Aula 9 - Analise de Complexidade e Estrutura de Dados

Docupedia Export

Author:Silio Leonardo (SO/OPM-TS21-BR)
Date:10-May-2024 16:59

Table of Contents

Já aprendemos sobre linked lists e vetores dinâmicos. Também já implementamos uma Stack e uma Queue. Nesta aula vamos ver as última estrutura de dados útil e aprender a usar as estruturas de dados prontas do Java. O conhecimento de estrutura de dados e seus tempos de execução são fundamentais para construção de bons algoritmos. Vamos compreender o que é análise de complexidade, de forma resumida, já que este conteúdo é extra e geralmente é aplicado para se aprender em dezenas de horas de estudo, vamos aprender o resumo em bem menos de 2h. Então vamos lá.

1 Análise de Complexidade Computacional

Análise de Complexidade Computacional é basicamente o estudo de como algoritmos ficam mais lentos conforme a quantidade de dados processado fica maior. Por exemplo, ao queremos ordenar uma coleção de dados desordenada, é evidente que o tempo será maior conforme mais dados temos que ordenar mas quanto? Para tornar essa conversa palpável vamos a uma pequena análise simples:

1. Não é necessário existir uma precisão exata de tempo, já que cada sistema computacional pode agir totalmente diferente do outro, além disso, sistemas operacionais e processadores podem ser instáveis.
2. É muito mais interessante compreender o efeito de quanto existem muitos dados (casos críticos) do que quando são poucos dados.
3. Grande parte dos algoritmos crescem muito seu tempo de execução quando os dados de entrada aumentam, isso significa que proporcionalidades (algoritmo A é sempre duas vezes mais rápido que o algoritmo B) são supreendentemente desprezíveis perante a quanto um algoritmo pode piorar (algoritmo A e B são 300 vezes mais lentos que C quando processamos 1000 dados).
4. Como pode-se perceber, aqui precisamos analisar uma função, a saída é algo proporcional ao tempo e a entrada um número que representas a quantidade de dados inseridos, para representar cada algoritmo.

Dito isso tempo a notação Big-O:

$$\mathrm{f(n)} \in \mathrm{O(g(n))} \iff \exists \mathrm{C} \lim_{n\to\infty} \frac{f(n)}{g(n)} < \mathrm{C}$$

Uma definição equivalente é:

$$\mathrm{f(n)} \in \mathrm{O(g(n))} \iff \exists \mathrm{M}, \mathrm{n_0} \forall \mathrm{n} > \mathrm{n_0} (\mathrm{f(n)} < \mathrm{Mg(n)})$$

Isso pode aparecer muito assustador em um primeiro minuto, mas pode ser muito simples. Vamos pensar na função $\mathrm{f(n)} = 6\mathrm{n}^2 + \mathrm{n} - 3$, e pense em $\mathrm{g(n)} = \mathrm{n}^2$, então queremos analisar se existem $\mathrm{n_0}$ onde todos os valores a partir deste $\mathrm{g(n)}$ domina $\mathrm{f(n)}$. Note que a impressão que temos é que $\mathrm{f(n)}$ sempre maior que $\mathrm{g(n)}$, mas se nosso M for algum valor como 7, o oposto ocorre. Por limites seria a seguinte análise:

$$\lim_{n\to\infty} \frac{6n^2 + n - 2}{n^2} = \lim_{n\to\infty} 6 + \frac{1}{n} - \frac{2}{n^2} = 6 < \mathrm{C} = 7$$ (quando n vai ao infinito, algo dividido por infinito vai a zero)

Na prática esse é o pensamento que teremos: É possível provar que:

$$\mathrm{a_0\, n^0 + a_1 n^1 + \ldots + a_k n^k} \in \mathrm{O(n^k)}$$

E isso é totalmente de acordo com nossos primeiros pensamentos: Se um algoritmo demora $5\,\mathrm{n}^2 + \mathrm{n}$ milissegundos para executar, quando temos 1000 dados de entrada teríamos 5 milhões por parte do termo a esquerda e só 1000 por parte do termo a direita, isso faz esse algoritmo ser $\mathrm{O(n^2)}$. Pois o n^2 é o que realmente importa em toda a análise.

Para calcular a ordem de complexidade basta "contar" as linhas que vão de fato ser proporcionais ao tempo de execução. Vamos a um bom exemplo para que isso seja de fato compreendido: O seguinte algoritmo é uma implementação de um Bubble Sort:

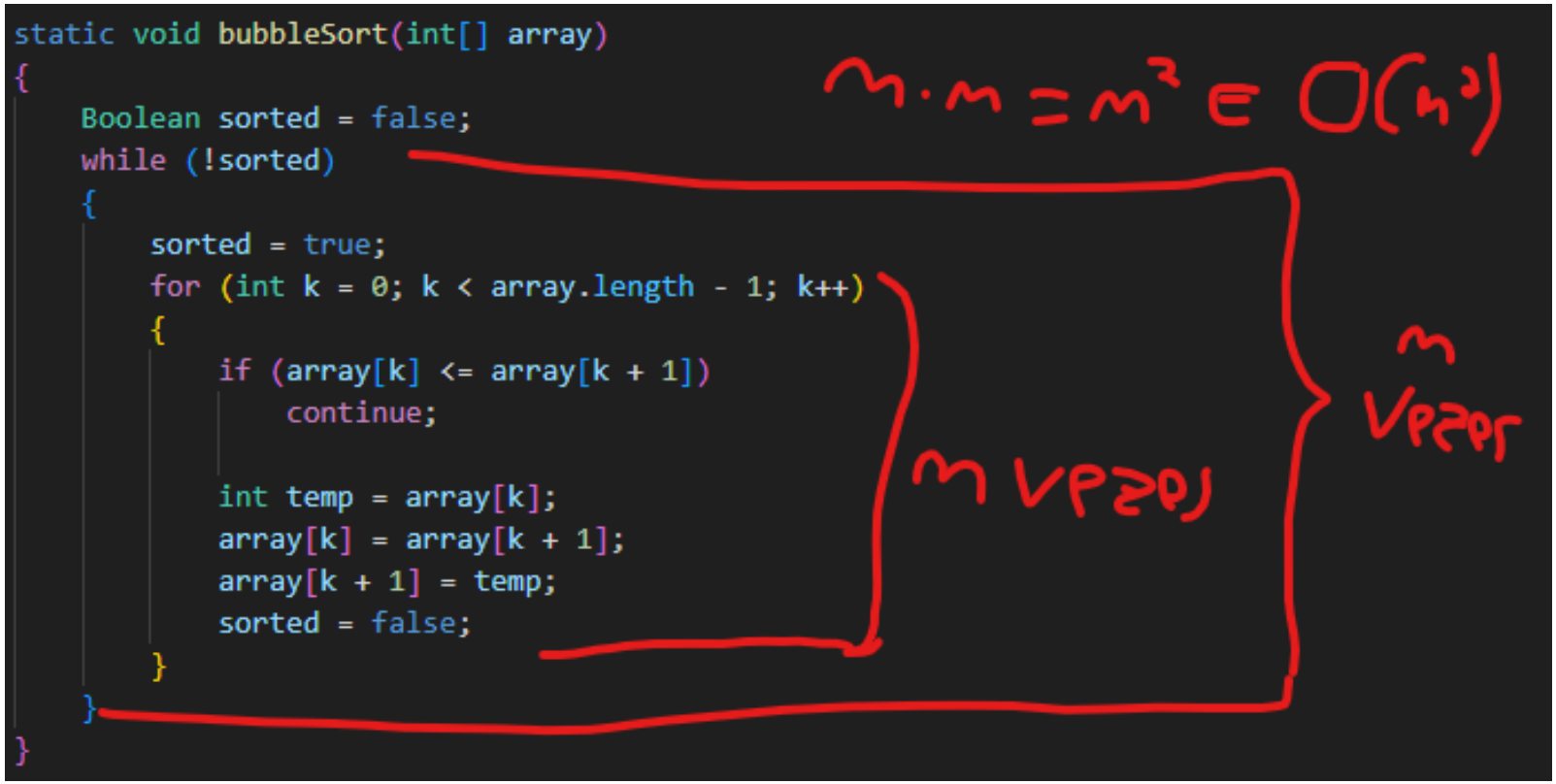
```
1  import java.util.Random;
2
3  class Main
4  {
5      public static void main(String[] args)
6      {
```

```java
        for (int size = 5000; size <= 80000; size *= 2)
        {
            long duration = testTime(size);
            System.out.println("n = " + size + "; t =" + duration + " milliseconds");
        }
    }

    static long testTime(int size)
    {
        int[] array = generateRandomArray(size);

        long startTime = System.currentTimeMillis();
        bubbleSort(array);
        long endTime = System.currentTimeMillis();

        long duration = endTime - startTime;
        return duration;
    }

    static void bubbleSort(int[] array)
    {
        Boolean sorted = false;
        while (!sorted)
        {
            sorted = true;
            for (int k = 0; k < array.length - 1; k++)
            {
                if (array[k] <= array[k + 1])
                    continue;

                int temp = array[k];
                array[k] = array[k + 1];
                array[k + 1] = temp;
                sorted = false;
            }
        }
    }

    static int[] generateRandomArray(int size)
```

```
46      {
47          int[] array = new int[size];
48          Random random = new Random();
49          for (int i = 0; i < size; i++) {
50              array[i] = random.nextInt(1000);
51          }
52          return array;
53      }
54  }
```

O resultado obtido foi:

```
n = 5000; t =20 milliseconds
n = 10000; t =73 milliseconds
n = 20000; t =365 milliseconds
n = 40000; t =2351 milliseconds
n = 80000; t =10132 milliseconds
```



Como temos um loop dentro do outro que roda cada uma n vezes, ou seja para $n = 100$ rodamos o código 100*100=10'000 (10 mil) vezes. Note que o interior do loop impacta na duração, mas ainda sim, o mais relevante que muda a ordem é o n^2. Observe que conforme dobramos o tamanho da entrada, temos uma boa consistência em quadruplicar ($(2n)^2 = 4n^2$) o tempo.

Analisando os nossos algoritmos anteriores podemos perceber que:

- A lista ligada tem inserção de $O(1)$, ou seja, instantâneo, mas somente se ele tem um ponteiro para o final/inicio da lista.
- O array dinâmico tem inserção $O(1)$, na média, mas algumas vezes tem $O(n)$.
- Um problema é a remoção no array dinâmico que envolve descolocar todos os dados no vetor para primeira posição, sendo assim: $O(n)$
- Todas as remoções em lista ligadas são $O(1)$, desde que existam ponteiros para inicio ou fim da lista.
- Buscar dados por índice no array dinâmico é $O(1)$ e na lista ligada é $O(n)$.

2 Hash

Um Hash é basicamente um armazenamento por index. Isso faz com que guardemos cada valor em um índice específico e acessemos por ele. A diferença é que para economizar espaço vamos usar uma função de Hash, isso significa que vamos mapear cada valor do de entrada em um índice específico. Isso nos permitira guardar e acessar valores em $O(1)$, ou seja, independe de quantos dados temos na estrutura o tempo é sempre o mesmo, o que é muito rápido. Uma função de Hash é uma função que mapeia os valores em um valor inteiro. Vamos supor que você queira mapear valores String em um Hash, podemos converter o primeiro carácter ASCII para um inteiro e guardar neste índice. Um número inteiro pode ser mapeado para ele mesmo (que daria um vetor muito grande) ou até melhor: Para ele em módulo de 100. Isso significa que com um vetor de 100 posições podemos ir guardando vários valores inteiros. o 75 vai na casa 75 do vetor e o 176 vai na casa 76.

A pergunta é: E se dois valores tiverem o mesmo resultado (mesmo resultado da Hash Function)?
A resposta é simples: Isso se chama colisão é resolvida com cada casa do vetor sendo uma lista ligada ou array dinâmico que vamos chamar de bucket. Toda vez que adicionamos um valor colocamos ele na lista da sua casa no final da lista. Por exemplo, no caso de adicionar 176 e 276, teríamos uma lista com esses valores na casa 76. Quando eu pergunto ao meu HashSet se eu possuo ou não o valor 176, ele vai na casa 76 e busca linearmente na lista para ver se encontra o valor. Note que isso pode fazer um Hash ser muito ineficiente se os valores armazenados sempre colidirem, fazendo as operações serem $O(n)$. Podemos colocar outras estruturas de dados mais eficientes se desejarmos tornando o Hash ainda mais poderoso e rápido.

Outra pergunta: Isso não significa que o Hash vai ficando pior ao longo do tempo?
A resposta é: Colocamos uma fator máximo de preenchimento que chamaremos de load factor. Quando preencher essa porcentagem (digamos 70%) da tabela vamos recriar o HashSet internamente aumentando a quantidade de casas que temos. Por exemplo, podemos dobrar a quantidade da casas para 200 e agora o 176 que antes ficava na casa 76 vai para casa 176 e apenas o 276 compartilha o bucket com o 76. Isso aumenta e eficiência embora exigirá uma operação $O(n)$ uma vez a cada muitas inserções.
Note que o Hash é horrível para Busca de fato, lento todos os valores da estrutura, mas ele é especialmente bom para encontrar um valor a partir da sua chave, sendo interessante para conjuntos onde precisamos dizer se algo está ou não dentro dos dados e dicionários.
Cada implementação do Hash define função de hash padrão, load factor, tamanho inicial de buckets padrão e estrutura de dados interna para usar nos buckets.

3 Fila de Prioridade

Fila de prioridade é uma fila que mantém os dados sempre ordenados. Isso faz com que cada inserção possa ser mais lenta (dependendo da estrutura de dados interna usada para representar a Fila) mas tendo a remoção do menor elemento (ou maior) de forma muito rápida, em $O(1)$. Os elementos ainda podem ter uma chave como prioridade e um valor que se usará como um dicionário ordenado. Sendo uma fila, as operações de Enqueue, que coloca um valor, desta vez não no final mas na ordem de acordo com a prioridade, e Dequeue removendo a maior/menor prioridade estão presentes e são as principais.
É possível implementar uma fila de prioridade facilmente usando uma lista encadeada e procurando onde inserir o valor desejado.

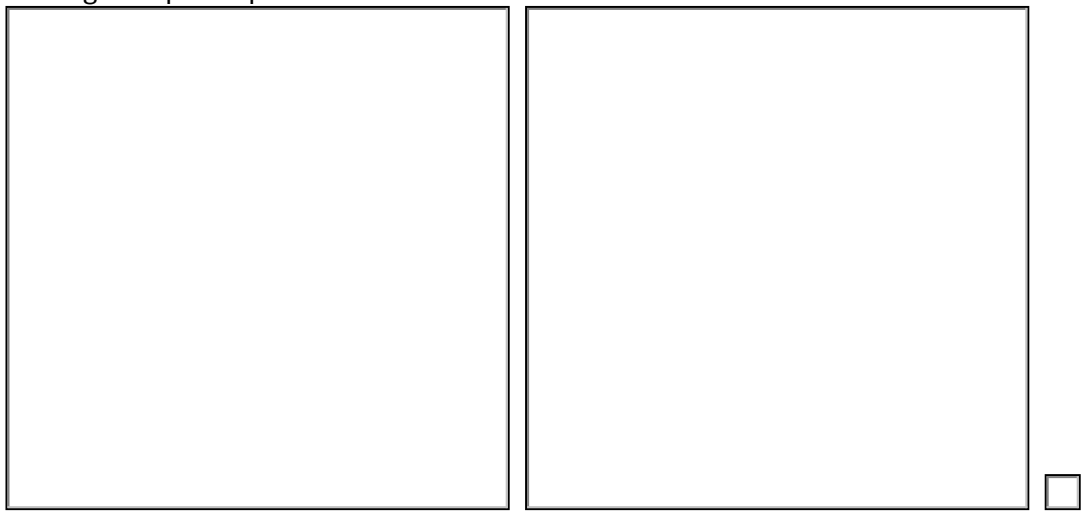
4 Árvores Binárias

Outra estrutura interessante é a Árvore Binárias, que é na verdade, o mais utilizado para implementar os buckets de um Hash e também para implementar Filas de Prioridade. Essa é a única estrutura, em teoria, não linear que veremos. Ela é não linear porquê é representado usualmente como uma árvore binária. Isso significa que cada nó tem um valor e pode ter até, no máximo, dois nós filhos. Além disso, um heap que pode ser maximal ou mínima, tem o valor mais extremo, digamos o maior, na raiz (inicio) da árvore. (Por isso é super interessante para filas de prioridade). Árvores binárias de busca (regras um pouco diferentes) podem ser muito rápidas para busca (por isso ótimas para Hash), onde um valor intermediário fica no meio e os valores a esquerda sempre são menores e a direita maiores.
A forma clássica de representar árvores binárias é com um vetor, onde o valor na casa k tem filhos nas casas $2 * k + 1$ e $2 * k + 2$, podendo usar uma lista de array dinâmico para guardar os valores da sua árvore.

Existem vários tipos de Heap e cada um tem tempos e implementações diferentes:

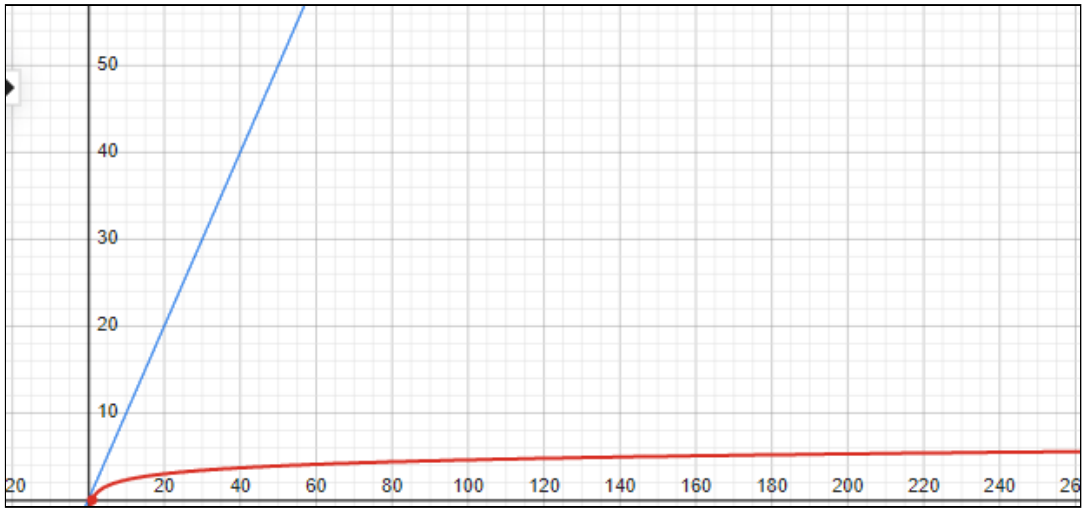
Operation	find-min	delete-min	insert	decrease-key	meld
Binary	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Leftist	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binomial	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Skew binomial	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Pairing	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
Rank-pairing	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
Fibonacci	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
Strict Fibonacci	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
Brodal	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(1)$
2–3 heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	*?*

Observe o processo de inserção em um Heap Binário que mantem as qualidades de um Heap. Aqui iremos inserir o valor 15 no final do vetor o que vai necessitar um rearranjo dos dados, onde o 15 sobe até chegar ao ponto que deveria estar:



Agora um exemplo removendo o maior valor, onde trocamos ele com valores extremos e movemos o valor errado ao lugar certo:

Uma busca em um Heap demora, por isso uma árvore binária é mais recomendada para um Hash. Uma árvore binária leva tempo $O(\ln(n))$ na busca que é muito melhor que $O(n)$:



5 Em Java

- Array List: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Linked List e Queue: https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
- Hash Set: https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html
- Hash Map: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html
- Stack: https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html
- Priority Queue: https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
- Geralmente, árvores são implementadas pelo usuário.