GERENCIAMENTO DE MEMÓRIA EM JAVA

AUTORES: FLÁVIO PINTO E HEDLEY LUNA

INTRODUÇÃO

A cada dia que passa os desenvolvedores de software necessitam de mais memória e mais programas rodando simultaneamente para poderem processar um número maior de informações. O tratamento necessário da memória utilizada não é uma tarefa fácil de ser implementada, como sabemos, a memória é um componente finito do computador, gerando erros quando a mesma não possui mais espaço disponível. Portanto, o gerenciamento de memória nas linguagens de programação têm influência direta no desempenho do sistema e na capacidade do programador em escolher a estrutura de dados adequada para determinada situação.

A gerência de memória pode ser definida como a alocação de memória de forma eficiente para empacotar tantos processos na memória quanto possível para evitar a ociosidade do processador. A linguagem Java é famosa por possuir um gerenciador de memória automático, o *Garbage Collector*, com ele é possível recuperar uma área de memória inutilizada por um programa, o que pode evitar problemas de vazamento de memória. Este processo difere com o gerenciamento manual de memória, em que o programador deve especificar explicitamente quando e quais objetos devem ser desalocados e retornados ao sistema, entretanto, algumas JVM (Java Virtual Machine) possibilitam que o programador altere as configurações de acordo com sua necessidade.

No decorrer deste trabalho, mostraremos o funcionamento da máquina virtual em relação a execução e ao gerenciamento de memória, alguns tipos de algoritmos usados para a coleta de lixo, as diferentes regiões da memória onde eles atuam, discutiremos detalhes sobre a criação e destruição de objetos em Java. Finalmente, faremos testes onde implementaremos estruturas de lista encadeada simples e vetores estático e dinâmico, apresentando uma análise comparativa ao final das implementações.

USO DA MEMÓRIA EM JAVA

RECURSOS

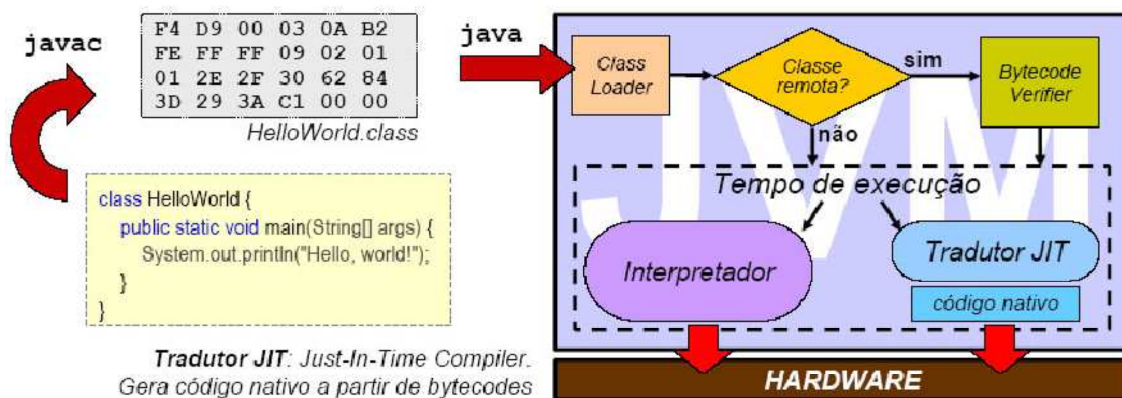
A linguagem Java é fortemente tipificada, fazendo com que toda variável e toda expressão tenha seu tipo conhecido em tempo de compilação. Os tipos em Java são classificados em tipos referência e tipos primitivos, estes são os booleanos e os primitivos, enquanto aqueles são os tipos *class*, *interface* e *array*, existindo também um tipo especial chamado de *null*. Os tipos referência são assim denominados porque seus valores são referências para objetos. No entanto, as desreferenciações em Java são sempre implícitas.

A utilização de referências do Java é a principal responsável pela segurança da linguagem, diferente do C++, que utiliza ponteiros explícitos. A JVM não permite que o programador tenha acesso direto ao endereço de memória, não permitindo que o programador descubra o endereço real da variável. Nesta seção, mostraremos alguns recursos necessários para entender o gerenciamento de memória desta linguagem.

JVM

Na Ciência da computação, máquina virtual é o nome dado a uma máquina implementada através de software, executando programas de forma semelhante a um computador real. Atualmente a JVM é uma das mais importantes máquinas virtuais, existem simuladores para ela em grande parte dos computadores atuais, desde computadores de grande porte até telefones celulares, tornando as aplicações Java extremamente portáteis.

A JVM executa um código de máquina portável (*bytecode*) armazenado em formato de arquivo .class, geralmente gerado por uma compilação de um código-fonte Java. Diferentemente de linguagens mais antigas, como FORTRAN e COBOL, em que os compiladores determinavam a quantidade de memória, a JVM esconde detalhes de segmentação da memória do programador (onde fica a pilha, heap, etc.), também não informando qual o algoritmo utilizado para liberar memória.



A forma utilizada para organizar a memória depende da JVM utilizada, existem diferentes implementações para JVM. Exemplos de maquinas virtuais Java são:

- *Sun HotSpot JVM* - É a primeira máquina virtual Java para desktops e servidores produzidos pela Oracle Corporation. Ela apresenta técnicas em tempo de compilação fazendo adaptações destinadas a melhorar o desempenho.
- *Sun KVM* - Desenvolvida para dispositivos móveis, como palmtops e celulares, tendo como objetivo executar aplicações J2ME.

- *Jikes JVM*: é uma máquina de código aberto, desenvolvida pela IBM, geralmente utilizadas por cientistas. Grande parte dos artigos sobre coleta de lixo são testados na Jikes JMV, mesmo que sejam desenvolvidos para outras plataformas, como .NET.

A JVM também possui uma ferramenta chamada *javap* que permite visualizar o conteúdo de um arquivo de classe. Essas informações fornecidas pela máquina virtual são obtidas usando as opções *–c* e *–verbose*, através delas também é possível ver a sequencia de instruções da JVM, o tamanho dos quadros de cada método, o conteúdo dos quadros, o *pool* de constantes, etc.

Exemplo de sintaxe:

```
javap -c nome.da.Classe
```

ÁREAS DE DADOS

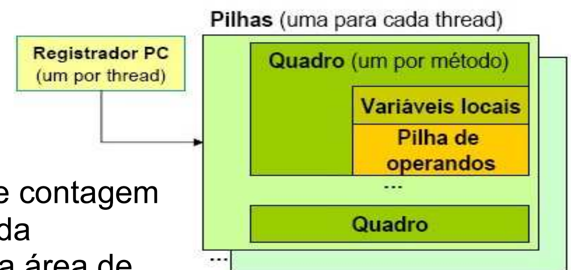
Entre as várias estratégias de alocação de memória, as principais são:

- Alocação *estática* - Os dados possuem um tamanho fixo e estão organizados seqüencialmente na memória do computador. As áreas de memórias são alocadas antes da execução do programa, não permitindo mudanças nas estruturas de dados.
- Alocação *linear* - Memória alocada em fila ou pilha, obrigando a remoção de objetos na ordem definida na criação.
- Alocação *dinâmica* - Os dados não precisam ter um tamanho fixo, pois podemos definir para cada dado quanto de memória que desejamos usar. Permitindo liberdade de criação e remoção em ordem arbitrária. Este tipo de alocação requer controle sobre o espaço ocupado e a identificação dos espaços livres.

O Java utiliza alocação dinâmica (*heap*) para objetos e alocação linear (*pilha*) para procedimentos sequenciais, mas todo o gerenciamento é feito automaticamente. Para os programadore de Java, as áreas da memória virtual conhecidas como *heap* e *pilha* são lugares imaginários, não havendo a necessidade de se preocupar, como ocorre em C ou C++, com a escolha do local para as alocações. O importante para o programador Java é ele conhecer onde são feitas as alocações para determinado tipo de dados.

Portanto, o estudo aprofundado sobre gerência de memória, da forma como existe em C ou C++, não é necessário em Java. É sabido que a escolha de algoritmos e arquitetura para JVM é de suma importância para saber como configurá-la para obter uma melhor performance. Mesmo com essa possibilidade de configuração, poucos parâmetros são disponibilizados para

alteração, sendo os mesmos voltados para administradores, oferecendo poucas opções para os programadores.



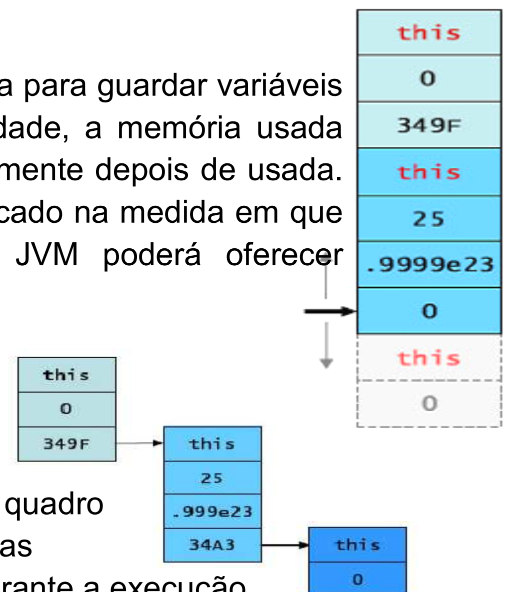
REGISTRADORES

Cada *thread* possui seu próprio registro de contagem de Programa. Eles são atualizados depois de cada *bytecode* executado, sendo estes encontrados na área de métodos. Os registradores da JVM são constituídos basicamente de quatro partes:

- FRAME - É uma referencia ao *stack frame*, contém um ponteiro para o método em execução, sendo este o primeiro método (frame) na *stack* da *thread*.
- OPTOP - Contém um ponteiro para o topo da operando *stack*, sendo utilizado para avaliar expressões aritméticas.
- VARS - Contém um ponteiro para as variáveis locais do método em execução.
- PC – Contém o endereço do próximo *bytecode* a ser executado.

PILHAS

Cada pilha tem um *thread* associado, ela é usada para guardar variáveis locais e resultados parciais. Dependendo da necessidade, a memória usada pela *pilha será* alocada no *heap* e liberada automaticamente depois de usada. A pilha pode ter um tamanho fixo ou poderá ser modificado na medida em que for necessário. Dependendo da implementação, a JVM poderá oferecer controle para o ajuste do tamanho de pilhas.



QUADROS E PILHAS

Cada vez que um método é chamado cria-se um quadro associado à ele, sempre associado ao thread local. Essas chamadas continuamente criam e destroem quadros durante a execução de operações. Todo quadro possui um vetor de variáveis e uma pilha LIFO conhecida como pilha de operandos, onde são carregados constantes e valores de variáveis locais, também preparando parâmetros a serem passados a métodos e para receber seus resultados.

HEAP

O *heap* é a área da memória compartilhada por todos os *threads*, onde os dados são alocados dinamicamente, podendo ter tamanho fixo ou não. Esta área é criada quando a máquina virtual é iniciada, não necessitando de uma parte contígua de memória. Os espaços alocados são reciclados automaticamente pelo coletor de lixo. O *heap* possui a área de métodos onde são guardados os códigos compilados de métodos e construtores. A área de métodos é criada na inicialização da JVM e geralmente alocada em uma área de alocação permanente, com localização não determinada pela especificação da máquina virtual.

ALOCAÇÃO E LIBERAÇÃO DE MEMÓRIA

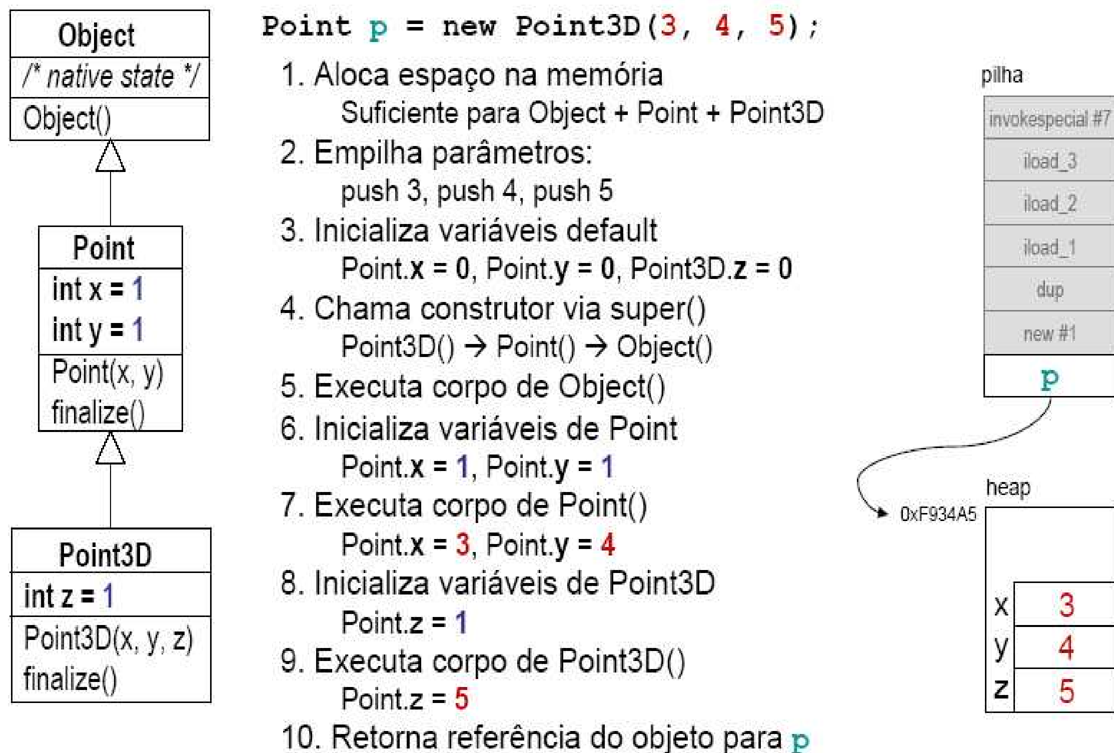
Quando um objeto é criado a JVM automaticamente aloca memória no *heap*, criando também uma referência deste objeto na pilha. Apenas os objetos String podem ser criados implicitamente, já os outros, são criados explicitamente através de uma expressão *new Classe()* ou através do método *newInstance()*, sendo os mesmos destruídos automaticamente pela JVM.

Nesta seção vamos explorar o processo de construção e destruição de objetos na visão do programador.

CRIAÇÃO DE OBJETOS

Quando criamos uma nova instância de uma classe, a JVM aloca memória para todas as variáveis de instância declaradas na classe e superclasse, gerando um erro se não houver memória suficiente. Após o endereço para o objeto ter sido reservado às variáveis são inicializadas com seus valores default, sendo posteriormente instanciadas com os valores passados como argumento. Quando chegar na classe *Object*, a única que não possui *super()*, a JVM executa o construtor dela e desce para o próximo construtor da hierarquia até chegar no construtor chamado pela instrução *new*.

Observe passo a passo a construção de um objeto na próxima ilustração:



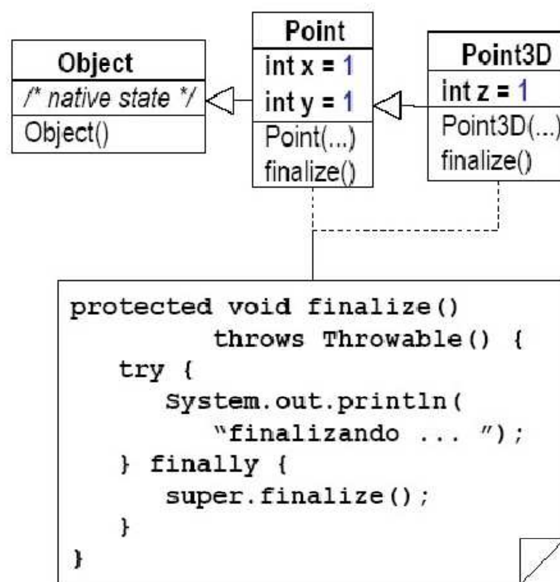
DESTRUIÇÃO DE OBJETOS

Como dito anteriormente, o programador Java não se preocupa com a destruição de objetos e possui poucas opções que interferem no coletor de lixo, entre elas podemos citar:

- Finalização.
- Chamadas explícitas ao coletor de lixo.
- Remoção das referências para um objeto.

O Java garante que antes da memória reservada a um objeto for liberada, haverá um finalizador para ele chamado através de um *thread*. No Java nem todos os objetos necessitam de finalizadores, sendo estes de uso obrigatório em arquivos, soquetes, *streams* e *threads*. Diferente do C e C++ o finalizador do Java (método *finalize()*) é opcional, já que ele é chamado automaticamente quando o objeto não for mais alcançado por referências raiz, exceto quando ele for chamado explicitamente e não haver necessidade de liberar memória.

Podemos observar o processo de destruição de objetos na figura seguinte:




```
p = null; // p é Point3D
```

1. Espera coleta de lixo
   Eventualmente GC executa
2. Objeto em fila de finalização
   Eventualmente GC tira objeto da fila
3. Executa **finalize()** de Point3D
   Imprime "finalizando ..."
   Chama super.finalize()
4. Executa **finalize()** de Point
   Imprime "finalizando ..."
   Chama super.finalize()
5. Executa **finalize()** de Object
   Termina finalize() de Point3D
6. Objeto finalizado espera liberação
   Eventualmente liberação ocorre
7. Objeto destruído

Na destruição explícita de objetos temos que tomar cuidado com o uso do método *finalize(),* este método somente deve ser utilizado de forma correta, já que ele não chama automaticamente o finalizador de sua superclasse. Um correto uso desse método deve chamar *super.finalize()* explicitamente, sendo recomendado colocá-lo dentro de um bloco *finally*, bloco que será executado independente de uma exceção.

```
protected void finalize() throws Throwable {
    try {
        // código de finalização
    } finally {
        super.finalize();
    }
}
```

POSSÍVEIS ERROS DE MEMÓRIA

- *OutOfMemoryError* - É apresentado quando a máquina virtual não consegue alocar um objeto por falta de memória e o *Garbagge Collector* não liberou mais memória ainda. Este é um erro que não deve ser encontrado por um bloco de tratamento de exceções (*try* ... *catch*), ou seja, se não há memória para alocar novos objetos, é óbvio que não haverá memória para tal procedimento, sendo recomendado deixar o programa terminar e verificar a causa do problema.
- *StackOverflowError* - É apresentado quando um *thread* precisar de uma pilha maior que a permitida. Este erro tem como principal fonte os métodos com muitas variáveis locais ou funções recursivas.

GARBAGE COLLECTOR

Um conhecimento sólido sobre o *Garbage Collector* (GC) é fundamental para se escrever aplicações robustas e de alto desempenho na plataforma Java. Se uma aplicação usa uma quantidade de memória tal que força o sistema operacional a usar memória virtual, esta aplicação sofrerá um impacto no desempenho. O GC tem como função procurar objetos que não são mais referenciados, liberando a memória alocada para eles. É impossível prever quando o GC vai entrar em ação, mas a JVM permite que o programador force sua chamada, através da instrução *System.gc()*, mas isso não garante a remoção de todos os objetos.

VANTAGENS

- Livra o programador de lidar manualmente com o gerenciamento de memória.
- Redução ou eliminação de certas categorias e defeitos de software.
- Não necessita a desalocação manual de memória.
- Liberar mais de uma vez uma região de memória.
- Diminui a probabilidade de ocorrer um esgotamento de memória.

DESVANTAGENS

- Utiliza processos que consomem recursos computacionais para decidir quais partes da memória podem ser liberadas.
- Diminui a eficiência do algoritmo devido à sobrecarga no uso do coletor.

ALGORITMOS PARA COLETA DE LIXOS

A liberação automática de memória do *heap* é realizada através de algoritmos de coleta de lixo. Há várias estratégias, com vantagens e desvantagens de acordo com a taxa em que objetos são criados e descartados. Apesar do impacto gerado na performance, a gerência explícita de memória também o possui, mas é muito mais complicada.

Entre os principais desafios dos algoritmos de coleta de lixo podemos citar a distinção do que não é lixo e a influência da coleta em alocações posteriores. Nas JVMs atuais todos os coletores utilizam algoritmos exatos, ou seja, garantem a identificação precisa de todos os ponteiros e a coleta de todo o lixo, tendo como principal critério para definir o que é lixo a alcançabilidade,

objetos alcançados através de uma corrente de referências partindo de um conjunto raiz de referências.

De acordo com a especificação da linguagem Java, uma JVM precisa incluir um algoritmo de coleta de lixo, não necessariamente informado, para reciclar memória não utilizada.

Entre os principais algoritmos podemos citar:

- *Reference counting algorithm* [Collins 1960] - Mantém, em cada objeto, uma contagem das referências que chegam nele. Objetos que têm contagem zero são coletados.
- *Cycle collecting algorithm* [Bobrow 1980] - Estende o algoritmo de contagem de referência para coletar ciclos (referências circulares).
- *Mark and sweep algorithm* [McCarthy 1960] - Rastrear objetos do *heap*, marca o que não é lixo e depois varre o lixo (libera memória).
- *Mark and compact algorithm* [Edwards] - Estende o algoritmo *Mark and sweep* que mantém o *heap* desfragmentado depois de cada coleta.
- *Copying algorithm* [Cheney 1970] - Divide o *heap* em duas partes. Cria objetos em uma parte do *heap* e deixa outra parte vazia. Recolhe o que não é lixo e copia para a área limpa, depois esvazia a área suja.

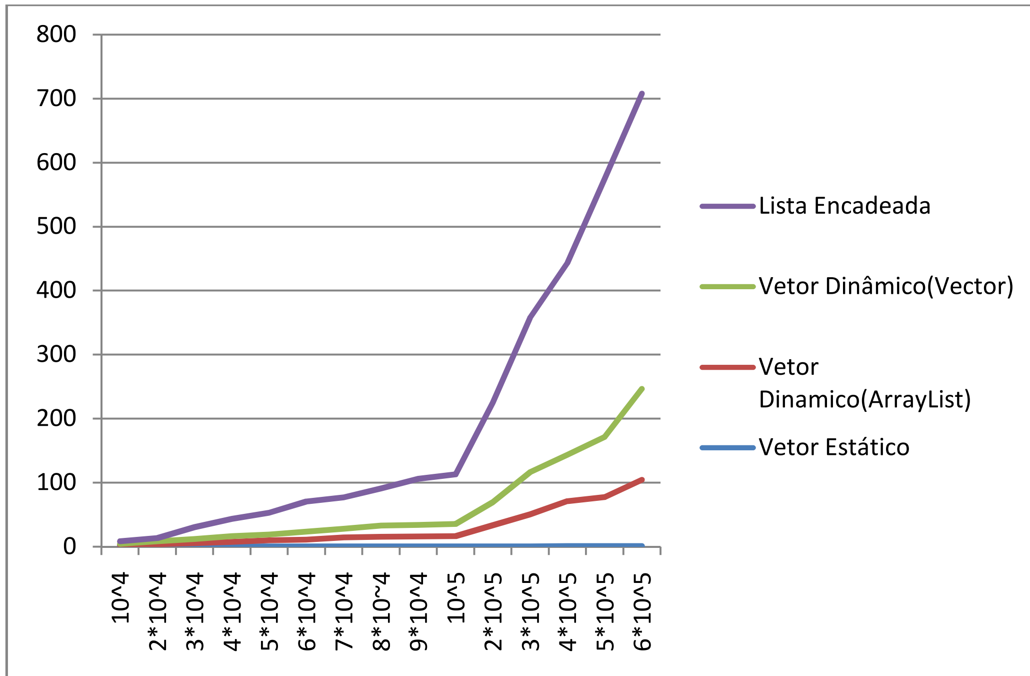
TESTES

Os testes consistiram em alocar uma estrutura de dados e posteriormente preechê-la com qualquer valor, para então coletar os dados sobre os tempos de execução. Os testes não se limitaram não apenas a uma linguagem e a um Sistema Operacional. Avaliamos tanto a linguagem Java quanto o C++ nos SO's: Ubuntu Linux 10.4 e Microsoft Windows Seven Ultimate Edition.

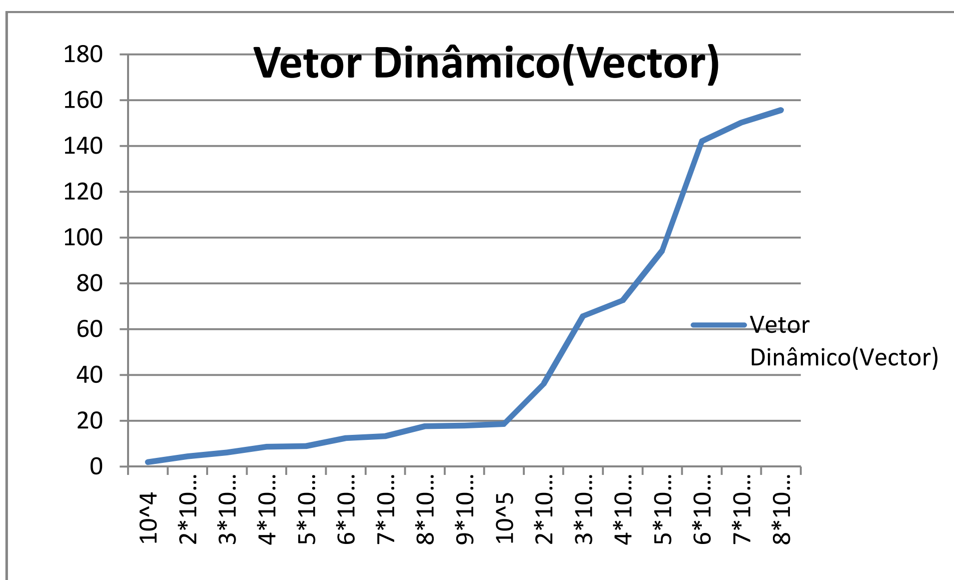
CONFIGURAÇÃO DA MÁQUINA UTILIZADA PARA OS TESTES

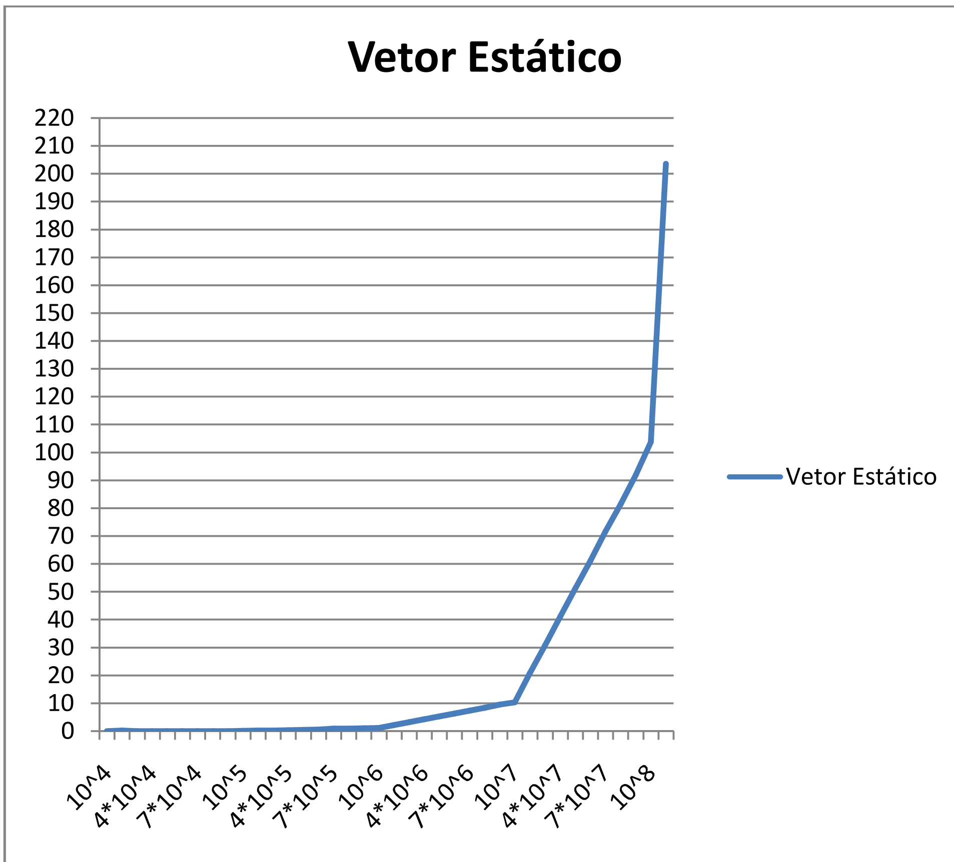
- **Processador:** Intel Core 2 Quad I5-650 3.2GHZ
- **Cache do processados:** 8MB
- **Memória RAM:** 4GB DDR3 1333
- **HD:**1TB SATA 2

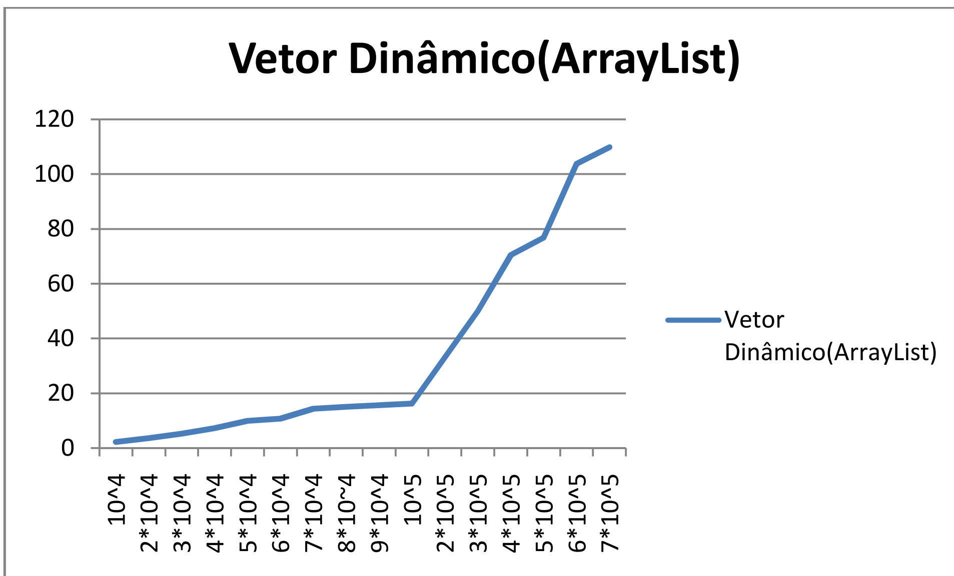
As estruturas avaliadas foram: vetor estático(stack), vetor dinâmico (heap) e lista simplesmente encadeada. Para a lista encadeada do C++ foi-se usada o contêiner List da Standard Template Library (STL). A seguir segue o gráfico comparativo dessas estruturas em Java no SO's Windows.

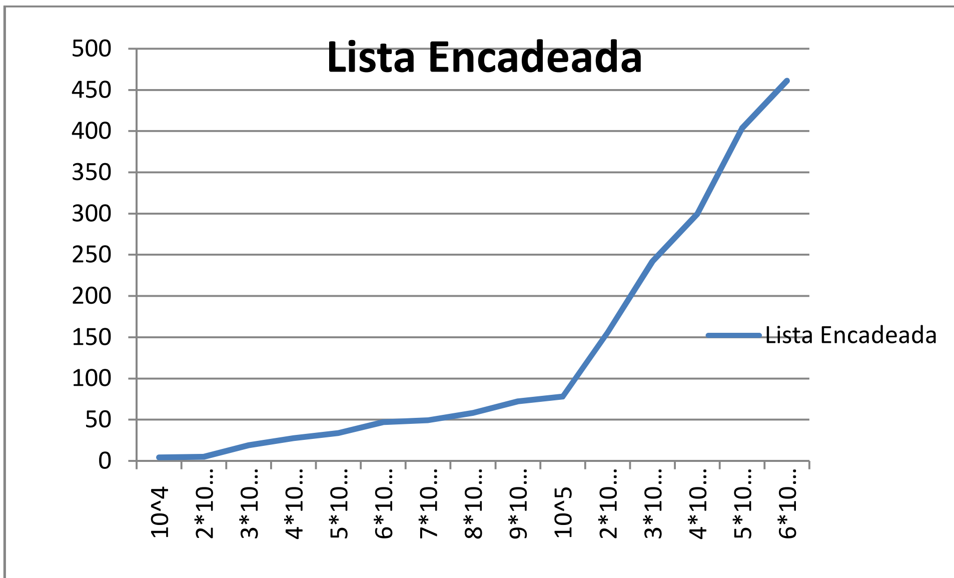


Como podemos ver a lista encadeada é a mais lenta das estruturas e o vetor estático é, de longe, a mais veloz. Uma observação a ser feita alem do tempo de execução das estruturas é a quantidade de elementos que cada estrutura suportou até estourar a memória. A lista encadeada conseguiu alocar até 6*10^5 elementos, já o vetor dinâmico do tipo ArrayList suportou até 7*10^5 elementos, o vetor dinâmico do tipo Vector 8*10^5 e o vetor estático 2*10^8. O gráfico dá uma falsa impressão de que o vetor estático consumiu nenhum tempo para sua execução, mas isso não é verdade, o gráfico apresenta dessa forma devido a dificuldade de por uma escala menor, segue abaixo os gráficos das respectivas estruturas individualmente.

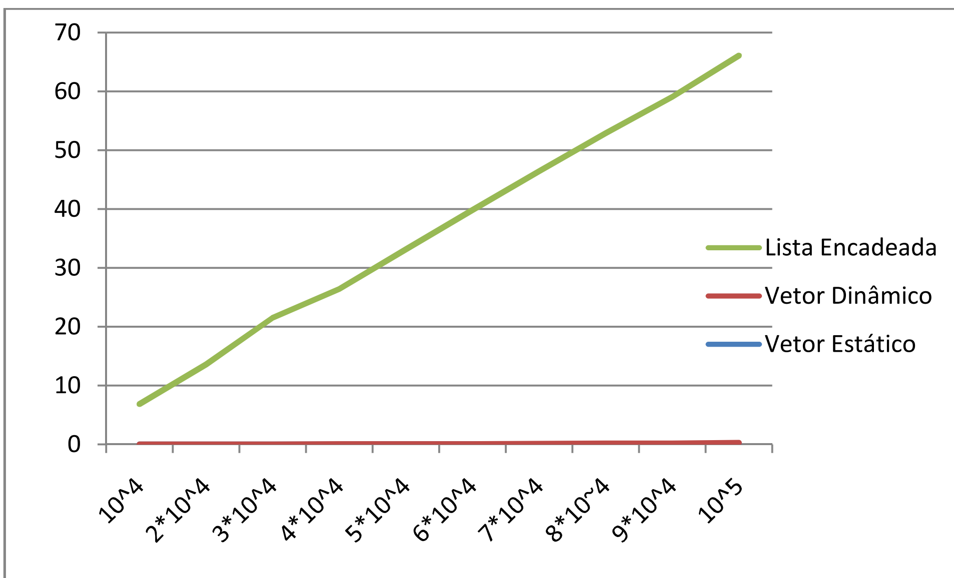



Vetor Estático
220
210
200
190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
10^4
4*10^4
7*10^4
10^5
4*10^5
7*10^5
10^6
4*10^6
7*10^6
10^7
4*10^7
7*10^7
10^8
Vetor Estático


Vetor Dinâmico(ArrayList)
120
100
80
60
40
20
0
10^4
2*10^4
3*10^4
4*10^4
5*10^4
6*10^4
7*10^4
8*10~4
9*10^4
10^5
2*10^5
3*10^5
4*10^5
5*10^5
6*10^5
7*10^5
Vetor Dinâmico(ArrayList)

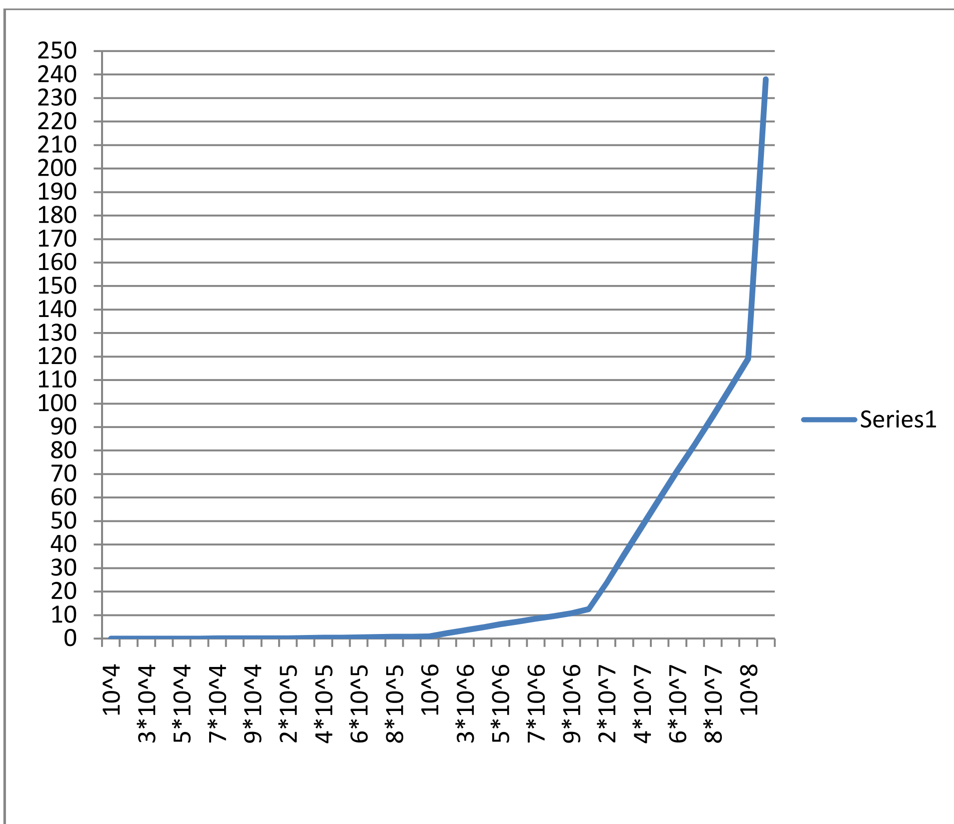


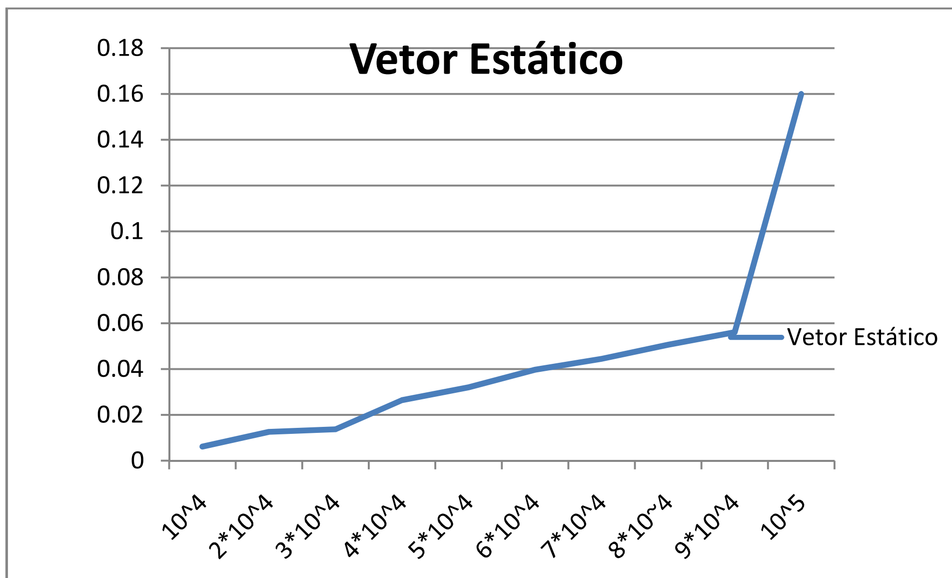
Agora vamos analisar a linguagem C++ no mesmo SO.



Mais uma vez o gráfico nos engana, pois podemos pensar que tanto o vetor estático como o dinâmico não consumiram tempo algum, para sanar essa dúvida segue abaixo os gráficos individuais.

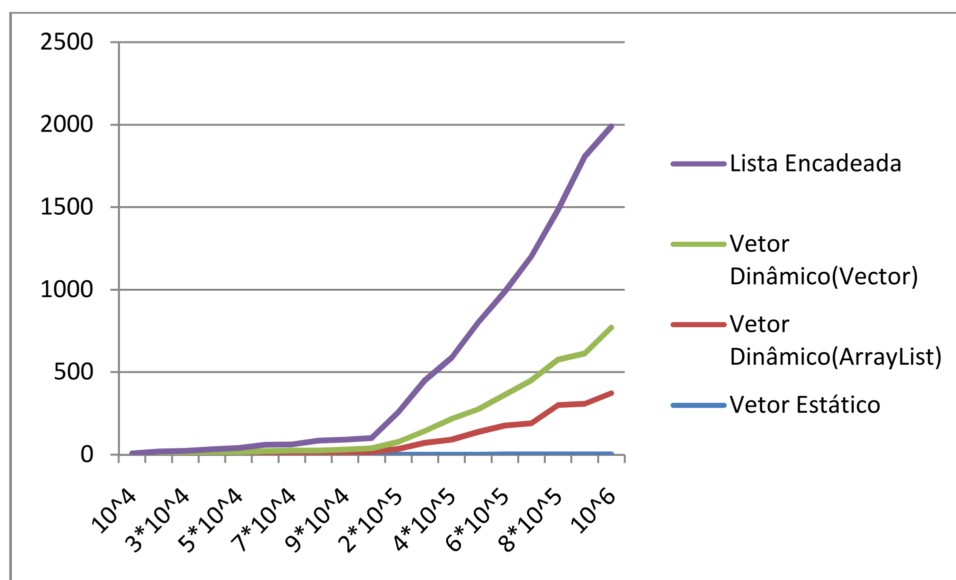
Lista Encadeada
700
600
500
400
300
200
100
0
10^4
3*10^4
5*10^4
7*10^4
9*10^4
2*10^5
4*10^5
6*10^5
8*10^5
10^6
Lista Encadeada


250
240
230
220
210
200
190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
10^4
3*10^4
5*10^4
7*10^4
9*10^4
2*10^5
4*10^5
6*10^5
8*10^5
10^6
3*10^6
5*10^6
7*10^6
9*10^6
2*10^7
4*10^7
6*10^7
8*10^7
10^8
Series1



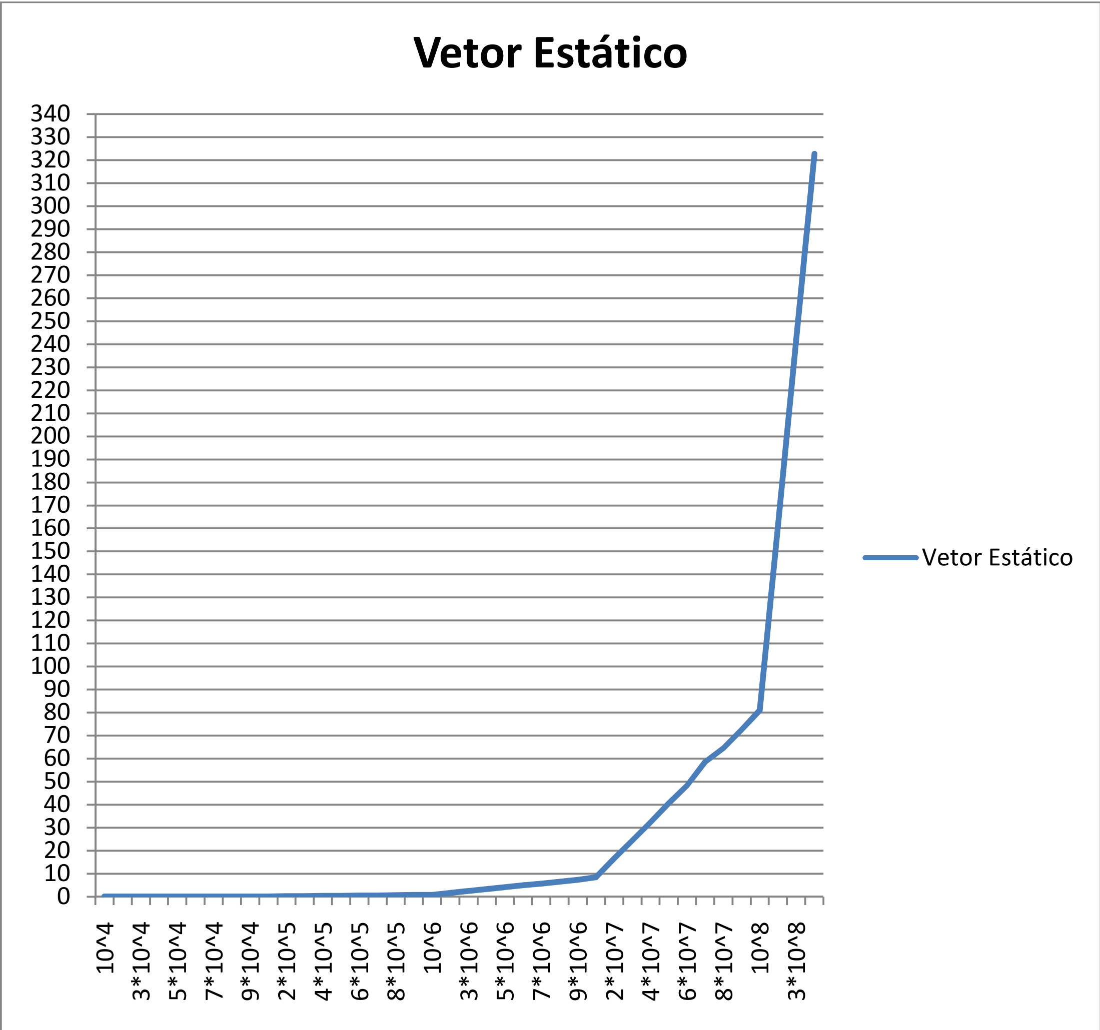
É complicado analisar o Java e o C++ no Windows, pois o gerenciamento de memória acontece de modo totalmente diferente, basta notar que o vetor estático do Java suporta até 2*10^8 elementos, já o do C++ só suporta 10^5. A única estrutura que podemos avaliar entre as duas linguagens é a lista encadeada. O Java alocou e preencheu a lista de 6*10^5 elementos em 461,04ms, já o C++ demorou 389,7 ms para a mesma quantidade. Com isso temos que o C++ foi cerca de 15% mais rápido que o Java, o que é aceitável visto que o Java é uma linguagem interpretada e o C++ é compilada.

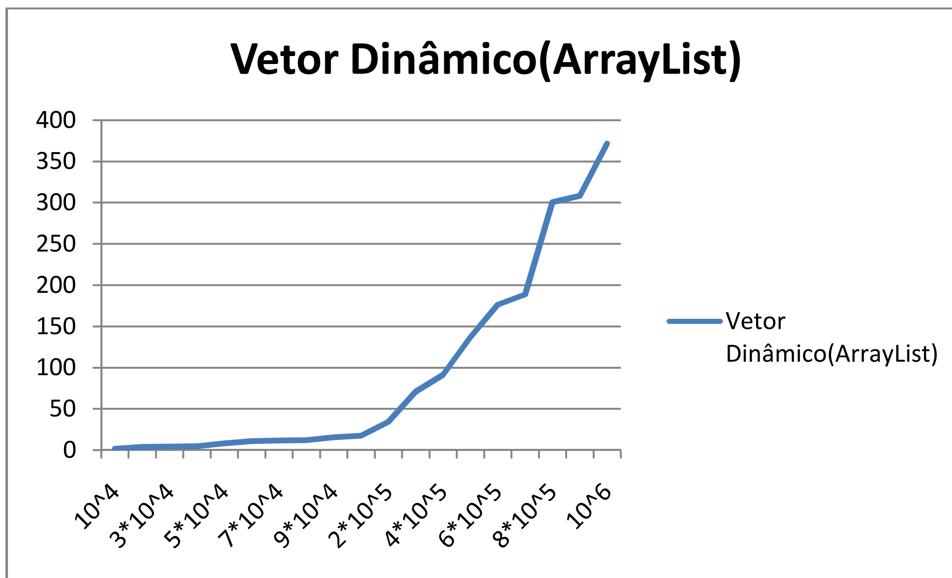
Agora vamos analisar as mesmas estruturas nas mesmas linguagens, mas no SO Ubuntu Linux. Segue abaixo o gráfico comparativo entre as estruturas usando o Java.

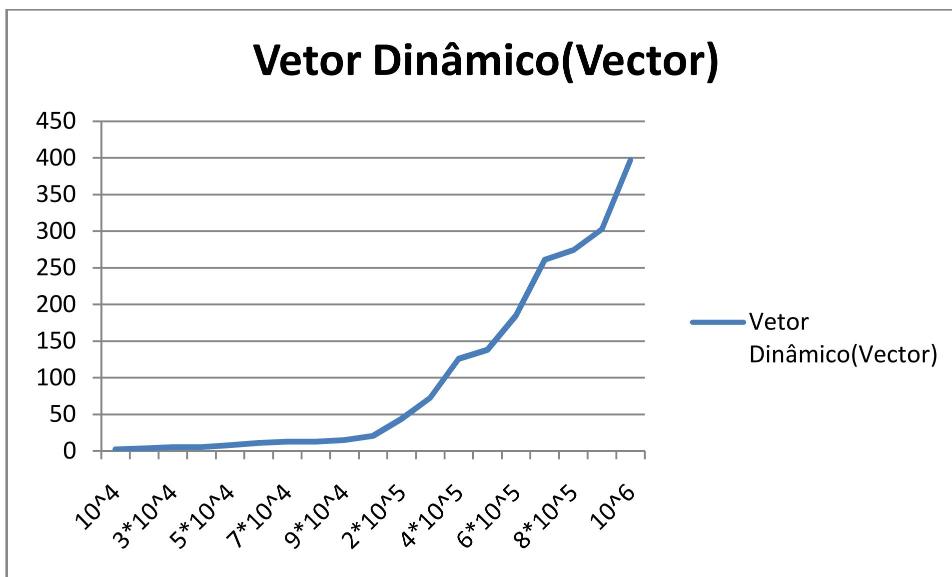


Mais uma vez o gráfico nos dá a falsa impressão de que o vetor estático não consome tempo algum, e por isso segue a abaixo os gráficos de cada estrutura em particular.

Lista Encadeada
1400
1200
1000
800
600
400
200
0
10^4
3*10^4
5*10^4
7*10^4
9*10^4
2*10^5
4*10^5
6*10^5
8*10^5
10^6
Lista Encadeada

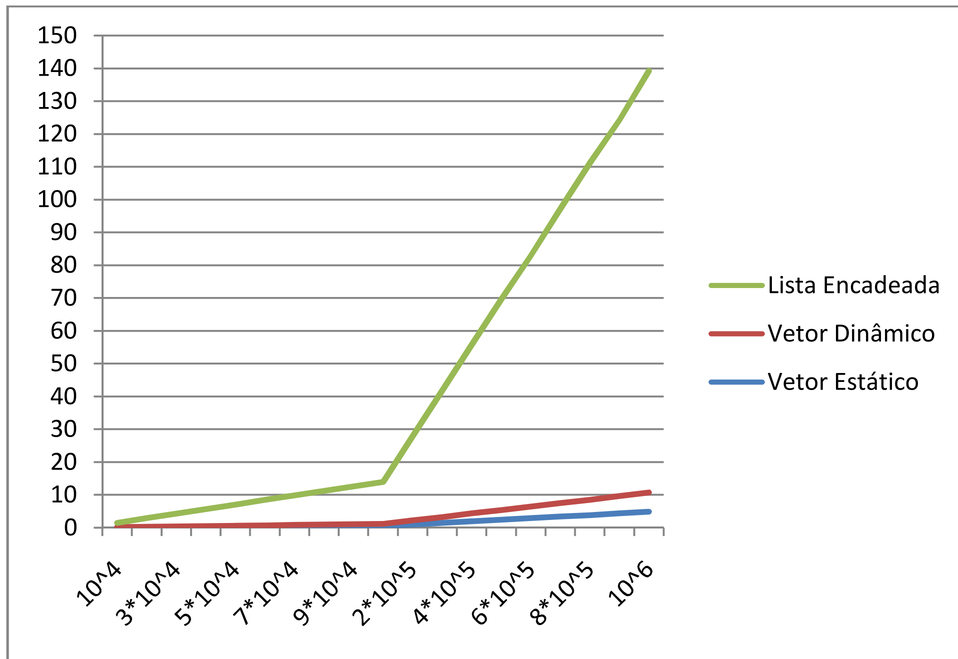

Vetor Estático
340
330
320
310
300
290
280
270
260
250
240
230
220
210
200
190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
10^4
3*10^4
5*10^4
7*10^4
9*10^4
2*10^5
4*10^5
6*10^5
8*10^5
10^6
3*10^6
5*10^6
7*10^6
9*10^6
2*10^7
4*10^7
6*10^7
8*10^7
10^8
3*10^8
Vetor Estático



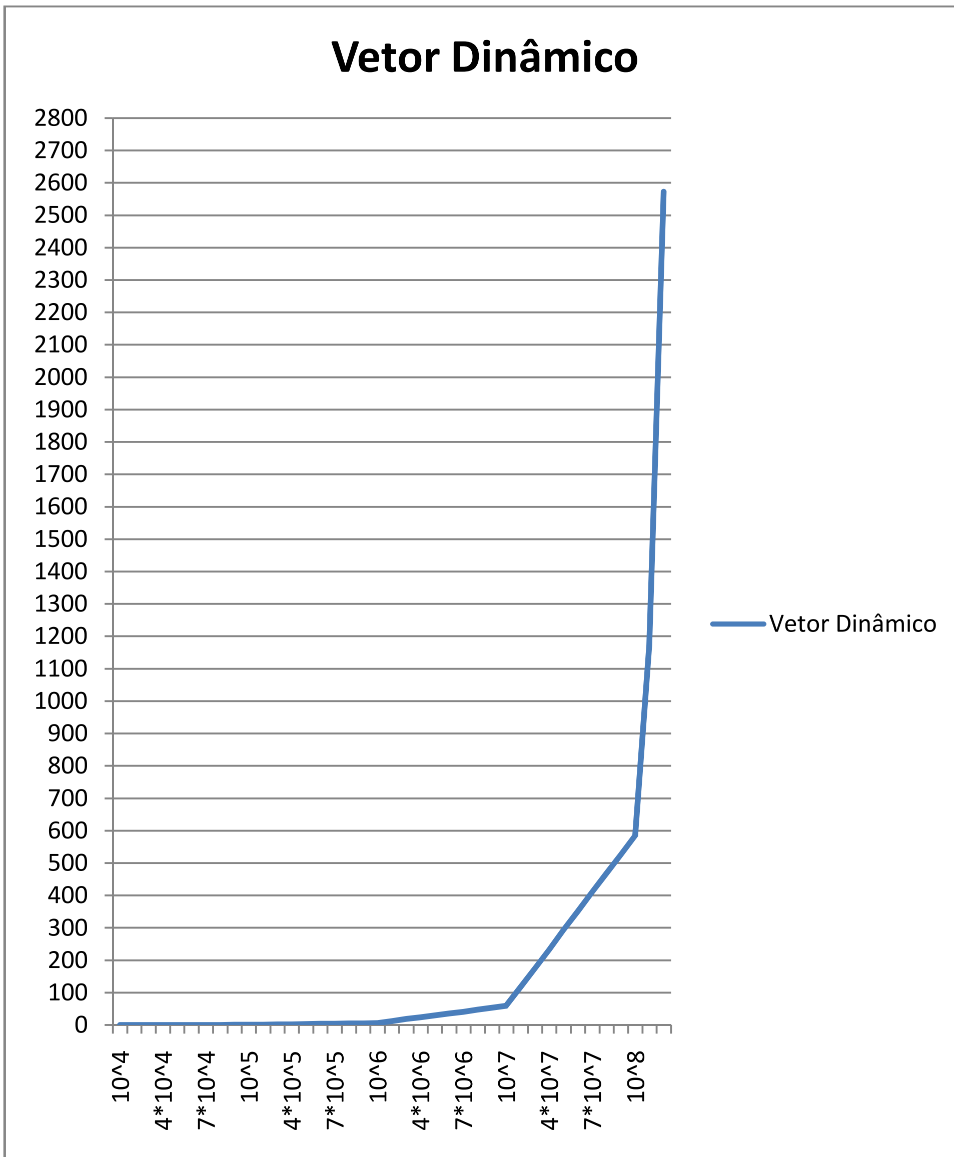


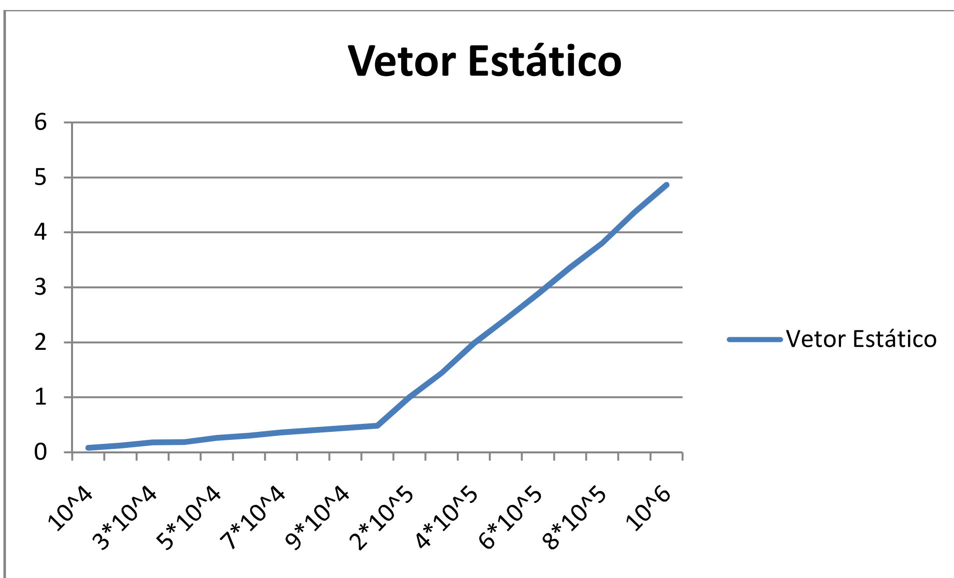
A maior diferença notada entre os SO's foram a quantidade de elementos alocados para cada estrutura, em todas as estruturas a quantidade de elementos alocados foi maior no Linux do que no Windows. Por exemplo, o vetor estático no Windows foi até 2*10^8 elementos, já no Ubuntu foi até 4*10^8, essa diferença é enorme devido a ordem de grandeza entre elas, cerca de 200 milhões de elementos a mais.

Agora vamos analisar o gráfico comparativo das estruturas no C++ no Ubuntu.



Esse gráfico retrata melhor a diferença entre as estruturas, pois, ao contrário dos gráficos anteriores, o vetor estático não está zerado. Percebemos mais uma vez a grande diferença entre a lista encadeada e as outras estruturas. Para deixar mais claro como cada estrutura de comportou segue abaixo os gráficos individuais.


Vetor Dinâmico
2800
2700
2600
2500
2400
2300
2200
2100
2000
1900
1800
1700
1600
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
10^4
4*10^4
7*10^4
10^5
4*10^5
7*10^5
10^6
4*10^6
7*10^6
10^7
4*10^7
7*10^7
10^8
Vetor Dinâmico


Vetor Estático
6
5
4
3
2
1
0
10^4
3*10^4
5*10^4
7*10^4
9*10^4
2*10^5
4*10^5
6*10^5
8*10^5
10^6
Vetor Estático

Nesse comparativo, assim como no Java, vemos a diferença entre a quantidade de elementos alocados no C++ entre os SO's. Por exemplo, a lista encadeada no Ubuntu foi até 4*10^6 elementos, já no Windows só foi até 10^6 elementos, o que dá uma diferença de cerca de 4 milhões de elementos. No Ubuntu, o C++ foi cerca de 10 vezes mais rápido que o Java. Em tempo absoluto temos, para 10^6 elementos, 128,46 ms do C++ contra 1220,04 ms do Java.

Finalmente, fazendo uma análise entre as estruturas, temos que a lista encadeada é disparada a mais lenta, pois há o custo de manter os elementos espaçados na memória. A estrutura mais rápida foi o vetor estático, como era esperado. Fazendo uma análise entre as linguagens, temos que o Java é mais lento que o C++, o que era esperado. O Java foi cerca de 15% mais lento no Windows e 10 vezes mais lento no Ubuntu, essa diferença pode ser explicada por uma possível diferença entre as implementações da JVM do Linux e do Windows. Fazendo um comparativo entre os SO's, temos que o C++ foi 5 vezes mais rápido no Ubuntu que o no Windows. Já o Java, ao contrário do esperado, foi cerca de 30% mais rápido no Windows do que Ubuntu. Essa é mais uma evidência de que,talvez a JVM do Windows tenha uma implementação diferente da JVM do Ubuntu.

REFERÊNCIAS

http://waltercunha.com/blog/index.php/2009/02/06/sobre-referencias-ponteiros-e-java/
http://pt.wikipedia.org/wiki/M%C3%A1quina_virtual
http://www.argonavis.com.br/cursos/java/j190/TutorialGerenciaMemoriaJava.pdf
http://javaguiadoscuriosos.blogspot.com/2009/08/primeiramente-ando-meio-sem-tempo-para.html
http://pt.wikipedia.org/wiki/M%C3%A1quina_virtual_Java
http://en.wikipedia.org/wiki/Java_Virtual_Machine
http://www.arquivodecodigos.net/dicas/java-entendendo-o-erro-outofmemoryerror-2142.html
http://javafree.uol.com.br/artigo/1386/Garbage-Collection.html
http://pt.wikipedia.org/wiki/Coletor_de_lixo