Resumo Sobre Estrutura de Dados em JAVA

Memória RAM

- **Memória RAM (Random Access Memory)** é um tipo de memória volátil usada para armazenar dados temporariamente enquanto o computador está ligado. Ela permite a leitura e escrita rápida de dados, facilitando a execução de programas e processos pelo sistema operacional. Quando o computador é desligado, todos os dados armazenados na RAM são perdidos.
- **Chips de memória**: A memória RAM é composta por vários chips que armazenam endereços de memória acessados pelo sistema operacional quando há uma requisição do usuário.
- **Dados binários e decimais**: Correto, os dados na memória são armazenados em formato binário (bits). Por exemplo, 0001 (binário) = 1 (decimal), 0010 (binário) = 2 (decimal), e assim por diante.
- **Bytes**: Um byte é composto por 8 bits. No entanto, uma variável `int` geralmente ocupa 4 bytes (32 bits) de memória, não 1 byte. Um `boolean` pode ocupar 1 byte, mas isso pode variar dependendo da linguagem de programação e do compilador.
- **Tipos de RAM**:
    - **DRAM (Dynamic RAM)**: É o tipo mais comum de RAM usada em computadores e dispositivos móveis. Ela precisa ser constantemente atualizada para manter os dados.
    - **SRAM (Static RAM)**: Mais rápida e cara que a DRAM, usada principalmente em cache de CPU. Não precisa ser atualizada constantemente.
- **Latência e Velocidade**:
    - **Latência**: Refere-se ao tempo que leva para a RAM responder a uma solicitação de leitura ou escrita.
    - **Velocidade**: Medida em MHz, indica a taxa de transferência de dados. RAM com maior velocidade pode melhorar o desempenho do sistema.
- **Dual Channel e Quad Channel**:
    - **Dual Channel**: Permite que dois módulos de RAM trabalhem juntos para dobrar a largura de banda.
    - **Quad Channel**: Permite que quatro módulos de RAM trabalhem juntos, aumentando ainda mais a largura de banda.

Tipos de Dados Primitivos em Java

Tipo de Dado	Tamanho (bits)	Tamanho (bytes)	Valor Mínimo	Valor Máximo	Precisão

`boolean`	1	1	`false`	`true`	N/A
`byte`	8	1	-128	127	N/A
`short`	16	2	-32,768	32,767	N/A
`int`	32	4	-2^31	2^31-1	N/A
`long`	64	8	-2^63	2^63-1	N/A
`float`	32	4	±1.4E-45	±3.4E+38	~7 dígitos
`double`	64	8	±4.9E-324	±1.7E+308	~15 dígitos
`char`	16	2	0	65,535	N/A

Dados em Geral

1. **Endereçamento de Memória**:
    - **Endereços de Memória**: Cada byte na RAM tem um endereço único. O sistema operacional usa esses endereços para acessar dados armazenados na RAM.
2. **Cache de CPU**:
    - **Cache L1, L2, L3**: Memória de alta velocidade usada pela CPU para armazenar dados frequentemente acessados. O cache L1 é o mais rápido e menor, enquanto o cache L3 é maior e mais lento.
3. **Memória Virtual**:
    - **Memória Virtual**: Técnica usada pelo sistema operacional para usar parte do disco rígido como se fosse RAM. Isso permite que o sistema execute mais programas do que a RAM física permitiria.
4. **Memória Flash**:

    - **Memória Flash**: Tipo de memória não volátil usada em SSDs, pen drives e cartões de memória. Ela mantém os dados mesmo quando o dispositivo está desligado.

Conversão de Bytes para Binário

- **Relação entre bytes e bits**: 1 byte = 8 bits
- **Cálculo do número total de bits**: 8 bytes = 8 * 8 bits = 64 bits

Representação em binário: Sequência de 64 bits, por exemplo:
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

Memória Cache L1, L2 e L3

A memória cache é uma memória de alta velocidade que serve como intermediária entre a CPU e a RAM. Ela armazena temporariamente dados e instruções frequentemente usados pela CPU, melhorando o desempenho do sistema. A memória cache é dividida em três níveis:

1. **Cache L1**:
    - **Localização**: Dentro do núcleo da CPU.
    - **Velocidade**: A mais rápida entre os três níveis.
    - **Tamanho**: Menor capacidade, geralmente entre 16KB e 64KB por núcleo.
    - **Função**: Armazenar dados e instruções mais frequentemente acessados.
2. **Cache L2**:
    - **Localização**: Pode estar dentro do núcleo da CPU ou ser compartilhada entre núcleos.
    - **Velocidade**: Mais lenta que a L1, mas mais rápida que a L3.
    - **Tamanho**: Maior capacidade que a L1, geralmente entre 256KB e 1MB por núcleo.
    - **Função**: Armazenar dados e instruções que não cabem na L1.
3. **Cache L3**:
    - **Localização**: Fora dos núcleos da CPU, compartilhada entre todos os núcleos.
    - **Velocidade**: A mais lenta entre os três níveis, mas ainda mais rápida que a RAM.
    - **Tamanho**: Maior capacidade, geralmente entre 2MB e 32MB.
    - **Função**: Armazenar dados e instruções que não cabem na L2.

Memória Virtual

A memória virtual é uma técnica de gerenciamento de memória que usa uma parte do disco rígido como se fosse RAM. Isso permite que o sistema operacional execute mais programas do que a RAM física permitiria. Aqui estão alguns pontos importantes:

- **Funcionamento**: Quando a RAM está cheia, o sistema operacional move dados menos usados para um espaço no disco rígido chamado arquivo de paginação ou swap.

Quando esses dados são necessários novamente, eles são movidos de volta para a RAM.
- **Benefícios**: Permite a execução de mais programas simultaneamente, melhora a multitarefa e aumenta a segurança dos dados.
- **Desvantagens**: O acesso ao disco rígido é muito mais lento que o acesso à RAM, o que pode causar perda de desempenho.

Memória Flash

A memória flash é um tipo de memória não volátil usada para armazenamento de dados. Ela mantém os dados armazenados mesmo quando a energia é desligada. Aqui estão algumas características:

- **Tipos**: Existem dois tipos principais de memória flash, NAND e NOR.
    - **NAND**: Usada em SSDs, cartões de memória e pendrives. É mais rápida para operações de escrita e leitura sequencial.
    - **NOR**: Usada em firmware e BIOS. Permite acesso aleatório rápido, mas é mais lenta para operações de escrita.
- **Vantagens**: Não volátil, rápida, silenciosa, resistente e portátil.
- **Desvantagens**: Tem um número finito de ciclos de escrita e apagamento, o que pode limitar sua vida útil.

Memória Stack e Heap

Stack

- **A Memória Stack** é usada para armazenar variáveis locais e chamadas de métodos.
- A memória é gerenciada automaticamente pelo compilador.
- As variáveis armazenadas na stack têm um tempo de vida curto, pois são destruídas assim que o método em que foram declaradas termina.
- Acesso à memória é rápido.

Heap

- **A Memória Heap** é usada para armazenar objetos e variáveis de instância.
- A memória é gerenciada pelo Garbage Collector.
- As variáveis armazenadas na heap têm um tempo de vida mais longo, pois permanecem na memória até que não haja mais referências a elas.
- Acesso à memória é mais lento em comparação com a stack.

String Pool e Memória Heap

- **String Pool**: É uma área especial na memória heap onde Strings literais são armazenadas. Quando você declara uma String literal como `String nomeUm = "Daniel";`, o compilador verifica se uma String com o mesmo valor já existe no String

Pool. Se existir, a nova variável referencia o mesmo objeto. Se não existir, uma nova String será criada no String Pool.
- **Memória Heap**: Quando você usa o operador `new` para criar uma String, como em `String nomeDois = new String("Daniel");`, um novo objeto String é criado na memória heap, independentemente de uma String com o mesmo valor já existir no String Pool.

Memória Stack

- **Memória Stack**: É usada para armazenar variáveis locais e chamadas de métodos. As variáveis locais são armazenadas na stack, mas as Strings que elas referenciam podem estar na heap (se criadas com `new`) ou no String Pool (se literais).

Portanto, o **String Pool não faz parte da memória Stack**. Ele é uma área especial dentro da memória Heap. A diferença nas referências de memória ocorre porque:

- `String nomeUm = "Daniel";` usa o String Pool na memória Heap.
- `String nomeDois = new String("Daniel");` cria um novo objeto na memória Heap.

Comparação de Strings (JAVA)

Quando você usa `==` para comparar Strings, você está comparando as referências de memória, não o conteúdo. Por isso, `nomeUm == nomeDois` retorna `false`, pois as referências de memória são diferentes. Para comparar o conteúdo das Strings, use `equals()`.

Quando você declara String nomeUm = "Daniel";, a variável nomeUm em si está na memória Stack, mas o valor "Daniel" que ela referencia está no String Pool, que é uma área especial dentro da memória Heap.

Por outro lado, quando você declara String nomeDois = new String("Daniel");, a variável nomeDois também está na memória Stack, mas o valor "Daniel" que ela referencia é um novo objeto criado na memória Heap, fora do String Pool.

Portanto, mesmo que nomeUm e nomeDois contenham o mesmo valor "Daniel", suas referências de memória são diferentes:

- nomeUm referência o valor no String Pool da memória Heap.
- nomeDois referência um novo objeto na memória Heap.

É por isso que nomeUm == nomeDois retorna false, mas nomeUm.equals(nomeDois) retorna true, pois equals() compara o conteúdo das Strings, enquanto == compara as referências de memória.

Memória Heap e Tipos Primitivos

1. **Objetos na Heap**:
    - Quando você armazena um objeto na memória heap e modifica esse objeto, todas as variáveis que referenciam esse objeto terão seus valores modificados, pois todas apontam para o mesmo endereço de memória.
2. **Tipos Primitivos**:
    - Tipos primitivos (como `int`, `float`, `boolean`, `double`, etc.) são armazenados na stack. Quando você atribui um valor primitivo a outra variável, você está copiando o valor, não a referência. Portanto, modificar uma variável primitiva não afeta a outra.

String Pool em Java

O String Pool é uma área especial na memória heap onde as strings literais são armazenadas. Aqui estão alguns pontos importantes sobre o String Pool:

1. **Imutabilidade das Strings**:
    - Strings em Java são imutáveis. Uma vez criada, uma string não pode ser modificada. Qualquer operação que pareça modificar uma string na verdade cria uma nova string.
2. **Internamento de Strings**:
    - Quando você cria uma string literal, o Java verifica se essa string já existe no String Pool. Se existir, ele retorna a referência para a string existente. Se não existir, ele adiciona a nova string ao pool. Exemplo:

      ```
      String str1 = "Hello";

      String str2 = "Hello";

      System.out.println(str1 == str2); // Output: TRUE
      ```

    - Nesse exemplo, str1 e str2 referenciam a mesma string no String Pool.
3. **Strings Criadas com `new`**:
    - Quando você cria uma string usando o operador `new`, um novo objeto string é criado na heap, fora do String Pool. Exemplo:

      ```
      String str1 = new String("Hello");

      String str2 = new String("Hello");

      System.out.println(str1 == str2); // Output: FALSE
      ```

    - Nesse caso, `str1` e `str2` são referências a diferentes objetos string na heap. Não compartilham a mesma referência, mesmo que tenham o mesmo valor os objetos String.

Estruturas de Dados Encadeadas em Nós

1. **Nós**:
    - Um nó é uma unidade básica de uma estrutura de dados, como uma lista encadeada.
    - Cada nó contém um valor (ou dados) e uma referência (ou ponteiro) para o próximo nó na sequência.
2. **Encadeamento**:
    - O encadeamento é o processo de conectar nós em uma sequência.
    - Em uma lista encadeada, cada nó aponta para o próximo nó, formando uma cadeia.
    - Existem diferentes tipos de listas encadeadas:
        - **Lista Encadeada Simples**: Cada nó aponta para o próximo nó.
        - **Lista Duplamente Encadeada**: Cada nó aponta para o próximo e o nó anterior.
        - **Lista Circular**: O último nó aponta de volta para o primeiro nó, formando um ciclo.
3. **Exemplo de Lista Encadeada Simples**

```
class No {
    int valor;
    No proximo;

    No(int valor) {
        this.valor = valor;
        this.proximo = null;
    }
}

public class ListaEncadeada {
    public static void main(String[] args) {
        No no1 = new No(1);
        No no2 = new No(2);
        No no3 = new No(3);

        no1.proximo = no2;
        no2.proximo = no3;

        System.out.println(no1.proximo.proximo.valor); // Output: 3
    }
}
```

4. **Encadeamento de Nós**:
    - Em estruturas de dados como listas encadeadas, cada nó contém um valor e uma referência (ou ponteiro) para o próximo nó na sequência.

    - Nesse exemplo:, no1.proximo.proximo retornaria o valor do terceiro nó, que é 3, porque no1 aponta para no2, e no2 aponta para no3.
5. **Diferença entre Java e C**:
    - **C**: Usa ponteiros explicitamente para referenciar endereços de memória. Você precisa gerenciar manualmente os ponteiros para acessar os próximos nós.
    - **Java**: Usa referências de objetos. Em Java, a referência ao próximo nó é gerenciada automaticamente pelo sistema de gerenciamento de memória. Você não lida diretamente com endereços de memória, mas com referências a objetos.
6. **Referência de Nó semelhante a referência de objetos**:
    - Similar ao comportamento de referências de objetos, onde cada nó contém uma referência ao próximo nó. Modificar um nó afeta todos os nós que referenciam este nó.