Universidade do Algarve

Faculdade de Ciências e Tecnologia

Licenciatura em Engenharia Informática

ALGORITMOS E ESTRUTURAS DE DADOS

RELATÓRIO

Projeto 2 – Coleções

Implementação de uma Fast Int List e Análise de Complexidade Temporal

Realizado por: Diogo Freitas nº 90147, Diogo Carvalho nº 90247.

Turma/Grupo: PL1-G10.

Regente da Disciplina: João Miguel Dias

Responsável pelas Aulas Práticas: Nestor Cataño Collazos

Ano Letivo: 2025/2026

Introdução

No Projeto 2 exploramos a consolidação de conhecimentos sobre estruturas de dados dinâmicas, com ênfase nas listas ligadas e nas implicações de localidade de referência. Em vez da abordagem tradicional baseada em nós e apontadores, adotamos uma implementação em armazenamento contíguo (vetor) que procura tirar partido da cache e reduzir overheads de gestão de memória, mantendo a semântica lógica de uma lista ligada.

A Parte A foca-se na construção de uma lista "rápida" de inteiros, seguindo a especificação fornecida e privilegiando operações eficientes nos extremos e acesso sequencial coeso em memória.

A Parte B dedica-se à análise de complexidade temporal desta solução: discutimos os custos teóricos e práticos das principais operações (inserções e remoções em diferentes posições, acessos, procura, percursos), e comparamos com uma implementação ligada tradicional.

PARTE B

Ponto 1

Como indicado vamos utilizar o número mais baixo que é 90147, para determinar qual o método vamos usar, 90147%4=3, logo o método é o removeAt.

Na parte A, desenvolvemos FintList que é um vetor com buffer circular. Ao remover em i, desloca-se o **lado mais curto** para fechar o buraco. Reduções de capacidade têm custo **O(1) amortizado** e não alteram a ordem.

Tempo por operação:

O número de elementos deslocados é

$$\text{shift}(i,n) = \min\{\, i,\ n-1-i \,\} \quad \Rightarrow \quad \mathrm{T}_{\text{removeAt}}(i,n) \sim min(i, n-1-i).$$

O Melhor caso é quando se trata uma remoção numa das extremidade (i=0 ou i=n-1), pois não há nenhum deslocamento, e fica $\boxed{\mathrm{T}_{\text{Melhor}} \sim 1.}$

No Pior caso acontece uma remoção ao meio (i\approx (n-1)/2), em que se desloca aproximadamente $n/2$, logo $\boxed{\mathrm{T}_{\text{Pior}} \sim n.}$

Ponto 2

LinkedList é uma lista duplamente ligada. E para remover em i, percorre-se a lista a partir do extremo mais próximo (head/tail) até ao nó i e faz-se o *unlink* em O(1). Não há *resize*; custo amortizado = custo real.

Tempo por operação:

$$min\{i, n-1-i\} \Rightarrow T_{removeAt}(i,n) \sim \min(i, n-1-i)$$

No Melhor caso, i=0 ou i=n-1, percorre 0 nós e da *unlink* em O(1) ⇒ $\boxed{T_{\text{Melhor}} \sim 1}$.

Já no Pior caso, i\approx(n-1)/2, percorre aproximadamente $n/2$ nós ⇒ $\boxed{T_{\text{Pior}} \sim n}$.

Ponto 3

Testes de razão dobrada — LinkedList<T>

Método: addAt (meio)

i	n	Tempo(ms)	r estimado
1	00128	000,5208	----------
2	00256	001,5625	3,000
3	00512	006,2500	4,000
4	01024	014,0625	2,250
5	02048	025,5208	1,815
6	04096	047,9167	1,878
7	08192	094,7917	1,978
8	16384	158,8542	1,676
9	32768	398,9583	2,511
10	65536	851,5625	2,134

Método: removeAt (meio)

i	n	Tempo(ms)	r estimado
1	00128	001,5625	----------
2	00256	006,2500	4,000
3	00512	004,6875	0,750
4	01024	019,2708	4,111
5	02048	032,8125	1,703
6	04096	047,9167	1,460
7	08192	094,2708	1,967
8	16384	176,5625	1,873
9	32768	379,1667	2,147
10	65536	846,8750	2,234

Testes de razão dobrada — FintList

Método: addAt (meio)

i	n	Tempo(ms)	r estimado
1	00128	000,5208	----------
2	00256	000,5208	1,000
3	00512	001,0417	2,000
4	01024	002,0833	2,000
5	02048	006,2500	3,000
6	04096	016,1458	2,583
7	08192	026,5625	1,645
8	16384	040,1042	1,510
9	32768	094,2708	2,351
10	65536	159,8958	1,696

Método: removeAt (meio)

i	n	Tempo(ms)	r estimado
1	00128	000,0000	----------
2	00256	000,0000	0,000
3	00512	000,0000	0,000
4	01024	002,6042	0,000
5	02048	004,6875	1,800
6	04096	014,5833	3,111
7	08192	022,9167	1,571
8	16384	040,6250	1,773
9	32768	098,4375	2,423
10	65536	158,3333	1,608

Como foram gerados os exemplos

Para cada tamanho n (progressão n, 2n, 4n, …), criamos listas com os inteiros 0..n−1.

- Operações testadas (por ponto n):
    - addAt: inserção no meio, de index = size/2);
    - removeAt: remoção no meio;

Para estabilizar a medição, repetimos a operação REPS vezes por ponto.

O que foi medido

Foi medido o tempo médio por ponto T(n) e razão dobrada r = T(2n)/T(n), usando TemporalAnalysisUtils. Quando há repetição interna, o tempo médio por chamada ≈ T(n)/REPS.

Resultados (valores de r)

Os últimos valores de r ficaram próximos de **2** em todos os métodos e em ambas as classes, para evitar leituras anómalas a conclusão baseia-se nas 3–4 últimas linhas.

Classe	**Método**	**r(mediana das ultimas linhas)**	**Complexidade**
FintList	addAt	~2	**Θ(n)**
FintList	removeAt	~2	**Θ(n)**
LinkedList	addAt	~2	**Θ(n)**
LinkedList	removeAt	~2	**Θ(n)**

FintList

- **addAt: Θ(n)**
  r nas últimas iterações é aproximadamente 2, logo o tempo cresce linearmente com o tamanho da lista.
- **removeAt: Θ(n)**
  r é aproximadamente 2, a remoção no centro implica deslocações/compactação linear.

LinkedList

- **addAt: Θ(n)**
  r é aproximadamente 2, é necessário percorrer praticamente n/2 nós até ao meio e inserir.

- **removeAt: Θ(n)**
  r é aproximadamente 2, é necessário percorrer praticamente n/2 nós até ao meio e remover.

Interpretação

Em FintList, o custo vem de mover dados (escritas contíguas); em LinkedList, vem de caminhar nós (seguindo ponteiros) e o *unlink* é O(1).

As ordens são iguais (mesmo perfil por posição i), mas as constantes diferem:

FintList tende a ser mais rápido quando os dados estão em memória contígua (bom cache) e o custo é escrever blocos.

LinkedList evita mover dados, mas sofre com cache misses na travessia de nós.

Nas Extremidades (i=0 ou i=n-1): ambas dão ~ 1 e no **Meio** (i\approx n/2): ambas dão $\sim n$.

$$\boxed{\text{Ambas: } T(i,n) \sim \min\{i,\ n-1-i\};\ \text{Melhor } \sim 1;\ \text{Pior} \sim n.}$$

Ponto 4

Os ensaios de razão dobrada realizados para inserção e remoção no meio da estrutura mostram, nas duas classes, valores de $r = T(2n)/T(n)$a convergir para $r \approx 2$nas últimas linhas das tabelas. Isto confirma empiricamente um crescimento linear $\Theta(n)$para addAt e removeAt tanto na FintList (vetor com buffer circular) como na LinkedList (lista duplamente ligada). As pequenas oscilações nas primeiras não alteram a tendência observada no regime assintótico.

Do ponto de vista teórico, o custo de ambas as operações é dominado por $\min\{i, n-1-i\}$:

na FintList, pelo deslocamento contíguo do bloco mais curto para abrir/fechar espaço; na LinkedList, pela travessia desde o extremo mais próximo até ao índice i, sendo a

ligação/desligação de nós em $O(1)$. Assim, no melhor caso (extremidades) obtém-se $T \sim 1$, e no pior caso (meio) $T \sim n$, em consonância com os gráficos/tabulações apresentadas.

Em termos práticos, as duas estruturas têm a mesma ordem assintótica, mas constantes diferentes:

- a FintList tende a apresentar tempos absolutos mais favoráveis quando os dados cabem em cache, porque o trabalho é mover blocos contíguos;
- a LinkedList evita mover elementos, porém paga acessos dispersos de memória na travessia, o que penaliza a constante temporal.

Conclusão: Os resultados experimentais e a análise coincidem: addAt e removeAt são $\Theta(n)$em FintList e em LinkedList, com melhor caso ~ 1, nas extremidades e pior caso $\sim n$, no centro. As diferenças observadas entre as classes resultam sobretudo de **efeitos de cache** e das **constantes** associadas a "mover dados" (FintList) versus "percorrer nós" (LinkedList).

Ponto 5

Por Exemplo, uma fila de eventos com tempo determinado. Cada evento (inteiro/ID) expira após um tempo determinado(Δt) e periodicamente:
1) Removem-se em lote os eventos no início que já expiraram.
2) Inserem-se no fim os novos eventos chegados.

Justificação:
A FintList (vetor com *buffer* circular) executa remoções sucessivas no início e inserções no fim em $\tilde{O}(1)$por operação, pois basta avançar o head e escrever no tail, sem deslocar o restante conteúdo. Num vetor de inteiros normal, cada remoção no início requer deslocar $\Theta(n)$elementos; um lote de k expirados custa $\Theta(k \cdot n)$, baixando o desempenho.

Conclusão de eficiência:
Para um fluxo com expirações frequentes, o custo total com FintList é $\tilde{O}(N)$(número de operações), enquanto com um array simples cresce para $\Theta(N \cdot n)$devido às deslocações. Assim, uma fila de eventos com time-outs é mais eficiente e, portanto, mais apropriada com FintList do que com um vetor de inteiros.