Análise de Complexidade Temporal

Curso: Licenciatura em Engenharia Informática
Cadeira: Algoritmos e Estruturas de Dados
Alunos: Bernardo Freitas - 88612, Francisco Neves – 67585

Grupo: 50

Ano Letivo: 2024/2025

Introdução

O relatório tem como finalidade analisar de forma aprofundada a complexidade temporal da estrutura de dados FintList, uma implementação personalizada de lista duplamente ligada baseada em arrays paralelos. A FintList foi concebida como uma alternativa eficiente às estruturas nativas do Java, nomeadamente a LinkedList e o ArrayList, combinando as vantagens de ambos os paradigmas: acesso por índice e gestão dinâmica de elementos. Este estudo visa compreender o comportamento temporal tanto do ponto de vista teórico como empírico. Será também estabelecida uma comparação direta com a LinkedList do Java fornecido pelo dirigente da cadeira, professor João Dias, de modo a evidenciar as diferenças de desempenho e de implementação.

Aproximação tilde get(int index)

FintList:

Função get(int index), (67585 % 4 = 1 logo é o get());

```
public int get(int index)  no usages
{
        if(index < 0 || index >= this.size) throw new IndexOutOfBoundsException("Falhou no Get");

    int current;

    if (index < this.size / 2)
    {
        current = this.head;
        for (int i = 0; i < index; i++)
        {
            current = this.data_dir[current];
        }
    }
    else
    {
        current = this.tail;
        for (int i = this.size - 1; i > index; i--)
        {
            current = this.data_esq[current];
        }
    }

    return this.data[current];
}
```

Fig. – 1 Descrição da função Get do nosso código

Melhor caso: ~(1) quando o índice é igual a 0 ou size – 1 (extremidades).

Pior caso: ~(N/2) ocorre quando o índice que se pretende aceder se encontra no meio da estrutura de dados.

LinkedList:

Função get(int index):

```
public T get(int index) {  no usages
    Node n = this.first;
    while(index != 0)
    {
        n = n.next;
        index--;
    }

    return n.item;
}
```

Fig.2– demonstração da função Get do código fornecido pelo professor.

Melhor caso: ~(1) quando o índice é igual a 0

Pior caso: ~(n) quando o índice é n-1

3. Testes Empíricos FintList vs LinkedList:

AddAt():

```java
public static void main(String[] args) {
    System.out.println("ADD AT MINHA LISTA");
    TemporalAnalysisUtils.runDoublingRatioTest(
            (Integer n) -> {
                FintList list = new FintList();
                for (int i = 0; i < n; i++) {
                    list.add(i);
                }
                return list;
            },
            (FintList list) -> {
                for (int i = 0; i < list.size(); i += 5) {
                    list.addAt(i, i);
                }
            },
            iterations: 9);
```

Fig.3 - Demonstração do código utilizado para obter os resultados utilizados na comparação.

```
ADD AT MINHA LISTA
i   complexity  time(ms)    estimated r
0   125 0.0  ---
1   250 0.0 0.0
2   500 0.0 0.0
3   1000    0.520833    0.0
4   2000    0.520833    1.0
5   4000    1.5625  3.0000019200012287
6   8000    6.25    4.0
7   16000   26.041666   4.16666656
8   32000   105.729166  4.060000078336002
9   64000   430.208333  4.068965539745202
```

Fig.4 – demonstração do nosso caso que tem como razão dobrada o valor 2^2, logo complexidade $O(n^2)$

```java
System.out.println("ADD AT LISTA PROF");
TemporalAnalysisUtils.runDoublingRatioTest(
        (Integer n) -> {
            LinkedList<Integer> list = new LinkedList<>();
            for (int i = 0; i < n; i++) {
                list.add(i);
            }
            return list;
        },
        (LinkedList<Integer> list) -> {
            for (int i = 0; i < list.size(); i += 5) {
                list.addAt(i, i);
            }
        },
        iterations: 9);
```

Fig.5 - *Demonstração do código fornecido pelo professor para efeito de comparação.*

```
ADD AT LISTA PROF
i   complexity  time(ms)    estimated r
0   125 0.0  ---
1   250 0.0 0.0
2   500 0.520833    0.0
3   1000    0.520833    1.0
4   2000    1.041666    2.0
5   4000    5.208333    5.000002880001843
6   8000    21.875  4.200000268800017
7   16000   91.145833   4.166666651428572
8   32000   376.5625    4.131428586537796
9   64000   1570.3125   4.170124481327801
```

Fig.6 – demonstração do caso resultado do código fornecido pelo professor, que tem como razão dobrada o valor 2^2, logo complexidade $O(n^2)$

RemoveAt():

```
System.out.println("REMOVE AT MINHA LISTA");
TemporalAnalysisUtils.runDoublingRatioTest(
        (Integer n) -> {
            FintList list = new FintList();
            for (int i = 0; i < n; i++) {
                list.add(i);
            }
            return list;
        },
        (FintList list) -> {
            for (int i = list.size()-1; i >= 0; i -= 5) {
                list.removeAt(i);
            }
        },
        iterations: 9);
```

Fig.7 - Demonstração do código utilizado para obter os resultados utilizados na comparação

```
REMOVE AT MINHA LISTA
i   complexity  time(ms)    estimated r
0   125 0.0  ---
1   250 0.0 0.0
2   500 0.0 0.0
3   1000    0.0 0.0
4   2000    0.0 0.0
5   4000    1.5625  0.0
6   8000    4.166666    2.66666624
7   16000   17.1875 4.125000660000105
8   32000   67.708333   3.93939392
9   64000   273.4375    4.0384615583431955
```

Fig.8 – demonstração do nosso caso que tem como razão dobrada o valor 2^2, logo complexidade $O(n^2)$

```
System.out.println("REMOVE AT LISTA PROF");
TemporalAnalysisUtils.runDoublingRatioTest(
        (Integer n) -> {
            LinkedList<Integer> list = new LinkedList<>();
            for (int i = 0; i < n; i++) {
                list.add(i);
            }
            return list;
        },
        (LinkedList<Integer> list) -> {
            for (int i = list.size()-1; i >= 0; i -= 5) {
                list.removeAt(i);
            }
        },
        iterations: 9);
```

Fig.9 - Demonstração do código utilizado para obter os resultados utilizados na comparação

```
REMOVE AT LISTA PROF
i   complexity  time(ms)    estimated r
0   125 0.0  ---
1   250 0.0 0.0
2   500 0.520833    0.0
3   1000    0.0 0.0
4   2000    0.520833    0.0
5   4000    3.125   6.000003840002457
6   8000    10.416666   3.33333312
7   16000   47.916666   4.600000230400015
8   32000   193.229166  4.032608737844991
9   64000   854.166666  4.420485187003291
```

Fig. 10– demonstração do caso resultado do código fornecido pelo professor, que tem como razão dobrada o valor 2^2, logo complexidade $O(n^2)$

DeepCopy e ShallowCopy:

```java
System.out.println("DEEPCOPY MINHA LISTA");
TemporalAnalysisUtils.runDoublingRatioTest(
        (Integer n) -> {
            FintList list = new FintList();
            for (int i = 0; i < n; i++) {
                list.add(i);
            }
            return list;
        },
        (FintList list) -> {
            for (int i = 0; i < list.size(); i += 10) {
                list.deepCopy();
            }
        },
        iterations: 9);
```

Fig.11 - Demonstração do código utilizado para obter os resultados utilizados na comparação

```
DEEPCOPY MINHA LISTA
i   complexity  time(ms)    estimated r
0   125 0.0  ---
1   250 0.0 0.0
2   500 0.0 0.0
3   1000    0.0 0.0
4   2000    1.041666    0.0
5   4000    3.645833    3.5000019200012287
6   8000    13.541666   3.7142858710204223
7   16000   34.895833   2.5769231791716027
8   32000   126.5625    3.626865706286478
9   64000   510.9375    4.037037037037037
```

Fig.12 – demonstração do caso resultado do código fornecido pelo professor, que tem como razão dobrada o valor 2^2, logo complexidade

```
SHALLOWCOPY LISTA PROF
i   complexity  time(ms)    estimated r
0   125 0.0  ---
1   250 0.0 0.0
2   500 0.520833    0.0
3   1000    1.5625  3.0000019200012287
4   2000    5.729166    3.66666624
5   4000    23.4375 4.090909566942204
6   8000    100.0   4.266666666666667
7   16000   400.0   4.0
Exception in thread "main" java.lang.StackOverflowError Create breakpoint
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
    at aed.collections.LinkedList$Node.shallowCopy(LinkedList.java:16)
```

Fig.13 - Demonstração do código utilizado para obter os resultados utilizados na comparação

```java
        System.out.println("SHALLOWCOPY LISTA PROF");
        TemporalAnalysisUtils.runDoublingRatioTest(
                (Integer n) -> {
                    LinkedList<Integer> list = new LinkedList<>();
                    for (int i = 0; i < n; i++) {
                        list.add(i);
                    }
                    return list;
                },
                (LinkedList<Integer> list) -> {
                    for (int i = 0; i < list.size(); i += 5) {
                        list.shallowCopy();
                    }
                },
                iterations: 9);
    }
}
```

Fig.14 – demonstração do caso resultado do código fornecido pelo professor, que tem como razão dobrada o valor 2^2, logo complexidade

4. Comparação de resultados:

- AddAt():

  O nosso código da complexidade de O(n^2), tal como o código do professor, no entanto, o nosso código acaba por atingir uma velocidade de execução superior ao do professor, pois as listas ligadas de inteiros são mais eficientes para este tipo de casos do que os Arrays de Nós.
  Tendo em consideração que o nosso código não passou no teste de velocidade pretendido no Mooshak, concluímos que o nosso método não está tao otimizado como poderia estar.

- RemoveAt();

  À semelhança do que acontece no AddAt(), O nosso código da complexidade de O(n^2), tal como o código do professor, acabando por alcançar uma velocidade de execução superior.
  Tal como referido no método anterior, o nosso código não passou no teste de velocidade do Mooshak, continuamos a achar que não está otimizado o suficiente

- DeepCopy e ShallowCopy

  Como o ShallowCopy faz uma cópia superficial da lista mantendo as mesmas referências, e a DeepCopy cria novos objetos, tendo assim novas referências, verifica-se que são métodos com implementações bastante distintas e por tanto a sua comparação pode ser um pouco ambígua.

  Tendo isto em consideração, o facto de nos dar um erro no ShallowCopy, considera-se justificado.

5. Vantagens de usar FintLists:

A utilização de uma FintList (lista duplamente ligada de inteiros) é mais apropriada do que um vetor em situações onde há muitas inserções e remoções em posições arbitrárias da lista, como na gestão de filas dinâmicas ou de um histórico de navegação.
Nestes casos, a FintList evita o custo de deslocar elementos no array e permite realizar essas operações em tempo O(1), enquanto um vetor exigiria O(n) devido à necessidade de mover todos os elementos subsequentes na memória.