Documentação - Trabalho Prático 1 – Estruturas de Dados

Mateus Latrova Stephanin

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

mateusls@ufmg.br

1. Introdução

Esta documentação lida com o problema "Extração Z" explicado na especificação do trabalho. Como houve um mapeamento prévio do planeta para o qual os terráqueos enviariam sua nave espacial, a solução desse problema dependeu da automatização do recebimento e processamento das condições do planeta para o qual seria enviada essa nave, isto é, a quantidade de recursos, seres vivos e obstáculos ali presentes, além de suas localizações.

Além disso, ela tem como objetivo explicar e analisar clara e sucintamente como foram utilizados recursos, ideias e ferramentas para a solução desse problema. Para resolvê-lo, foi seguida uma estratégia de planejamento e implementação do código considerando cada entidade apresentada como uma classe(paradigma orientado a objetos), tendo, cada uma, seu papel e suas comunicações bem definidos dentro do programa.

2. Implementação

O primeiro aspecto de implementação analisado é a organização do projeto. Cada arquivo - tanto código-fonte(.cpp) como cabeçalho(.h) - se refere à descrição de uma única classe. Nos arquivos de cabeçalho está a declaração dos atributos e métodos, enquanto que nos arquivos de código-fonte está a definição de cada método(incluindo construtores e destrutores). A fim de evitar a dependência circular entre arquivos de cabeçalho, adotou-se a estratégia de declarações adiantadas de classes(em inglês, *forward declarations*), sendo que a inclusão desses arquivos foi feita sempre nos arquivos de código-fonte(a não ser no caso de alguns headers pré-implementados, como, por exemplo, <fstream>, <string> e <sstream>).

A seguir, está uma lista com todas as classes implementadas juntamente com uma breve explicação a respeito de sua estrutura/funcionamento:

- Node: similar à estrutura TipoCelula vista em aula, porém, neste caso, o TipoItem foi substituído por uma std::string, a fim de construir a próxima estrutura.
- LinkedQueue: uma fila encadeada de strings. Serviu para obedecer a política FIFO ao armazenar os comandos que deveriam ser executados pelos robôs além dos registros que cada um deles deveria guardar para apresentar o relatório.
- Pair: um par de números inteiros.
- Map: representa o mapa do arquivo de entrada. Possui um atributo tamanho(número de linhas e colunas), usado para se fazer a alocação de memória de uma matriz de

elementos do tipo char, na qual haverá os caracteres ‘B’, ‘R’, ‘H’, ‘O’ e ‘.’ passados no mapa de entrada.

- Robot: possui um ponteiro para a sua base, a fim de possibilitar sua comunicação com ela. Além disso, tem contadores para o número de recursos coletados e alienígenas eliminados, um par para armazenar sua posição, uma fila para guardar as ordens a ser executadas e, por fim, uma fila para guardar o histórico de ações.
- Base: representou a nave espacial que pousou no planeta em questão. Possui cinquenta robôs, um mapa para controlar suas posições, contadores para o número de recursos coletados e aliens eliminados pelos robôs e, também, uma fila para armazenar os comandos de entrada, os quais são distribuídos para os seus robôs.
- Order/Register: foi frealizado um comando “typedef”, definindo std::string como Order e, também, como Register, a fim de melhorar o entendimento do código. Order se refere aos comandos enviados aos robôs pela base e Register se refere ao registro de cada ação realizada pelos robôs, a qual é armazenada em uma de suas filas.

O programa principal foi responsável, basicamente, por criar *streams* nos quais seriam abertos os arquivos de entrada – passados como parâmetro pelo terminal – além de fechá-los ao final. Após abertos os arquivos, criou-se a base dos robôs utilizando seu construtor. Foi passado como parâmetro o stream que continha o arquivo do mapa, a fim de o mapa ser construído dentro dessa variável base.

Posteriormente, utilizou-se o método getRobots() a fim de numerar os 50 robôs já construídos a mandado do construtor da base e configurar sua posição inicial como (0,0) – a posição da base. Agora, chamando o método getOrders() e passando como parâmetro o *stream* que contém os comandos, a base armazena todos os comandos a serem enviados aos seus robôs.

Em seguida, por meio do método sendOrders(), a base envia os comandos obtidos aos robôs determinados. Os robôs receberão cada comando através do método receiveOrder(), o qual discernirá o tipo de comando recebido e chamará o método correspondente, fazendo o robô realizar a ação equivalente caso seja um comando direto, ou armazenar o comando na fila de comandos caso seja uma ordem de comando. Por fim, é chamado o método que faz a base imprimir o relatório final na saída padrão, os arquivos são fechados e o programa é terminado.

As configurações utilizadas para testar o programa foram as seguintes:

- Sistema Operacional: Linux(Ubuntu 20.04);
- Linguagem de programação: C++;
- Compilador: GNU C++ Compiler;
- Processador: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz - 8GB de RAM.

3. Instruções de compilação e execução

A implementação estará no diretório “projeto”, que conterá os diretórios “bin”, “include”, “obj” e “src”. Acesse o diretório “projeto” pelo terminal e, nele, execute o comando “make” para a compilação do código. Após a compilação, o arquivo executável “run.out” estará no diretório “bin”. Acesse-o pelo terminal e o execute passando como parâmetro, respectivamente, o *path* do arquivo que contém o mapa e o do arquivo que contém os comandos do jogo: “./run.out map.txt commands.txt”.

4. Análise de complexidade

Método Base(std::ifstream& mapFile):

Sendo m o número de linhas e n o número de colunas do mapa:

- **complexidade de tempo:** é conveniente considerar como operação de interesse a atribuição de valores aos elementos da matriz que representa o mapa - a qual ocorre dentro do construtor do mapa - pois as outras operações(há apenas atribuições) dentro deste construtor possuem custo constante(O(1)), isto é, são independentes do tamanho da entrada. Por conta disso, já que serão feitas $m \cdot n$ atribuições na matriz em questão(há dois loops aninhados para isso), a ordem de complexidade deste método em relação ao tempo é $O(m \cdot n)$.
- **complexidade de espaço:** de maneira semelhante, o custo de espaço de todas as atribuições neste método é constante, exceto a atribuição relacionada ao mapa. Nela, será chamado o construtor do mapa, no qual será alocado espaço para uma matriz de tamanho $m \cdot n$. Por conta disso, a ordem de complexidade deste método em relação ao espaço é $O(m \cdot n)$.

Método getRobots() :

- **complexidade de tempo:** visto que sempre é realizado o mesmo número de operações(100 atribuições em relação aos atributos dos robôs), a ordem de complexidade deste método em relação ao tempo é O(1).
- **complexidade de espaço:** visto, também, que a quantidade de memória é utilizada neste método é sempre a mesma(não há alocação para antigas ou novas estruturas, apenas atribuição de objetos anteriormente construídos), a ordem de complexidade deste método em relação ao espaço é O(1).

Método getOrders():

Sendo n o número de comandos no arquivo de entrada:

- **complexidade de tempo:** observando que tanto a função getline() como o método push() possuem custo constante em relação ao tempo, o que irá determinar o número de instruções realizadas neste método é a quantidade de vezes que o corpo do loop é executado, que

equivale ao número de vezes que a função getline() é chamada, que, por sua vez, é igual a n. Logo, a ordem de complexidade deste método em relação ao tempo é $n \cdot O(1) = O(n)$.

- **complexidade de espaço:** o que irá determinar esta complexidade também é o número de comandos n, pois a única alocação de memória variável é a relacionada à fila "this->_orders". A cada comando, um Node é alocado e encaixado ao final da fila. Dessa forma, a ordem de complexidade deste método em relação ao espaço é O(n).

Método receiveOrder():

Este método chama vários outros métodos de ação dos robôs, por isso, sua complexidade será determinada em função deles. Sendo n o número de comandos armazenados no robô e m o número de algarismos da maior coordenada da posição para a qual um robô deverá ser movido:

- **complexidade de tempo:** esta complexidade de tempo é determinada pelo máximo das complexidades dos métodos chamados por este método. Portanto, já que esse máximo vem do método execute(), que no pior caso é $O(n \cdot m)$, a ordem de complexidade de tempo deste método é $O(n \cdot m)$.
- **complexidade de espaço:** de maneira análoga, já que todos os métodos internos a este possuem custo de espaço constante, a ordem de complexidade de espaço deste método é O(1).

Método sendOrder() :

- **complexidade de tempo:** neste método, há uma sequência inicial de custo constante(número de operações é sempre o mesmo). Porém, na última linha, chama-se o método receiveOrder(), o qual tem complexidade de tempo $O(n \cdot m)$. Dessa forma, a ordem de complexidade de tempo deste método é $O(1) + O(n \cdot m) = O(n \cdot m)$.
- **complexidade de espaço:** já que tanto a sequência de instruções quanto os métodos internos ao métodos possuem complexidade de espaço constante, sua complexidade também é constante(O(1)).

Método sendOrders() :

Sendo p o número de ordens armazenadas na fila da base e já que temos um loop de p iterações chamando a função sendOrder():

- **complexidade de tempo:** a ordem de complexidade deste método em relação ao tempo é $p \cdot$ (complexidade de sendOrder) $= p \cdot O(n \cdot m) = O(n \cdot m \cdot p)$.
- **complexidade de espaço:** a ordem de complexidade deste método em relação ao espaço é $p \cdot$ (complexidade de sendOrder) $= p \cdot O(1) = O(p)$.

Método printFinalReport():

- **complexidade de tempo:** o número de instruções executadas aqui é o mesmo em qualquer caso(há apenas uma instrução de impressão), fazendo com que a ordem de complexidade deste método em relação ao tempo seja $\Theta(1)$.
- **complexidade de espaço:** não há alocação de memória para estruturas estáticas ou dinâmicas, o que faz com que a ordem de complexidade deste método em relação ao espaço seja $\Theta(1)$.

5. Conclusão

Este trabalho lidou com o problema "Extração Z" e resolveu-o com uma abordagem clara, organizada e com a maior simplicidade alcançada pelo autor. Procurou-se fazê-lo de forma que qualquer programador pudesse entender a solução apenas analisando o código.

Durante a implementação da solução, houve diversas dificuldades, as quais se tornaram grandes ganhos em conhecimento para a realização de futuros projetos. Foi possível praticar muitos conceitos antes vistos, porém que não haviam ficado claros sem "colocar a mão na massa".

Dentre os aprendizados obtidos mediante as dificuldades, podem ser listados:

- Estrutura e uso de streams(filestream e stringstream principalmente);
- Manipulação de std::strings e c-strings;
- Orientação a objeto no geral, mas principalmente a parte de polimorfismo utilizando templates, embora foi decidido ao final do projeto que não seriam mais utilizados devido a ocorrência de muitos erros;
- Dependência circular(como resolver utilizando *forward declarations*);
- Funcionamento de construtores e destrutores de objetos alocados estática e dinamicamente(ajudou a resolver erros de segmentação);
- Prática de debugação;
- Conceitos sobre makefile;
- Organização de projetos robustos de programação.

6. Bibliografia

- http://web.eecs.utk.edu/~bvanderz/teaching/cs365Sp12/notes/templates.html
- https://www.youtube.com/watch?v=0ebzPwixrJA
- http://www.cplusplus.com/
- https://stackoverflow.com/
- https://www.geeksforgeeks.org/