Algoritmos e Estruturas de Dados I
Trabalho 2 - A Calculadora

Daniele Ramos de Souza

16 de setembro de 2015

Resumo

Este trabalho é a segunda das avaliações da disciplina de Algoritmos e Estruturas de Dados I do curso de Engenharia de Software da Faculdade de Informática (FACIN) da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) e tem como objetivo a construção de uma calculadora capaz de ler arquivos de entrada com números inteiros e comandos de operação e executar casos de teste, utilizando uma pilha.

1 Introdução

O desafio deste trabalho é implementar um programa com a funcionalidade de calculadora, capaz de ler arquivos de entrada com números inteiros e comandos de operação. Os números são armazenados em uma pilha, por meio de um método chamado `inteiro`, e são operados dois a dois ou individualmente, conforme o comando dado. Nas operações, os números envolvidos são removidos da pilha e os resultados obtidos são inseridos com ponto flutuante, para serem utilizados em outras operações.

Os métodos que realizam as operações de soma e de multiplicação utilizam dois números, retirando o elemento do topo e o seguinte, operando eles e devolvendo o resultado para a pilha.

As operações de subtração e divisão também utilizam dois números, exigindo neste momento um cuidado especial com a leitura dos arquivos de entrada, o armazenamento dos elementos na pilha e a retirada para operação. Um arquivo que apresenta `2 3 -` como dados de entrada é lido. O programa armazena primeiro o número 2 e depois o número 3 na pilha, que fica como `[3 2]`, sendo o número 3 o elemento no topo da pilha e o número 2 o seguinte. A operação de subtração retira o número 3 e depois o número 2 da pilha e calcula `3 - 2 = 1`, retornando para a pilha o número 1 como resultado. E não o contrário. Veja na figura 1:

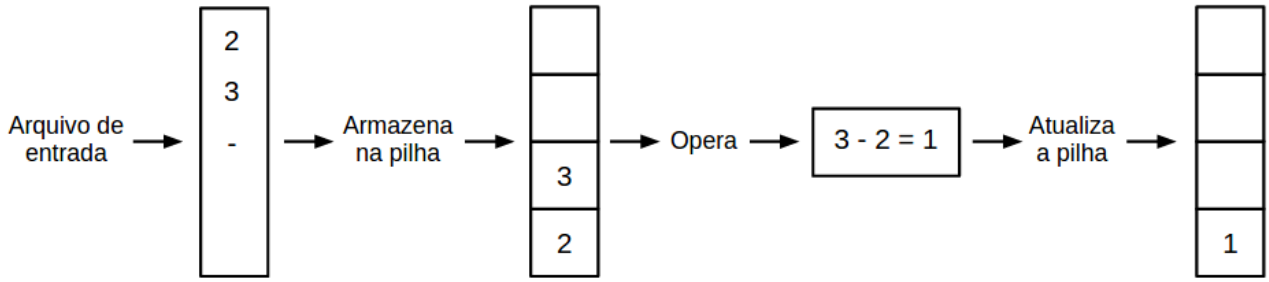

Figura 1: Processo de leitura de arquivo na operação de subtração

O programa também realiza o cálculo do seno e do cosseno de um número, e do arco-tangente, utilizando dois valores, com o mesmo cuidado necessário nos métodos de subtração e divisão. Além das operações matemáticas, a calculadora deve ser capaz de descartar o último resultado (método `pop`), repetir o último resultado (método `dup`) e trocar de ordem os dois últimos resultados (método `swap`).

Com o programa implementado, é hora de executar os oito casos de teste que nos foram dados junto com o enunciado do trabalho. O resultado deles está no tópico 4 e a descrição da implementação da calculadora e das suas operações no tópico 3. Mas antes vamos fazer alguns apontamentos sobre pilhas (tópico 2) e sobre RPN (tópico 3), a fim de construir um embasamento teórico mais consistente para dar continuidade ao trabalho.

2 Sobre pilhas

Uma pilha é uma estrutura de dados muito simples, que funciona basicamente acessando o primeiro elemento, ou seja, o elemento que está no topo dela. É através dele que ela manipula elementos, empilhando e desempilhando eles. Desta forma, é possível dizer que a ordem em que ela opera estes elementos segue o conceito de LIFO (Last In First Out), que significa, em português, que o último elemento a entrar é o primeiro a sair, como mostra a figura 2:

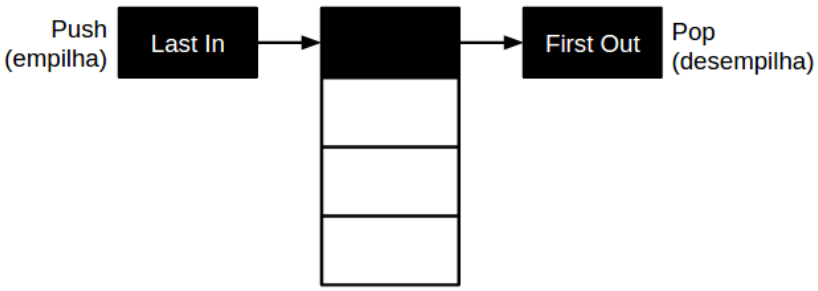

Figura 2: Funcionamento de uma pilha

Existem três operações principais quando estamos trabalhando com pilhas: a de inserir elementos, que podemos chamar `push`, a de remover elementos, `pop`, e a de verificar se a estrutura está vazia, `empty` ou `isEmpty`. As demais operações podem ser feitas a partir delas.

Várias linguagens de programação possuem uma classe pronta já implementada para manipular esta estrutura de dados. Em Java, por exemplo, a classe Stack oferece os métodos `push`, `pop` e `empty` para realizar as operações principais, o método `peek` para acessar o elemento que está no topo da pilha sem removê-lo dela e o método `search` para navegar pela estrutura em busca de um elemento específico e descobrir a sua posição. Ela extende a classe Vector, que extente a classe ArrayList. Estas seriam duas opções caso quiséssemos implementar uma pilha sem utilizar a classe pronta.

Podemos implementar uma pilha com um vetor utilizando uma variável para contar o número de elementos e informar onde inserir e de onde remover elementos. Entretanto, em algum momento, poderíamos ter que lidar com um problema que seria o de armazenar mais elementos do que o tamanho do vetor permite. Neste caso, poderíamos remanejar o tamanho do vetor ou utilizar um ArrayList, que faria isto por nós. Mas há ainda uma terceira opção, ideal em casos nos quais o número máximo de elementos armazenados na pilha não é conhecido, que é de utilizar uma estrutura de dados com alocação dinâmica, seguindo os conceitos de listas simplesmente encadeadas, que usam ponteiros para acessar os elementos.

3 Sobre RPN

RPN, ou Reverse Polish Notation (em português, notação polonesa inversa, também chamada de notação de pós-fixo), é uma notação matemática inventada pelo filósofo e cientista da computação Charles Hamblin por volta de 1950 com o objetivo de aprimorar a notação polonesa, ou notação de prefixo, criada pelo matemático polonês Jan Lukasiweicz por volta de 1920 para simplificar a lógica das sentenças matemáticas. Na RPN, os operadores devem vir depois (ou à direita) dos dois números a serem operados. Por exemplo, `a - b` passa a ser representado por `a b -`. Parênteses não são utilizados.

O presente trabalho utiliza o conceito de RPN para a implementação dos métodos da calculadora. O operador vem depois dos dois números a serem operados no arquivo de entrada que é lido. Entretanto, como os números são armazenados na pilha antes de serem operados, eles são retirados dela e utilizados na ordem inversa à que vieram no arquivo original. O arquivo `a b -` é transformado em `b a -` na hora de operar, seguindo o que foi solicitado no enunciado, conforme a figura 3:

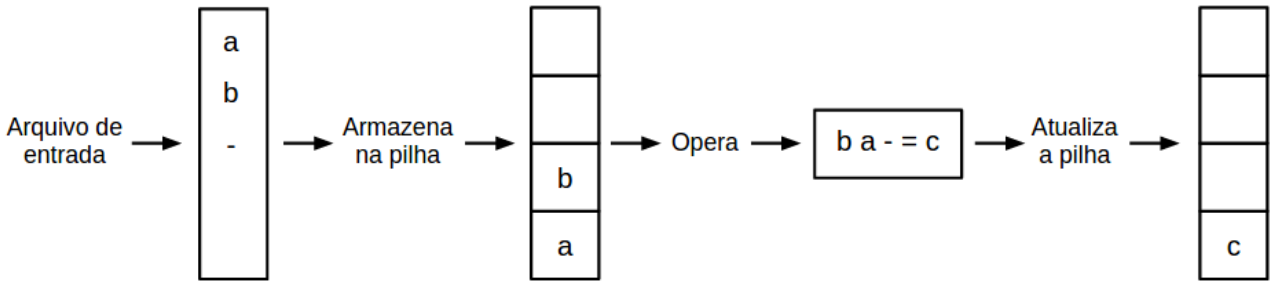

Figura 3: Utilização do conceito de RPN na operação de subtração

4 Implementando a Calculadora

Para desenvolver a calculadora deste trabalho foi utilizada uma pilha construída através de uma estrutura simplesmente encadeada. Esta estrutura foi escolhida dentre as demais (vetor, ArrayList e a própria classe Stack já implementada na linguagem Java) levando-se em consideração os seus benefícios e as necessidades do programa em questão. Estamos falando de uma calculadora que recebe uma série de números e comandos de operação através da leitura de oito arquivos de tamanhos diferentes. Ou seja, o tamanho da pilha nem sempre será o mesmo. Logo, podemos dizer que o tamanho é indeterminado. Por isso, uma estrutura de dados com alocação dinâmica é recomendada.

4.1 A estrutura simplesmente encadeada

Primeiramente, foi necessário construir a base da estrutura simplesmente encadeada. Dentro de uma classe chamada Calculadora, criamos uma classe chamada Node, que permite que somente a classe Calculadora tenha acesso à objetos do tipo Nodo. Estes possuem dois atributos: um valor do tipo `double`, que será o número armazenado na calculadora, e uma referência para um objeto do tipo Nodo que virá a seguir, ou seja, o número que vem na sequência, que chamaremos e `next`. Como estamos falando de uma pilha e precisamos ter acesso ao primeiro elemento dela, ou seja, o elemento que está no topo dela, para fazer qualquer tipo de operação, a nossa classe Calculadora, então, possui um atributo privado do tipo Nodo chamado de `first`, que aponta para o primeiro elemento no topo da pilha, de acordo com a figura 4:

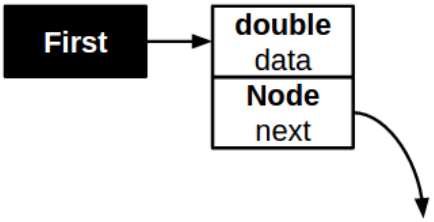

Figura 4: First apontando para o primeiro elemento da pilha

Para auxiliar na contagem de elementos armazenados na pilha e na navegação pela estrutura, criamos, também, uma variável do tipo int chamada `cont`. Esta poderia ser um método independente que navega recursivamente pela pilha e vai somando o número de elementos armazenados nela. Mas optamos por fazê-la um atributo da classe Calculadora, tendo em vista que, desta forma, ela será alterada sempre que um novo elemento for adicionado ou retirado da pilha e ficará sempre disponível. Caso contrário, toda a vez que quisermos saber o número de elementos teremos que percorrer toda a estrutura de dados e, se esta for muito grande, isso custará caro em termos de processamento e memória.

4.2 A essência da pilha

Com a base da estrutura simplesmente encadeada pronta, chegou a hora de implementar os principais métodos que caracterizam uma pilha, que realizam as operações de inserir um elemento, remover um elemento e verificar se a estrutura está vazia.

O método `push` foi solicitado no enunciado deste trabalho com o nome de `inteiro`, pois ele é responsável por armazenar na pilha os inteiros lidos no arquivo de entrada. Porém, ele também é responsável pela inserção dos números com ponto flutuante, resultantes das operações realizadas pela calculadora, que retornam um elemento para a pilha. Por isso, ele recebe por parâmetro um `double`. Assim, ele cria um novo objeto do tipo Nodo e atribui à ele o valor recebido por parâmetro. Faz ele apontar para o `first` com sua referência `next` (isto pode ser feito na própria criação do Nodo, passando a informação por parâmetro) e faz o `first` apontar para ele. Como temos uma variável `cont` que guarda o número de elementos armazenados na pilha, devemos incrementá-la. A figura 5 mostra co comportamento do método `push`:

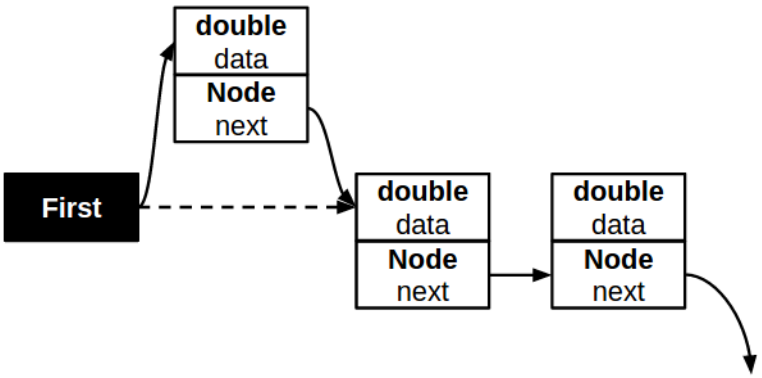

Figura 5: Comportamento do método push

O método **pop** deve remover o elemento que está no topo da pilha e retornar o valor dele, portanto seu retorno é do tipo `double`. Justamente por ele ter que retornar alguma coisa, não podemos simplesmente removê-lo. Antes disso, é necessário armazenar o valor dele em uma variável auxiliar, para então removê-lo, decrementar o `count` de elementos da pilha e, por fim, retornar a variável auxiliar. Como estamos trabalhando com Java, que é uma linguagem que possui Garbage Collector, para removê-lo podemos simplesmente alterar a referência do `first` para o `next` do Nodo a ser removido, ainda que o ideal mesmo fosse, antes disso, alterar o `next` do Nodo a ser removido para `null`. Se estivéssemos trabalhando com outra linguagem sem coletor de lixo, precisaríamos remover de fato o Nodo. A figura 6 mostra o comportamento do método `pop`:

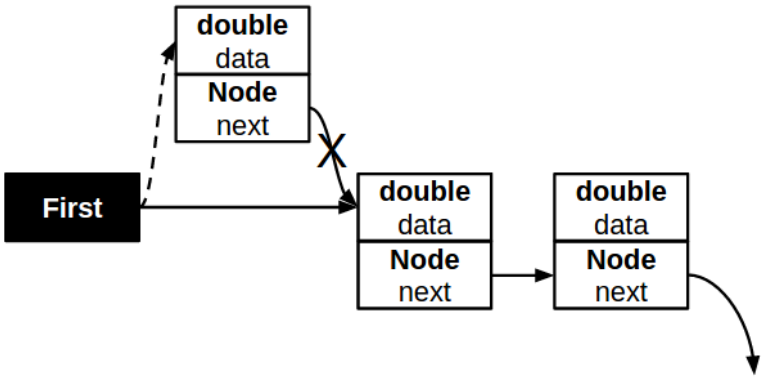

Figura 6: Comportamento do método pop

O método `isEmpty` não foi solicitado no enunciado do trabalho. Entretanto, ele foi feito para auxiliar outros métodos. Neste caso, como temos uma variável `count` que armazena o número de elementos existentes na pilha, basta verificar se esta variável é igual a 0 e retornar a resposta, tendo em vista que o retorno deste método é do tipo `boolean`.

4.3 As operações matemáticas

Os métodos `soma`, `multiplicacao`, `subtracao` e `divisao` têm uma implementação bastante parecida. Como eles utilizam dois números para realizar as operações, todos começam com uma verificação do número de elementos armazenados na pilha. Depois, como queremos operar dois números e devolver para a pilha o resultado disso, chamamos o método `pop` duas vezes para remover e retornar o valor dos números que queremos operar, fazemos a operação entre eles e chamamos o método `push` para devolver para a pilha o resultado, que é passado por parâmetro. Veja na figura 7:

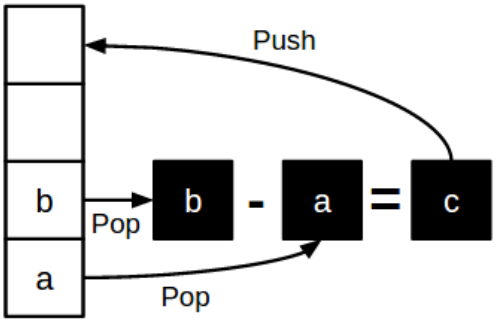

Figura 7: Comportamento do método da operação subtração

O cálculo nos métodos do `seno` e do `cosseno` segue a mesma lógica, porém, operando com apenas um número. Desta forma, não é necessário remover nenhum elemento. Podemos apenas alterar o valor do elemento no topo da pilha para o resultado da operação. Esta pode ser realizada utilizando os métodos de seno e cosseno da classe Math já existente.

Para gerar o arco-tangente, a classe Math oferece dois métodos. O primeiro, `atan`, retorna o arco tangente de um valor. O ângulo é devolvido no intervalo $\left[-\frac{\pi}{2} \ldots \frac{\pi}{2}\right]$. O segundo, *atan*2, retorna o ângulo teta da conversão de coordenadas rectangulares (x, y) para coordenadas polares (r, theta). O enunciado do trabalho especifica que devemos trabalhar com dois números. Portanto, devemos utilizar a segunda opção. A partir daí, o método se comporta da mesma maneira que as primeiras operações de soma, multiplicação, subtração e divisão. Há apenas um detalhe: a classe Math trabalha com `double`. Se a pilha do nosso programa tivesse elementos do tipo `float`, deveríamos tomar cuidado na hora de pegar o resultado da operação para devolver à estrutura. Neste caso, deveríamos fazer uma conversão de tipos do resultado para `float` e, então, devolver para a pilha.

4.4 Os demais métodos

Por fim, o enunciado do trabalho fala em mais dois métodos: o `dup` e o `swap`. O `dup` foi o mais fácil de implementar. Ele repete o último valor armazenado na pilha, ou seja, faz uma cópia dela. Para isso, podemos inserir um novo Nodo e atribuir à ele o mesmo valor do elemento que está no topo da estrutura.

O método `swap` é um pouco mais complicado. O que ele faz parece simples, trocar de lugar os dois primeiros elementos da pilha, mas envolve a alteração de uma série de referências e apontamentos entre os Nodos. Esta é uma questão muito importante quando estamos lidando com estruturas encadeadas, pois qualquer descuido pode fazer com que percamos uma referência para sempre.

Primeiramente, criamos um Nodo auxiliar que aponta para o `first`. No caso, ele guarda a referência do Nodo para o qual o `first` estava apontado, que seria o primeiro elemento da pilha, e libera o `first` para apontar para outros Nodos, como é possível ver na figura 8:

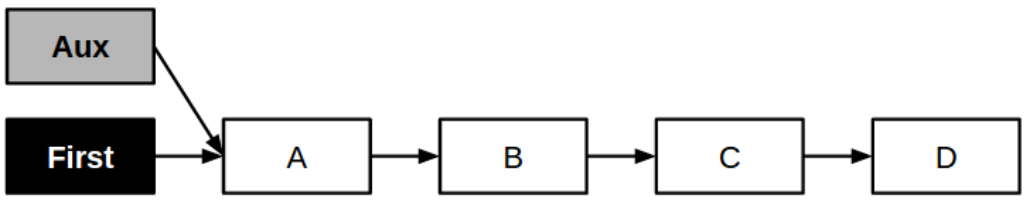

Figura 8: Primeiro passo do método swap

Desta forma, podemos mudar a referência do `first` para o `next` dele, que é o segundo elemento da pilha, sem perder a referência anterior, que ficou armazenada no auxiliar, como na figura 9:

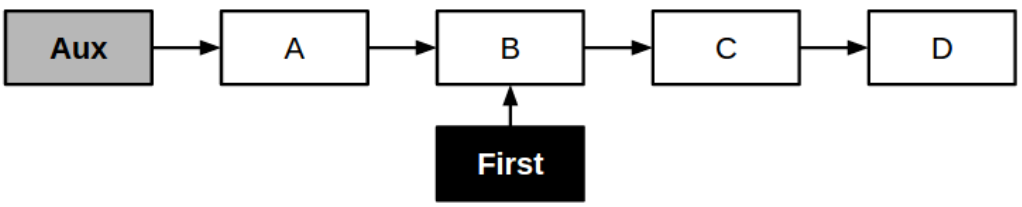

Figura 9: Segundo passo do método swap

Na sequência, mudamos a referência do `next` do Nodo para o qual o auxiliar está apontado para o `next` do Nodo para o qual o `first` agora está apontando. Ou seja, o `first` agora está apontando para o segundo elemento da pilha, então o auxiliar vai apontar para o seguinte, que é o terceiro Nodo da pilha, de acordo com a figura 10:

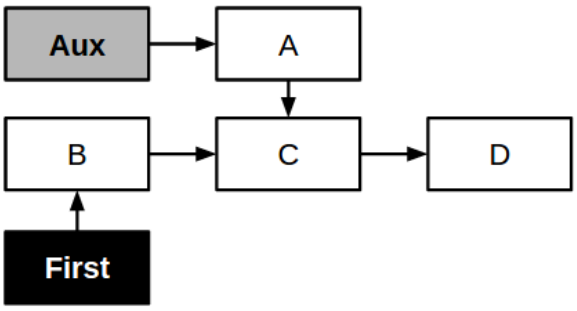

Figura 10: Terceiro passo do método swap

Por último, o Nodo para o qual o `first` está apontado, ou seja, o segundo Nodo, que agora é o primeiro Nodo, deve apontar para o Nodo que o auxiliar está guardando, que era o primeiro da pilha e agora passa a ser o segundo, e está apontando para o terceiro, da mesma forma que a figura 11:

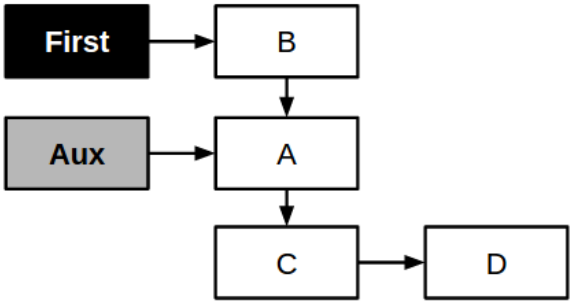

Figura 11: Quarto passo do método swap

Assim, conseguimos fazer com que o primeiro e o segundo elementos da pilha trocassem de lugar. O que era o primeiro agora é o segundo e aponta para o terceiro. E o que era o segundo deixou de apontar para o terceiro e passou a ser o primeiro, apontando para o segundo. Sem perder nenhuma referência.

5 Casos de Teste

Os casos de teste consistem em oito arquivos de texto com números inteiros e comandos de operações que são lidos pelo programa da calculadora e manipulados na classe Main, que verifica o que cada linha que está sendo lida faz (insere ou remove um novo número inteiro, faz uma operação matemática, duplica um elemento ou troca dois elementos de lugar). O enunciado do trabalho pede que, para cada caso de teste, forneçamos as seguintes informações: o tamanho máximo atingido pela pilha interna da caluladora, o tamanho da pilha ao final da execução do programa dado e o valor no topo da pilha ao fnal do programa.

Para descobrir o tamanho máximo atingido pela pilha interna da calculadora, criamos um atributo para a classe Calculadora chamado `max` que é atualizado no método `push`, de acordo com o valor da variável `count`. Ou seja, a cada vez que o `count` é incrementado, existe uma verificação para saber se ele é maior do que o número máximo atingido pela pilha já armazenado. Caso seja, a variável `max` é modificada para o mesmo valor da variável `count`.

Para saber o tamanho da pilha ao final da execução do programa, basta retornar o valor final da variável `count`.

Por fim, para saber o valor no topo da pilha ao final do programa, podemos criar uma variável auxiliar para armazenar o valor que está no topo da pilha chamando o método `pop`, depois dar um `push` com este valor para devolvê-lo à estrutura e retornar o valor que ficou armazenado na auxiliar. Podemos fazer isso na própria Main ou podemos criar um método chamado `peek` na classe Calculadora que faz exatamente isso, mas fica mais organizado e cumpre o mesmo papel do método existente na classe Stack já existente em Java, por exemplo.

A tabela 1 apresenta os resultados obtidos após rodar os oito casos de teste:

Caso de teste	**Tamanho máximo**	**Tamanho final**	**Valor no topo**
1	499	1	-2.791603
2	1465	1	3414
3	2555	1	-5519.8896
4	3539	1	1632.34
5	4497	1	10093.304
6	5478	1	7341.601
7	6621	1	1.4215462
8	7577	1	2356144.0

Tabela 1: Resultados dos casos de teste

Note que em todos eles o tamanho final da pilha foi 1 e que o tamanho máximo que a pilha atingiu foi cada vez maior.

6 Considerações

Levando-se em consideração os resultados obtidos com os oito casos de testes, podemos refletir sobre o fato de que o tamanho máximo que a pilha atigiu foi cada vez maior. No caso de teste número 8, por exemplo, a pilha chegou a armazenar 7.577 elementos. Se estivéssemos trabalhando com um vetor ou ArrayList, isso poderia ficar complicado. Da mesma forma, utilizando uma estrutura simplesmente encadeada, os métodos recursivos também exigem muito processamento e memória.

Na primeira vez em que os casos de teste foram rodados, o método `print`, que imprime todos os elementos armazenados na pilha, que foi implementado de maneira recursiva e estava sendo chamado a cada alteração realizada na estrutura, estava exigindo muito processamento e memória. Nos primeiros casos de teste, em que a pilha atinge um tamanho grande, mas aceitável, o programa demorava para rodar, mas era viável. Quanto mais perto do último caso de teste, no qual a pilha armazena mais de 7 mil elementos, foi ficando inviável. No caso de teste número 3, por exemplo, o programa estava demorando cerca de 277 segundos para rodar. Por isso, tivemos que parar de chamar o método `print`. Assim, o programa passou a rodar em menos de 1 segundo.