ALGORITMOS E ESTRUTURAS DE DADOS 2

TRABALHO PRÁTICO 1

Listas

Autor: Everaldo Chaves de Oliveira Junior

Prof.: Jefferson de Oliveira Balduino

Serra - ES

Abril/2021

Sumário

1. OBJETIVO 3
2. FUNCIONAMENTO GERAL 3
3. IMPLEMENTAÇÃO 5

    3.1 LINKEDLIST 5
    3.2 NODE 7
    3.3 LINKEDLISTITERATOR 8
    3.4 COMMANDTYPES 9
    3.5 BLOCKCOMMAND 9
    3.6 COMMANDPARSER 10
    3.7 BLOCK 11
    3.8 BLOCKINFO 11
    3.9 TBLOCOS 13
4. CÓDIGO-FONTE 16

    4.1 LINKEDLIST 16
    4.2 NODE 18
    4.3 LINKEDLISTITERATOR 18
    4.4 COMMANDTYPES 19
    4.5 BLOCKCOMMAND 19
    4.6 COMMANDPARSER 20
    4.7 BLOCK 22
    4.8 BLOCKINFO 23
    4.9 TBLOCOS 23
    4.10 MAIN 30

1. OBJETIVO

Implementar um mundo de blocos[1] simples sob certas regras e restrições. Respondendo a um conjunto limitado de comandos dados pelo usuário, simulando um braço robótico.

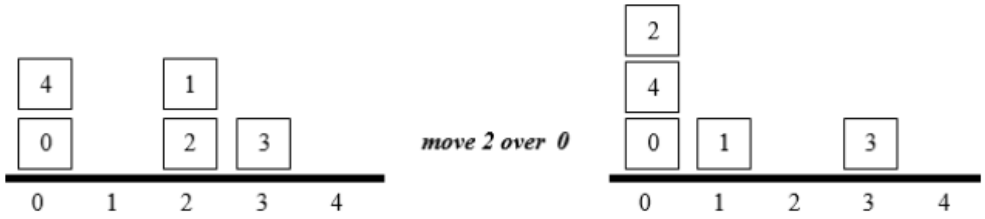
2. FUNCIONAMENTO GERAL

O mundo de blocos possui um conjunto limitado de comandos (onde comandos com a = b ou onde a e b estejam no mesmo monte, devem ser ignorados), sendo eles:

1. **move a onto b**: move o bloco a para cima do bloco b retornando eventuais blocos que já estiverem sobre a ou b para as suas posições originais.

(move 2 onto 0)

2. **move a over b:** Coloca o bloco a no topo do monte onde está o bloco b retornando eventuais blocos que já estiverem sobre a às suas posições originais.

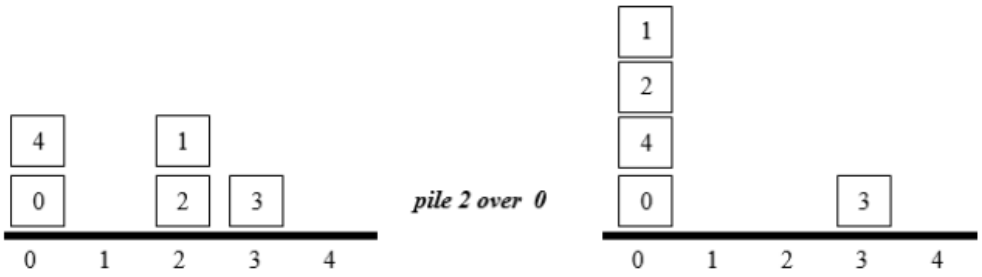


(move 2 over 0)

3. **pile a onto b:** coloca o bloco a juntamente com todos os blocos que estiverem sobre ele em cima do bloco b, retornando eventuais blocos que já estiverem sobre b as suas posições originais.

(pile 2 onto 0)

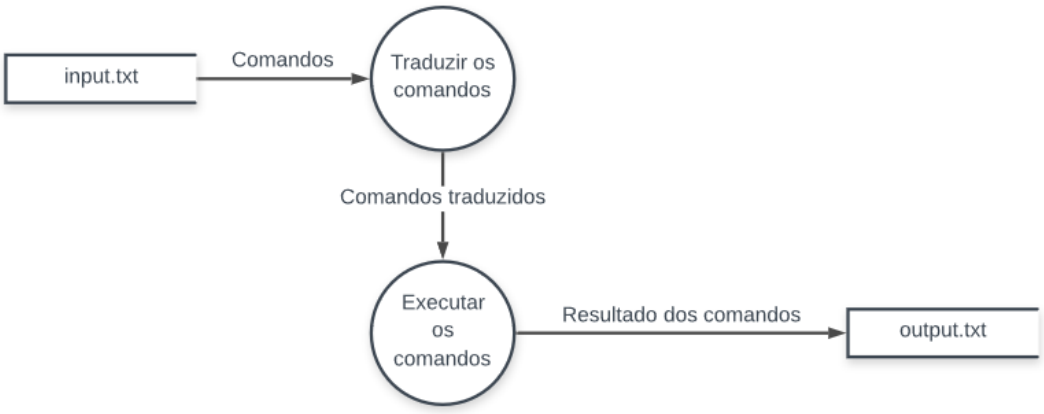
4. **pile a over b:** coloca o bloco a juntamente com todos os blocos que estiverem sobre ele sobre o monte que contém o bloco b.



(pile 2 over 0)

5. **quit:** termina a execução.

O funcionamento do programa é baseado em duas classes principais, TBlocos e CommandParser, sendo a primeira responsável por gerenciar e executar todas as manipulações dos blocos e a segunda encarregada de traduzir os comandos dados pelo usuário para o padrão dos blocos. Abaixo o digrama para representar o fluxo dos dados.



(Diagrama de fluxo de dados)

3. IMPLEMENTAÇÃO

3.1 LinkedList

Cria uma lista encadeada genérica. Implementa Iterable para seus nós poderem ser alcançados, por uma estrutura de repetição.

Estrutura:

```
public class LinkedList<T> implements Iterable<T>
```

Propriedades

FirstNode

Representa o primeiro nó da lista.

Estrutura:

```
private Node<T> firstNode
```

LastNode

Representa o último nó da lista.

Estrutura:

```
private Node<T> lastNode
```

Length

Representa o número total de elementos na lista.

Estrutura:

```
private int length
```

Métodos

GetLength

Retorna um inteiro com o total de elementos na lista.

Estrutura:

```
public int GetLength()
```

GetFirstNode

Retorna o primeiro nó da lista, um node do tipo T.

Estrutura:

```
public Node<T> GetFirstNode()
```

Add

Adiciona um item do tipo T no final da lista.

Estrutura:

```
public void Add(T item)
```

Remove

Remove o item do tipo T da lista.

Estrutura:

```
public void Remove(T item)
```

Reverse

Inverte a ordem dos nós da lista.

Estrutura:

```
public void Reverse()
```

Iterator

Retorna o elemento responsável por realizar a iteração da lista.

Estrutura:

```
@Override
public Iterator<T> iterator()
```

3.2 Node

Representa um nó genérico da lista encadeada.

Estrutura:

```
public class Node<T>
```

Propriedades

Item

Representa um valor genérico armazenado no nó.

Estrutura:

```
private T item
```

NextNode

Representa o próximo nó.

Estrutura:

```
private Node nextNode
```

Métodos

GetItem

Retorna o item armazenado no nó.

Estrutura:

```
public T GetItem()
```

GetNextNode

Retorna o próximo nó.

Estrutura:

```
public Node<T> GetNextNode()
```

UpdateNextNode

Aponta o nó atual para outro nó.

Estrutura:

```
public void UpdateNextNode(Node next)
```

3.3 LinkedListIterator

Classe responsável por realizar a iteração sobre a lista encadeada genérica.

Estrutura:

```
public class LinkedListIterator<T> implements Iterator<T>
```

Propriedades

CurrentNode

Representa o nó para o qual a estrutura de repetição está apontando no momento.

Estrutura:

```
private Node<T> currentNode
```

Métodos

HasNext

Retorna um boolean caso exista o próximo elemento para iterar.

Estrutura:

```
@Override
 public boolean hasNext()
```

Next

Retorna o item armazenado no nó e atualiza o próximo nó.

Estrutura:

```
@Override
  public T next()
```

3.4 CommandTypes

Enum responsável por representar os tipos de comandos.

3.5 BlockCommand

Classe responsável por representar um comando já convertido da linguagem do usuário para o mundo dos blocos.

Estrutura:

```
public class BlockCommand
```

Propriedades

CommandType

Representa o tipo do comando.

Estrutura:

```
private CommandTypes commandType
```

Arguments

Um array com todos os argumentos desse comando.

Estrutura:

```
private int[] arguments
```

Métodos

GetType

Retorna o tipo desse comando.

Estrutura:

```
public CommandTypes GetType()
```

GetArguments

Retorna os argumentos desse comando.

Estrutura:

```
public int[] GetArguments()
```

3.6 CommandParser

Classe responsável por converter os comandos da linguagem do usuário para o mundo dos blocos.

Estrutura:

```
public class CommandParser
```

Propriedades

FileScanner

Scanner com todas as linhas do arquivo de entrada, contendo os comandos do usuário.

Estrutura:

```
private Scanner fileScanner
```

Métodos

Parse

Converte todos os comandos e retorna uma lista encadeada com todos os comandos convertidos.

Estrutura:

```
public LinkedList<BlockCommand> Parse()
```

3.7 Block

Representação do bloco.

Estrutura:

```
public class Block
```

Propriedades

Id

Corresponde ao id do bloco.

Estrutura:

```
private int id
```

Métodos

GetId

Retorna o id do bloco.

Estrutura:

```
public int GetId()
```

3.8 BlockInfo

Classe responsável por armazenar um conjunto de atributos dos blocos.

Estrutura:

```
public class BlockInfo
```

Propriedades

Block

Bloco a qual pertence as propriedades.

Estrutura:

```
private Block block
```

CurrentPosition

Representa em qual pilha o bloco está.

Estrutura:

```
private int currentPosition
```

Stacked

Uma lista encadeada com todos os blocos empilhados no bloco atual.

Estrutura:

```
private LinkedList<Block> stacked
```

Métodos

GetBlock

Retorna o bloco.

Estrutura:

```
public Block GetBlock()
```

GetCurrentPosition

Retorna em qual pilha o bloco está.

Estrutura:

```
public int GetCurrentPosition()
```

GetStackedBlocks

Retorna uma lista encadeada com todos os blocos empilhados no bloco atual.

Estrutura:

```
public LinkedList<Block> GetStackedBlocks()
```

3.9 TBlocos

Classe que gerencia todo o mundo dos blocos.

Estrutura:

```
public class TBlocos
```

Propriedades

Blocks

Um array de lista encadeadas, onde cara posição do array representa uma pilha de blocos.

Estrutura:

```
private LinkedList<Block>[] blocks
```

Métodos

PrintBlocks

Função para imprimir na tela o mundo de blocos.

Estrutura:

```
public void PrintBlocks()
```

SaveOutput

Salva um arquivo texto com o resultado da manipulação dos blocos em um determinado caminho.

Estrutura:

```
public void SaveOutput(String path)
```

ExecuteCommands

Recebe uma lista de comandos e executa todos os movimentos.

Estrutura:

```
public void ExecuteCommands(LinkedList<BlockCommand>
commands)
```

CreateBlocks

Cria as pilhas com os blocos baseando-se na quantidade informada pelo usuário.

Estrutura:

```
private void CreateBlocks(int count)
```

MoveOnto

Move o bloco a para cima do bloco b retornando eventuais blocos que já estiverem sobre a ou b para as suas posições originais.

Estrutura:

```
private void MoveOnto(int a, int b)
```

MoveOver

Coloca o bloco a no topo do monte onde está o bloco b retornando eventuais blocos que já estiverem sobre a às suas posições originais.

Estrutura:

```
private void MoveOver(int a, int b)
```

PileOnto

Coloca o bloco a juntamente com todos os blocos que estiverem sobre ele em cima do bloco b, retornando eventuais blocos que já estiverem sobre b as suas posições originais.

Estrutura:

```
private void PileOnto(int a, int b)
```

PileOver

Coloca o bloco a juntamente com todos os blocos que estiverem sobre ele sobre o monte que contém o bloco b.

Estrutura:

```
private void PileOver(int a, int b)
```

CanExecuteCommand

Comandos onde a = b ou a e b estejam no mesmo monte retornam false.

Estrutura:

```
private boolean CanExecuteCommand(int a, int b)
```

ReturnToDefaultPosition

Recebe uma lista de blocos e retorna eles para sua posição inicial.

Estrutura:

```
private void
ReturnToDefaultPosition(LinkedList<Block> returnList)
```

GetBlockInfo

Retorna os atributos de um bloco baseado no Id do bloco.

Estrutura:

```
private BlockInfo GetBlockInfo(int id)
```

BlocksToString

Converte o resultado da manipulação para uma string.

Estrutura:

```
private String BlocksToString()
```

4. CÓDIGO-FONTE

4.1 LinkedList

```
package com.everaldojunior.Utils.List;

import com.everaldojunior.Blocks.Block;
import java.util.Iterator;

public class LinkedList<T> implements Iterable<T>
{
    private Node<T> firstNode;
    private Node<T> lastNode;
    private int length;

    public LinkedList()
    {
        this.firstNode = null;
        this.lastNode = null;
        this.length = 0;
    }

    public int GetLength()
    {
        return this.length;
    }

    public Node<T> GetFirstNode()
    {
        return this.firstNode;
    }

    public void Add(T item)
    {
        var newNode = new Node<>(item, null);

        //Checa se a lista está vazia, caso esteja preenche o
primeiro node
        if (this.firstNode == null)
        {
            this.firstNode = newNode;
        }
        else
        {
            this.lastNode.UpdateNextNode(newNode);
        }

        //Atualiza o final da lista e incrementa o comprimento
        this.lastNode = newNode;
        this.length++;
    }
```

```
    public void Remove(T item)
    {
        if(item == null)
            return;

        var currentNode = this.firstNode;
        Node<T> lastNode = null;

        //Percorre todos os elementos da lista e deleta o
primeiro que combinar
        while (currentNode != null)
        {
            //TODO Checar se a verificação direta pega todos os
     casos ou trocar por Equals
            if(currentNode.GetItem() == item)
            {
                var nextNode = currentNode.GetNextNode();
                //Checa se existe um node anterior e se tiver
ele já atualiza pro proximo node
                if(lastNode != null)
                    lastNode.UpdateNextNode(nextNode);

                //Verificacoes do head da lista
                if(currentNode == this.firstNode)
                    this.firstNode = nextNode;
                if(currentNode == this.lastNode)
                    this.lastNode = lastNode;

                length--;
                break;
            }

            lastNode = currentNode;
            currentNode = currentNode.GetNextNode();
        }
    }

    public void Reverse()
    {
        var currentNode = this.firstNode;
        this.firstNode = this.lastNode;
        this.lastNode = currentNode;

        Node<T> nextNode = null;
        Node<T> lastNode = null;
        while (currentNode != null)
        {
            nextNode = currentNode.GetNextNode();
            currentNode.UpdateNextNode(lastNode);
            lastNode = currentNode;
```

```
            currentNode = nextNode;
        }
    }

    @Override
    public Iterator<T> iterator()
    {
        //Iterador pra conseguir usar o for e não ter que fazer
o while sempre
        return new LinkedListIterator<>(this);
    }
}
```

4.2 Node

```
package com.everaldojunior.Utils.List;

public class Node<T>
{
    private T item;
    private Node nextNode;

    public Node(T item, Node next)
    {
        this.item = item;
        this.nextNode = next;
    }

    public T GetItem()
    {
        return this.item;
    }

    public Node<T> GetNextNode()
    {
        return this.nextNode;
    }

    public void UpdateNextNode(Node next)
    {
        this.nextNode = next;
    }
}
```

4.3 LinkedListIterator

```
package com.everaldojunior.Utils.List;

import java.util.Iterator;
```

```
public class LinkedListIterator<T> implements Iterator<T>
{
    private Node<T> currentNode;

    public LinkedListIterator(LinkedList<T> list)
    {
        this.currentNode = list.GetFirstNode();
    }

    @Override
    public boolean hasNext()
    {
        return this.currentNode != null;
    }

    @Override
    public T next()
    {
        var item = this.currentNode.GetItem();
        this.currentNode = this.currentNode.GetNextNode();
        return item;
    }
}
```

4.4 CommandTypes

```
package com.everaldojunior.Utils.Command;

public enum CommandTypes
{
    CREATE_BLOCKS,
    MOVE_ONTO,
    MOVE_OVER,
    PILE_ONTO,
    PILE_OVER,
    QUIT
}
```

4.5 BlockCommand

```
package com.everaldojunior.Utils.Command;

public class BlockCommand
{
    private CommandTypes commandType;
    private int[] arguments;

    public BlockCommand(CommandTypes type, int[] args)
```

```
    {
        this.commandType = type;
        this.arguments = args;
    }

    public CommandTypes GetType()
    {
        return this.commandType;
    }

    public int[] GetArguments()
    {
        return this.arguments;
    }
}
```

4.6 CommandParser

```
package com.everaldojunior.Utils.Command;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import com.everaldojunior.Utils.List.LinkedList;

public class CommandParser
{
    private Scanner fileScanner;

    public CommandParser(String path)
    {
        //Carrega o arquivo e cria o scanner
        var file = new File(path);

        try
        {
            this.fileScanner = new Scanner(file);
        }
        catch (FileNotFoundException e)
        {
            System.out.println("ERRO: Não foi encontrado o
arquivo no caminho: " + path);
        }
    }

    public LinkedList<BlockCommand> Parse()
    {
        var commands = new LinkedList<BlockCommand>();

        if(this.fileScanner == null)
```

```
        {
            System.out.println("ERRO: Arquivo inválido");
            return commands;
        }

        //Faz o parser dos comandos
        while (this.fileScanner.hasNextLine())
        {
            var line = this.fileScanner.nextLine();

            //Separa os comandos pelos espaços entre eles
            var tokens = line.split(" ");

            if(tokens.length == 1)
            {
                var type = tokens[0];
                BlockCommand command;

                if(type.equals("quit"))
                    command = new
BlockCommand(CommandTypes.QUIT, null);
                else
                    command = new
BlockCommand(CommandTypes.CREATE_BLOCKS, new
int[]{Integer.parseInt(type)});

                commands.Add(command);
            }
            else if(tokens.length == 4)
            {
                var type = tokens[0];
                var subType = tokens[2];
                var args = new
int[]{Integer.parseInt(tokens[1]),
Integer.parseInt(tokens[3])};

                if(subType.equals("onto"))
                {
                    BlockCommand command;

                    if(type.equals("move"))
                        command = new
BlockCommand(CommandTypes.MOVE_ONTO, args);
                    else
                        command = new
BlockCommand(CommandTypes.PILE_ONTO, args);

                    commands.Add(command);
                }
                else
                {
```

```
                    BlockCommand command;

                    if(type.equals("move"))
                        command = new
BlockCommand(CommandTypes.MOVE_OVER, args);
                    else
                        command = new
BlockCommand(CommandTypes.PILE_OVER, args);

                    commands.Add(command);
                }
            }
        }

        return commands;
    }
}
```

4.7 Block

```
package com.everaldojunior.Blocks;

public class Block
{
    private int id;

    public Block(int id)
    {
        this.id = id;
    }

    public int GetId()
    {
        return this.id;
    }
}
```

4.8 BlockInfo

```
package com.everaldojunior.Blocks;

import com.everaldojunior.Utils.List.LinkedList;

public class BlockInfo
{
    private Block block;
    private int currentPosition;
    private LinkedList<Block> stacked;
```

```
    public BlockInfo(Block block, int currentPosition,
LinkedList<Block> stacked)
    {
        this.block = block;
        this.currentPosition = currentPosition;
        this.stacked = stacked;
    }

    public Block GetBlock()
    {
        return this.block;
    }

    public int GetCurrentPosition()
    {
        return this.currentPosition;
    }

    public LinkedList<Block> GetStackedBlocks()
    {
        return this.stacked;
    }
}
```

4.9 TBlocos

```
package com.everaldojunior.Blocks;

import com.everaldojunior.Utils.Command.BlockCommand;
import com.everaldojunior.Utils.List.LinkedList;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class TBlocos
{
    private LinkedList<Block>[] blocks;

    //Função para printar todos os blocos formatados na tela
    public void PrintBlocks()
    {
        System.out.print(BlocksToString());
    }

    //Função para salvar o output em um arquivo texto
    public void SaveOutput(String path)
    {
```

```
        try
        {
            var outputFile = new File(path);

            var writer = new FileWriter(outputFile);
            writer.write(BlocksToString());
            writer.close();
        }
        catch (IOException e)
        {
            System.out.println("Erro: Ocorreu um erro ao salvar
o output");
        }
    }

    //Função para ler os comandos e executar os movimentos
    public void ExecuteCommands(LinkedList<BlockCommand>
commands)
    {
        for (var command : commands)
        {
            var args = command.GetArguments();
            switch (command.GetType())
            {
                case CREATE_BLOCKS:
                    CreateBlocks(args[0]);
                    break;
                case MOVE_ONTO:
                    MoveOnto(args[0], args[1]);
                    break;
                case MOVE_OVER:
                    MoveOver(args[0], args[1]);
                    break;
                case PILE_ONTO:
                    PileOnto(args[0], args[1]);
                    break;
                case PILE_OVER:
                    PileOver(args[0], args[1]);
                    break;
                case QUIT:
                    return;
            }
        }
    }

    //Comandos
    //Cria o array e preenche as listas
    private void CreateBlocks(int count)
    {
        this.blocks = new LinkedList[count];
```

```
        //Preenchendo com os blocos default
        for (var i = 0; i < count; i++)
        {
            var block = new Block(i);
            var list = new LinkedList<Block>();
            list.Add(block);

            this.blocks[i] = list;
        }
    }

    //move o bloco a para cima do bloco b retornando eventuais
blocos que já
    //estiverem sobre a ou b para as suas posições originais.
    private void MoveOnto(int a, int b)
    {
        //Verifica se pode executar o comando, caso não apenas
ignora
        if(!CanExecuteCommand(a, b))
            return;

        //Pega as informações do bloco A
        var blockAInfo = GetBlockInfo(a);
        Block blockA = blockAInfo.GetBlock();
        int blockAPosition = blockAInfo.GetCurrentPosition();
        var stackedOnA = blockAInfo.GetStackedBlocks();

        //Pega as informações do bloco B
        var blockBInfo = GetBlockInfo(b);
        int blockBPosition = blockBInfo.GetCurrentPosition();
        var stackedOnB = blockBInfo.GetStackedBlocks();

        //Retorna os blocos empilhados em cima do A para a
posição inicial
        ReturnToDefaultPosition(stackedOnA);

        //Retorna os blocos empilhados em cima do B para a
posição inicial
        ReturnToDefaultPosition(stackedOnB);

        //Desempilha o bloco A da posição que ele tava e
empilha em cima de B
        this.blocks[blockAPosition].Remove(blockA);
        this.blocks[blockBPosition].Add(blockA);
    }

    //Coloca o bloco a no topo do monte onde está o bloco b
retornando eventuais
    //blocos que já estiverem sobre a às suas posições
originais.
    private void MoveOver(int a, int b)
```

```
    {
        //Verifica se pode executar o comando, caso não apenas
ignora
        if(!CanExecuteCommand(a, b))
            return;

        //Pega as informações do bloco A
        var blockAInfo = GetBlockInfo(a);
        Block blockA = blockAInfo.GetBlock();
        int blockAPosition = blockAInfo.GetCurrentPosition();
        var stackedOnA = blockAInfo.GetStackedBlocks();

        //Pega as informações do bloco B
        var blockBInfo = GetBlockInfo(b);
        int blockBPosition = blockBInfo.GetCurrentPosition();

        //Retorna os blocos empilhados em cima do A para a
posição inicial
        ReturnToDefaultPosition(stackedOnA);

        //Desempilha o bloco A da posição que ele tava e
empilha em cima de B
        this.blocks[blockAPosition].Remove(blockA);
        this.blocks[blockBPosition].Add(blockA);
    }

    //coloca o bloco a juntamente com todos os blocos que
estiverem sobre ele
    //em cima do bloco b, retornando eventuais blocos que já
estiverem sobre b as suas posições
    //originais.
    private void PileOnto(int a, int b)
    {
        //Verifica se pode executar o comando, caso não apenas
ignora
        if(!CanExecuteCommand(a, b))
            return;
        //Pega as informações do bloco A
        var blockAInfo = GetBlockInfo(a);
        Block blockA = blockAInfo.GetBlock();
        int blockAPosition = blockAInfo.GetCurrentPosition();
        var stackedOnA = blockAInfo.GetStackedBlocks();

        //Pega as informações do bloco B
        var blockBInfo = GetBlockInfo(b);
        int blockBPosition = blockBInfo.GetCurrentPosition();
        var stackedOnB = blockBInfo.GetStackedBlocks();

        //Retorna os blocos empilhados em cima do B para a
posição inicial
        ReturnToDefaultPosition(stackedOnB);
```

```
        //Desempilha o bloco A da posição que ele tava e
empilha em cima de B
        this.blocks[blockAPosition].Remove(blockA);
        this.blocks[blockBPosition].Add(blockA);

        //Desempilha todos os blocos que esta em A e empilha em
B
        for (var block : stackedOnA)
        {
            this.blocks[blockAPosition].Remove(block);
            this.blocks[blockBPosition].Add(block);
        }
    }

    //coloca o bloco a juntamente com todos os blocos que
estiverem sobre ele
    //sobre o monte que contém o bloco b.
    private void PileOver(int a, int b)
    {
        //Verifica se pode executar o comando, caso não apenas
ignora
        if(!CanExecuteCommand(a, b))
            return;
        //Pega as informações do bloco A
        var blockAInfo = GetBlockInfo(a);
        Block blockA = blockAInfo.GetBlock();
        int blockAPosition = blockAInfo.GetCurrentPosition();
        var stackedOnA = blockAInfo.GetStackedBlocks();

        //Pega as informações do bloco B
        var blockBInfo = GetBlockInfo(b);
        int blockBPosition = blockBInfo.GetCurrentPosition();

        //Desempilha o bloco A da posição que ele tava e
empilha em cima de B
        this.blocks[blockAPosition].Remove(blockA);
        this.blocks[blockBPosition].Add(blockA);

        //Desempilha todos os blocos que esta em A e empilha em
B
        for (var block : stackedOnA)
        {
            this.blocks[blockAPosition].Remove(block);
            this.blocks[blockBPosition].Add(block);
        }
    }

    //Checa as seguintes condições:
    //Comandos onde a = b ou onde a e b estejam no mesmo monte,
devem ser ignorados.
```

```
    private boolean CanExecuteCommand(int a, int b)
    {
        if(a == b)
            return false;

        for (var i = 0; i < blocks.length; i++)
        {
            var foundA = false;
            var foundB = false;

            for (var block : this.blocks[i])
            {
                if(block.GetId() == a)
                    foundA = true;
                else if(block.GetId() == b)
                    foundB = true;

                if(foundA && foundB)
                    return false;
            }
        }

        return true;
    }

    private void ReturnToDefaultPosition(LinkedList<Block>
returnList)
    {
        //Reverte a lista para remover do ultimo até chegar no
bloco A
        returnList.Reverse();

        for (var blockToReturn : returnList)
            //Procura pelo bloco, remove e adiciona na posição
inicial
            for (var i = 0; i < blocks.length; i++)
            {
                //Procura pelo bloco e remove
                for (var block : this.blocks[i])
                    if(block.GetId() == blockToReturn.GetId())
                        blocks[i].Remove(block);

                //Adiciona o bloco na posição inicial
                if(i == blockToReturn.GetId())

blocks[blockToReturn.GetId()].Add(blockToReturn);
            }
    }

    //Pega o bloco, sua posição e quem está empilhado em cima.
Baseado no ID do bloco
```

```
    private BlockInfo GetBlockInfo(int id)
    {
        var stacked = new LinkedList<Block>();
        Block block = null;
        int blockPosition = 0;

        //Pega quais blocos estão em cima de A
        for (var i = 0; i < blocks.length; i++)
        {
            var found = false;
            for (var b : this.blocks[i])
            {
                //Armazena os blocos que estão em cima de A ou
B
                if(found)
                    stacked.Add(b);

                if(b.GetId() == id)
                {
                    block = b;
                    blockPosition = i;
                    found = true;
                }
            }
        }

        return new BlockInfo(block, blockPosition, stacked);
    }

    //Converte a estrutura de blocos para uma string
    private String BlocksToString()
    {
        var string = "";

        if(blocks == null)
            return string;

        for (var i = 0; i < blocks.length; i++)
        {
            string += i + ": ";
            for (var block : blocks[i])
                string += block.GetId() + " ";
            string += "\n";
        }

        return string;
    }
}
```

4.10 Main

```
package com.everaldojunior;

import com.everaldojunior.Blocks.TBlocos;
import com.everaldojunior.Utils.Command.CommandParser;

public class Main
{
    public static void main(String[] args)
    {
        var blocks = new TBlocos();
        var parser = new CommandParser("src/input.txt");
        var commands = parser.Parse();

        blocks.ExecuteCommands(commands);
        blocks.PrintBlocks();
        blocks.SaveOutput("src/output.txt");
    }
}
```