Algoritmos e estruturas de dados

Análise de complexidade temporal

Alexandre Guerreiro, n.º88489; Léo Souza, n.º90275.

1)

Escolha do método:

Número de aluno mais pequeno = 88489 % 4 = 1 = get(int index);

Melhor caso

Vamos considerar n = a quantidade de elementos na FintList = size. A função getNodeINdex(int index) possuí uma otimização importante, que a permite lembrar qual foi o último elemento acessado através das variáveis *LastArrayPosition* e *lastUsedNode*. Dessa forma, o melhor caso é quando o get(int index) busca a mesma variável usada anteriormente, pois getNodeIndex(int index) retorna nesta linha:

```
        if (index == lastUsedNode && lastArrayPosition != -1)
            return lastArrayPosition;
```

Tabela melhor caso

Instrução	Frequência	Notação Tilde
Declaração de variáveis	0	~0
Atribuição	1	~1
Comparação == ou !=	1	~2
Comparação <>	3	~3
Acesso lista	1	~1
Incremento	0	0

Pior caso

A função getNodeINdex(int index), além de lembrar o último nó acessado, também pode buscar um nó a partir do início ou do fim, escolhendo o melhor baseado na distância. Dessa forma, o pior caso é quando o get(int index) procura um index na posição $\frac{n}{2}$ e *lastArrayPosition* = -1 (não se lembra de nenhum index), pois assim o getNodeIndex(int index) passa por todas as comparações e retorna apenas no final:

```
return atual;
```

Tabela do pior caso

Instrução	Frequência	Notação Tilde
Declaração de variáveis	5	~5
Atribuição	$\frac{n}{2} + 6$	$\sim \frac{n}{2}$
Comparação == ou !=	2	~2
Comparação <>	$\frac{n}{2} + 3$	$\sim \frac{n}{2}$
Acesso lista	$\frac{n}{2}$	$\sim \frac{n}{2}$
Incremento	$\frac{n}{2}$ - 1	$\sim \frac{n}{2}$

2)

Melhor caso

Vamos considerar n = a quantidade de elementos na LinkedList = size. O melhor caso é quando o get(int index) procura o primeiro elemento da lista, pois ele procura da esquerda para direita.

Tabela do melhor caso

Instrução	Frequência	Notação Tilde
Declaração de variáveis	1	~1
Atribuição	1	~1
Comparação == ou !=	1	~1
Incremento	0	~0
Acesso a lista	1	~1
Comparação <>	0	~0

Pior caso

Como dito anteriormente, a lista procura da esquerda para direita, então o pior caso é quando procura um valor na posição n - 1, pois tem de percorrer todos os elementos da lista até essa posição.

Tabela do pior caso

Instrução	Frequência	Notação Tilde
Declaração de variáveis	1	~1
Atribuição	n	~n
Comparação == ou !=	n - 1	~n
Comparação <>	0	~0
Acesso lista	n	$\sim n$
Incremento	n - 1	~n

3)

Observações

Para realizar os testes empíricos, será utilizada a classe *TemporalAnalysisUtils* definida na package aed.collections e disponibilizada pelo docente. Todos os testes estão na classe Main do projeto, devidamente identificados.

Método AddAt

Para testar o método, utilizaremos a simulação de Monte Carlo. Esta técnica matemática estima os possíveis resultados de um evento incerto. No nosso caso, aplicaremos a classe *java.util.Random* para gerar posições aleatórias onde os elementos serão adicionados a uma lista de tamanho n que cresce gradualmente. Os testes serão realizados tanto na FintList desenvolvida quanto na LinkedList da package *aed.collections*.

Ensaios de razão dobrada:

Para estes testes foi usado uma complexidade inicial de n = 10000. Caso contrário, seria necessário horas para terminar os testes.

Resultados:

```
-----------------FintList----------------
i   complexity  time(ms)    estimated r
0   10000   48256.0  ---
1   20000   0.056418    1.1691395888594165
2   40000   0.045474    0.8060193555248325
3   80000   0.130936    2.879359634076615
4   160000  0.210826    1.610145414553675
5   320000  0.317756    1.5071955071955072
6   640000  0.326113    1.0263000541295837
7   1280000 0.615652    1.8878486904845866
8   2560000 1.139882    1.851503771611235
9   5120000 2.156825    1.892147608261206
10  10240000    3.537405    1.640098292629212
```

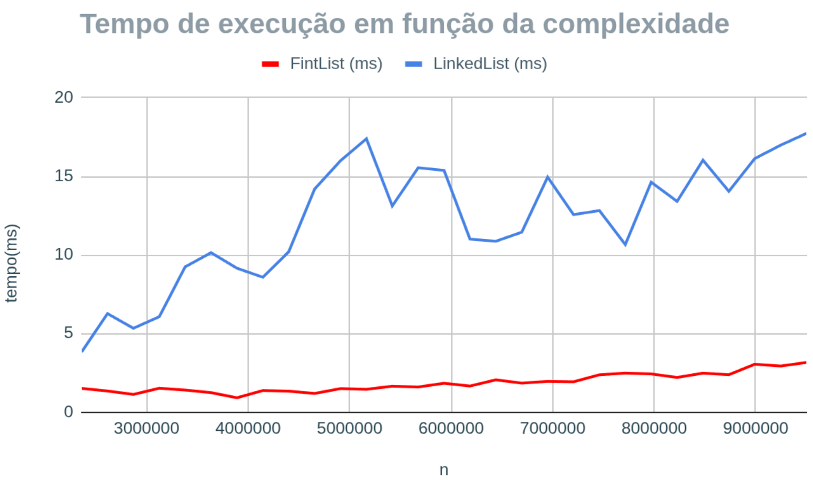
```
---------------LinkedList---------------
i   complexity  time(ms)    estimated r
0   10000   36246.0  ---
1   20000   0.030199    0.833167797825967
2   40000   0.035632    1.1799066194244843
3   80000   0.095585    2.6825606196677145
4   160000  0.213478    2.233383899147355б
5   320000  0.670318    3.139986321775546
6   640000  1.082623    1.6150886594123983
7   1280000 2.158091    1.9933910511784805
8   2560000 4.44126 2.057957704285871
9   5120000 15.572788   3.506389628168583
10  10240000    25.474462   1.6358318112338008
```

Após calcular uma aproximação do r médio para a FintList e para a LinkedList obtivemos o valor 1.62698 e 2.09781 respetivamente.

Através da fórmula $r = 2^b (=) b = \log_2 r$ é possível estimar a complexidade temporal do método addAt para cada classe, uma vez que T(n)~n^b. Assim concluímos que o método addAt na classe FintList tem uma complexidade temporal aproximada de n uma vez que $\log_2(1.62698) \simeq 1$ e na outra uma complexidade temporal aproximada de n dado que $\log_2(2.09781) \simeq 1$.

Ensaio gráfico:

Neste teste, ao invés de duplicar o valor de n a cada iteração, incrementamos por um valor fixo, para isso vamos utilizar uma complexidade inicial = 1850000 e um valor de incremento = 255000. Gráfico para comparação:



Complexidade assimptótica do método:

FintList

```
public void addAt(int index, int item) {
    if (index == size) { -----> O(1)
        add(item); -----> O(1)
        return; -----> O(1)
    }
    if (index < 0 || index > size) { -----> O(1)
        throw new IndexOutOfBoundsException("Índice
invalido");
    }
    if (free_index == -1 && size >= capacity) ----->
O(1)
        grow(capacity << 1);
    int next_free_index = -1; -----> O(1)
    int slot = size; -----> O(1)
    if (free_index != -1) /    { -----> O(1)
        next_free_index = next_index[free_index];
       slot = free_index; -----> O(1)
    }
    int atual; -----> O(1)
    atual = getNodeIndex(index); -----> O(n)
    elements[slot] = item; -----> O(1)
    next_index[slot] = atual; -----> O(1)
    prev_index[slot] = prev_index[atual]; ----->
O(1)
    lastArrayPosition = slot; -----> O(1)
    if (index == 0) { -----> O(1)
        head = slot; -----> O(1)
    } else {
        next_index[prev_index[atual]] = slot; ----->
O(1)
    }
    prev_index[atual] = slot; -----> O(1)
    free_index = next_free_index; -----> O(1)
    size++; -----> O(1)
}
```

LinkedList

```
public void addAt(int index, T item) {
    if (index == 0) { -----> O(1)
        add(item);  -----> O(1)
        return; -----> O(1)
    }
    Node newNode = new Node();  -----> O(1)
    newNode.item = item;  -----> O(1)
    Node n = this.first.next;  -----> O(1)
    Node previous = this.first;  -----> O(1)
    index--;  -----> O(1)
    while (index != 0) {  -----> O(n)x (
        previous = n;  -----> O(1)
        n = n.next;  -----> O(1)
        index--;  -----> O(1)
    } )
    newNode.next = n;  -----> O(1)
    previous.next = newNode;  -----> O(1)  }
```

Tanto a implementação do addAt da FintList e da LinkedList tem uma complexidade de tempo O(n)

Método RemoveAt

Tal como no método anterior, também vamos usufruir da simulação Monte Carlo para remover um elemento de um índice aleatório.

Ensaios de razão dobrada:

```
----------------FintList----------------
i   complexity  time(ms)    estimated r
0   11000   34218.0   ---
1   22000   0.028532    0.8338301478753872
2   44000   0.04357 1.527057339127996 7
3   88000   0.121036    2.777966490704613
4   176000  0.185279    1.5307759674807495
5   352000  0.135184    0.7296239724955338
6   704000  0.228676    1.6915907207953604
7   1408000 0.381371    1.6677351361751998
8   2816000 0.859349    2.253315013464579
9   5632000 1.356294    1.5782807683490643
10  11264000    3.84021 2.8313993868586014
```

```
---------------LinkedList---------------
i   complexity  time(ms)    estimated r
0   11000   36477.0   ---
1   22000   0.026542    0.727636592921567
2   44000   0.038481    1.449815386933916
3   88000   0.090804    2.359709986746706
4   176000  0.336057    3.7009052464649135
5   352000  0.740622    2.2038582740427963
6   704000  1.138439    1.5371390533902585
7   1408000 1.9267  1.692405126669061 8
8   2816000 3.401232    1.7653147869414023
9   5632000 9.620723    2.8285994604308087
10  11264000    19.453824   2.0220750561054506
```

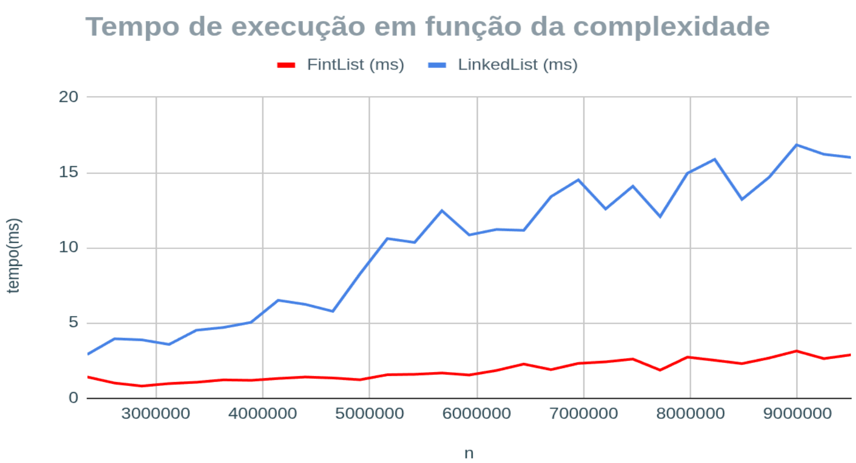
Após calcular uma aproximação do r médio para a FintList e para a LinkedList obtemos o valor 1.74276 e 2.02875 respetivamente.

Assim concluímos que o método removeAt na classe FintList tem uma complexidade temporal aproximada de n uma vez que $log_2(1.74276) \simeq 1$ e na classe LinkedList uma complexidade temporal aproximada de n já que $log_2(2.02875) \simeq 1$

Ensaio gráfico:

Teste com a mesma técnica utilizada para função AddAt, mas como apenas um elemento é eliminado a cada teste, vamos utilizar uma complexidade inicial e um valor de incremento maior. 1850000 e 255000 respectivamente.

Gráfico para comparação:



Complexidade assimptótica do método:

FintList

```
public int removeAt(int index) {
   if (index == size - 1) -----> O(1)
       return remove(); -----> O(1)
   int atual = getNodeIndex(index); -----> O(n)

   int next = next_index[atual]; -----> O(1)
   int prev = prev_index[atual]; -----> O(1)

   lastArrayPosition = next; -----> O(1)
   prev_index[next] = prev; -----> O(1)
   if (index != 0) -----> O(1)
       next_index[prev] = next; -----> O(1)
       head = next; -----> O(1)
    next_index[atual] = free_index; -----> O(1)
   free_index = atual; -----> O(1)
   size--; -----> O(1)
   return elements[free_index]; -----> O(1)
}
```

LinkedList

```
public T removeAt(int index) {
   if (index == 0) { -----> O(1)
       return remove(); -----> O(1)
   }

   Node n = this.first.next; -----> O(1)
   Node previous = this.first; -----> O(1)
   index--;

   while (index != 0) { -----> O(n)x(
       previous = n; -----> O(1)
       n = n.next; -----> O(1)
       index--; -----> O(1) )
   }

   previous.next = n.next; -----> O(1)

   return n.item; -----> O(1)}
```

Tanto a implementação do removeat da FintList e da LinkedList tem uma complexidade de O(n)

Método Deepcopy:

Dessa vez, não vamos utilizar da simulação de Monte Carlo, pois aumentar gradualmente o tamanho da lista nos permite ver melhor o crescimento do tempo de execução para cópia.

Como não há método deepCopy na package LinkedList, vamos usar comparar com a função shallowCopy presente na LinkedList.

Ensaios de razão dobrada:

Devido a natureza recursiva do shallowCopy, que faz o método criar um Stack frame em cada chamada recursiva, não é possível usar uma complexidade inicial elevada no computador em que foi testado. Caso contrário resultará num StackOverflow. Então, vai ser utilizada uma complexidade inicial de 23 para esta função e de 15000 para o deepCopy.

Resultados:

LinkedList

i	complexity	time(ms)	estimated r
0	23	12593.0	---
1	46	0.006102	0.4845549114587469
2	92	0.006128	1.0042608980662078
3	184	0.008748	1.4275456919060052
4	368	0.012566	1.4364426154549612
5	736	0.019889	1.5827630113003341
6	1472	0.041228	2.0729046206445774
7	2944	0.052168	1.2653536431551373
8	5888	0.100642	1.9291903082349333
9	11776	0.175534	1.7441426044792432
10	23552	0.493079	2.8090227534266865

FintList

i	complexity	time(ms)	estimated r
0	15000	245839.0	---
1	30000	0.468294	1.9048808366451213
2	60000	0.639706	1.3660350121931948
3	120000	1.226731	1.9176481070991986
4	240000	2.784026	2.2694673893461568
5	480000	5.19978	1.867719626181652
6	960000	8.98012	1.727019220043925
7	1920000	17.779121	1.97983111584255
8	3840000	34.965321	1.9666507134970284
9	7680000	69.681531	1.9928754836828182
10	15360000	154.137063	2.212021762265815

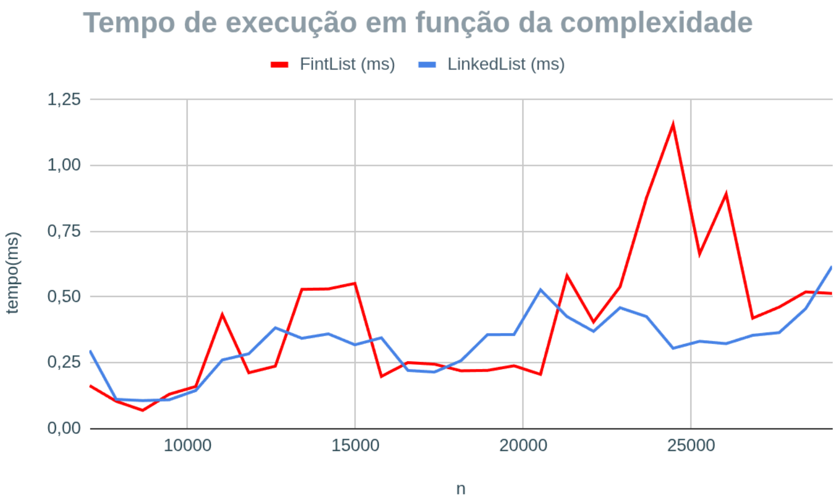
Após calcular uma aproximação do r médio para a FintList e para a LinkedList obtivemos o valor 1.91993 e 1.57562 respetivamente.

Assim concluímos que o método deepCopy na classe FintList tem uma complexidade temporal aproximada de n uma vez que $log_2(1.91993) \simeq 1$ e na classe LinkedList uma complexidade temporal aproximada de n dado que $log_2(1.57562) \simeq 1$

Ensaio gráfico:

Devido a natureza recursiva do shallowCopy, não será possível estudar bem a tendência das funções neste teste, pois terá de ser usada uma complexidade muito pequena.

Gráfico para comparação:



Complexidade assimptótica do método:

FintList

```
public FintList deepCopy() {
    if (isEmpty()) -----> O(1)
        return new FintList(); -----> O(1)
    FintList new_list = new
FintList(this.capacity); -----> O(1)
    new_list.elements = Arrays.copyOf(this.elements,
this.capacity); -----> O(n)
    new_list.next_index =
Arrays.copyOf(this.next_index,
this.capacity);-----> O(n)
    new_list.prev_index =
Arrays.copyOf(this.prev_index,
this.capacity);-----> O(n)

    new_list.head = this.head; -----> O(1)
    new_list.size = this.size; -----> O(1)
    new_list.tail = this.tail; -----> O(1)
    new_list.free_index = this.free_index;------>
O(1)

    return new_list; -----> O(1) }
```

LinkedList

```
public LinkedList<T> shallowCopy() {
    LinkedList<T> copy = new LinkedList<>(); -----> O(1)
    copy.size = this.size; -----> O(1)
    if (this.first != null) { -----> O(n)
        copy.first = this.first.shallowCopy();
    } -----> O(1)
    return copy;  -----> O(1)
}
```

Apesar de serem métodos diferentes, tanto a implementação da deepCopy de FintList e da shallowCopy da LinkedList possuem uma complexidade de O(n).

4)

Ao fazer a análise dos dados obtidos nos pontos anteriores, é perceptível que no geral a classe FintList tem uma ordem de crescimento menor que a classe LinkedList fornecida, pois até quando ambas possuem uma complexidade temporal linear, a FintList consegue se sobressair nos testes.

O primeiro exemplo está na análise das funções get(int index). O desempenho superior da LinkedList no melhor caso é praticamente irrelevante, pois ambos são, de certa forma, constantes. Contudo, no pior caso, a FintList apresenta metade da complexidade temporal da outra classe. Isso é graças às suas otimizações, que a permite percorrer a lista do final, início ou do último elemento acessado, enquanto o outro método apenas percorre do início.

De acordo com a análise assimptótica, todos os outros métodos estudados são O(n), crescem em tempo linear, pois todos precisam realizar uma travessia pelos elementos da lista. Contudo, os métodos da FintList mostram-se superiores em todos os testes, pois mesmo que as implementações sejam lineares, a FintList é superior devido a fatores constantes. Nos ensaios gráficos há uma representação visual clara, que mostra os métodos com uma ordem de crescimento menor e tempos de execução mais favoráveis comparados ao da LinkedList.

É crucial destacar os resultados do teste deepCopy. Infelizmente, não foi possível realizar testes de alta complexidade na função shallowCopy, o que impede a determinação exata da tendência do algoritmo. No entanto, este cenário realça a capacidade superior da Fintlist em lidar com valores significativamente mais elevados.

Por fim, a análise detalhada das classes revelou consistentemente a superioridade da FintList em termos de complexidade temporal. Os ensaios evidenciaram a robustez da FintList, que manteve um desempenho ótimo (por causa dos seus fatores constantes superiores) em cenários onde a LinkedList se mostrava lenta ou exigia restrições devido a sua natureza recursiva.

Portanto, os resultados empíricos e a análise assimptótica concluem que a FintList é a escolha mais eficiente e otimizada para as operações estudadas, superando a outra classe fornecida em praticamente todos os aspectos avaliados.

5)

A grande vantagem da FintList em relação a um vetor de inteiros é a sua capacidade de fazer inserções e remoções rápidas em qualquer posição, diferente do vetor que possuí tamanho fixo. Isso a torna mais versátil e fácil de gerenciar.

Pensando nisso, está biblioteca seria particularmente útil em motores de jogos, principalmente para o gerenciamento de objetos. Em jogos, elementos precisam ser manipulados, inseridos e removidos constantemente, como projéteis que são atirados e inimigos que aparecem e morrem. A ordem destes elementos é crucial, o que torna as trocas de posição frequentes uma necessidade. Além disso, a capacidade da FintList de reutilizar memória eliminada evita *resizes* constantes (comuns em vetores dinâmicos) e otimiza o uso da memória, um fator importante no desenvolvimento de jogos.

Embora o vetor possa superar a Fintlist na busca por objetos, a FintList compensa com sua capacidade de memorizar o último valor acessado, principalmente em objetos vitais para o funcionamento do jogo, que nunca deixam de ser acessados.