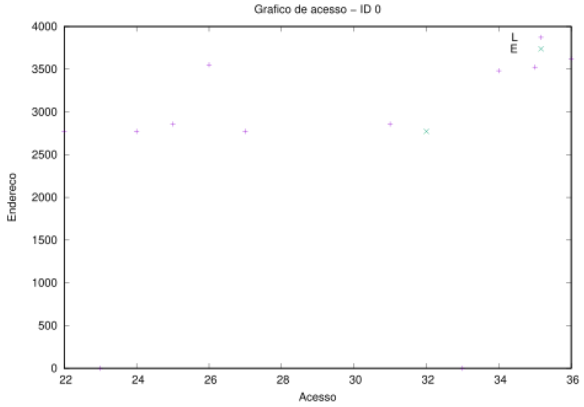
No caso acima, é possível ver que tivemos um tempo de execução maior nas consultas e exclusões do que nas de entrega, porém isso se deve ao fato de que a caixa de entrada já estava com 1000 mensagens, ou seja, houve 1000 entregas em cada um dos testes.

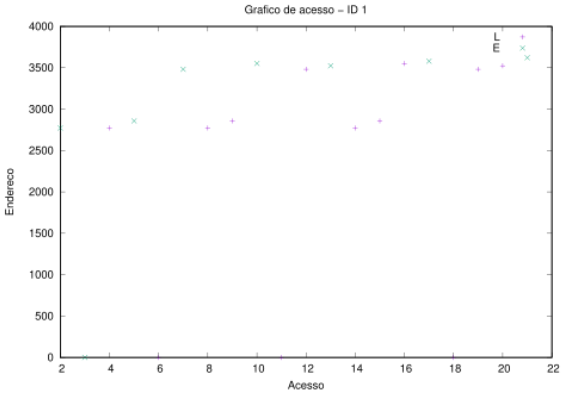
Conforme vimos na nossa análise de complexidade, quando efetuamos ações de entrega de mensagem é o momento em que temos o pior caso do nosso programa. Sendo assim, ao analisar os dados adquiridos acima, é possível entender as variações que nosso programa pode ter de acordo com as entradas fornecidas e, mesmo assim, nosso pior caso continua sendo o que encontramos na análise de complexidade.

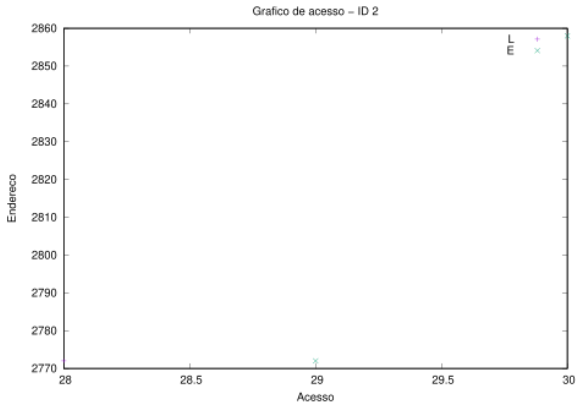
Por fim, ao utilizar o **gprof** para analisar cada operação efetuada em uma inserção de 5000 mensagens na caixa de entrada, é possível entender que o maior tempo de execução vem do método **Mailer.send**, no qual ocorre a inserção da mensagem dentro da árvore binária de forma recursiva.

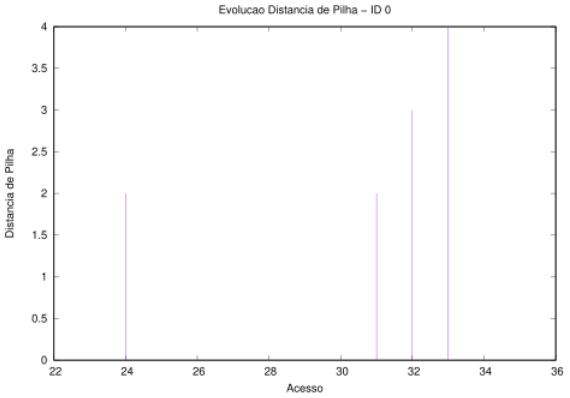
Nesse sentido, algo que pode ser feito para otimizar o desempenho é encontrar outras formas de inserir mensagens na caixa de entrada sem fazer uso de um loop para a leitura de cada palavra do conteúdo da mensagem, já que esse é o motivo de termos uma ordem de complexidade mais elevada.

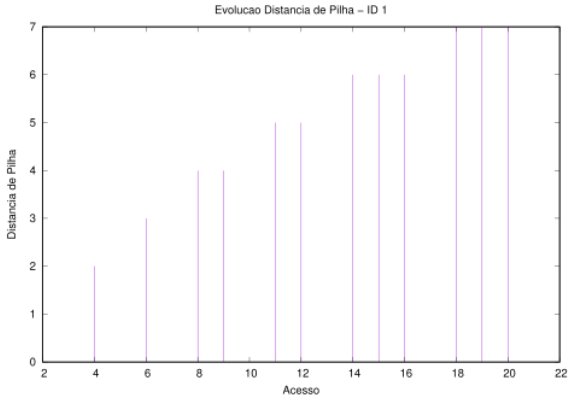
6.2. Análise de padrão de Acesso à Memória e Localidade de Referência

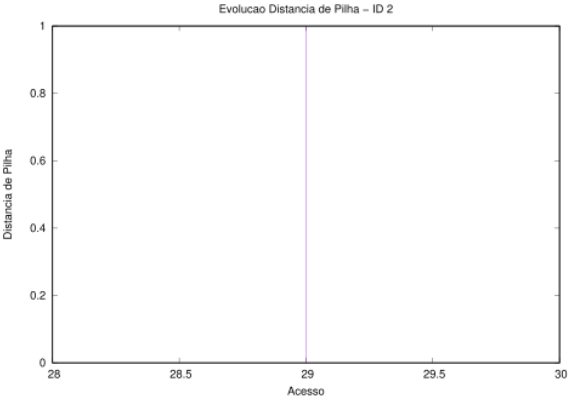


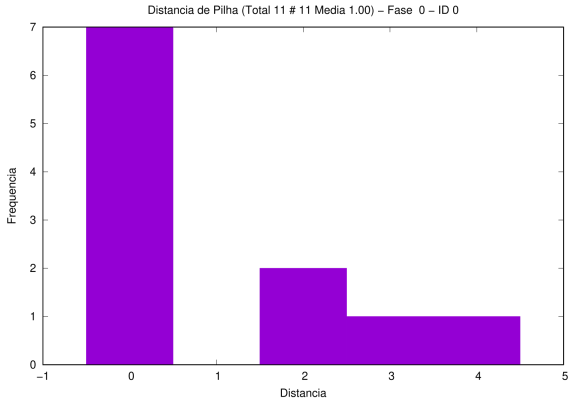


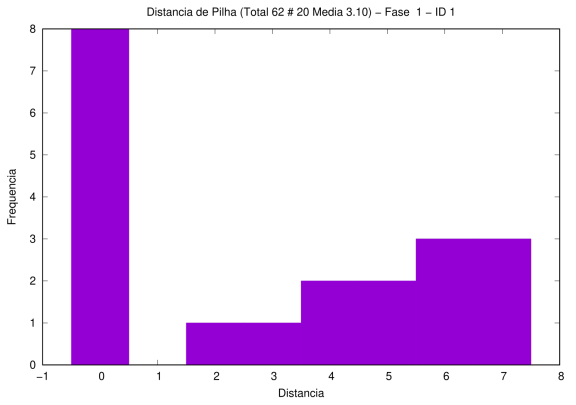


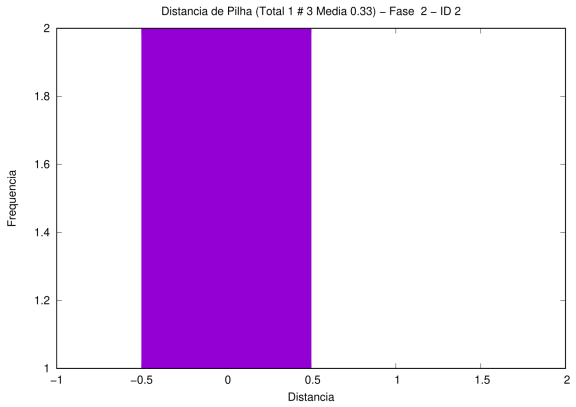


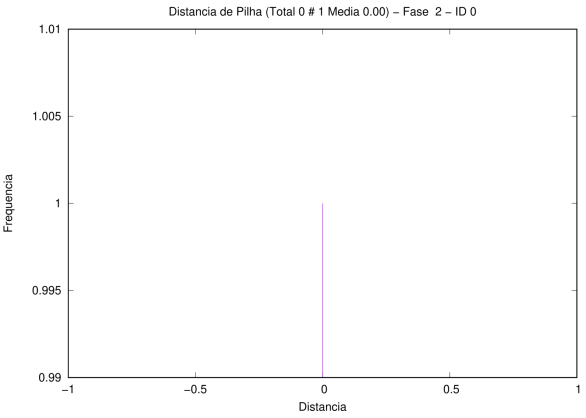












De início, segue abaixo o que cada ID e Fase significam nos gráficos mostrados acima:

Fases

- Fase 0: Leitura de e-mail
- Fase 1: Entrega de e-mail
- Fase 2: Remoção de e-mail

ID

- ID 0: Pesquisa na árvore binária
- ID 1: Inserção na árvore binária
- ID 2: Remoção da árvore binária

Nos gráficos de acesso durante operações de pesquisa, inserção e remoção, é possível perceber que não temos uma padronização de acesso completa. Isso ocorre que em todas essas operações é necessário navegar dentro dos nós da nossa árvore binária e, visto que o caminho que vamos percorrer vai depender da nossa chave, tem bastante chance do caminho mudar de acordo com o valor da chave inserida.

Além disso, é possível observar que a distância da pilha teve uma tendência de crescimento tanto na operação de pesquisa quanto na inserção, por essa questão do caminho a ser percorrido dentro da árvore mudar de acordo com a chave fornecida, ou seja, nem sempre iremos acessar um caminho parecido quando formos percorrer os nós.

Por fim, podemos ver que a distância de pilha se mantém em um tamanho menor para movimentações menores, visto que as chances de passarmos por um mesmo caminho é maior quando estamos lidando com chaves mais próximas da raiz da árvore binária e também quando a árvore binária ainda está com uma altura pequena.

7. Conclusões

Nesse trabalho prático, fizemos um programa que simula um servidor de e-mails, realizando operações importantes como consulta, inserção e remoção. Como estamos usando uma tabela hash para guardar as informações principais do servidor, é necessário realizar uma tratativa para prevenir eventuais bugs por conta da colisão de hash que é algo comum nessa estrutura de dados. Vale ressaltar também que usamos uma estrutura de dados de árvore binária para armazenar as mensagens.

Em seguida, através da execução desse programa, conseguimos gerar dados de desempenho computacional. Por fim, fomos capazes de tratar os dados gerados e analisá-los para entender o comportamento de forma mais aprofundada.

Diante disso, foi possível entender na prática como geralmente as informações desse contexto tão comum na nossa vida podem ser organizadas para garantir uma maior performance de pesquisa, além de imaginar possíveis decisões que podem ser tomadas para melhorar a qualidade desse serviço.

Por fim, é importante citar que uma das maiores dificuldades nesse trabalho surgiu no sentido de encontrar uma forma adequada para lidar com as possíveis colisões de hash que acontecem por conta do uso da estrutura de dados da tabela hash.

Apêndice

Instruções para compilação e execução

Antes de efetuar a compilação do programa principal, é necessário abrir a pasta **analisamem** e rodar o comando **make**.

Agora, para efetuar a compilação do programa, basta entrar na pasta **TP** e rodar o comando **make** em seu terminal.

Esse comando irá executar as tarefas presentes no arquivo **Makefile** e efetuar tanto a compilação quanto a execução do programa.