Relatório do Projeto 2 (Estrutura de Dados II)

Gabriel Sanches da Silva - N.USP: 11884693
Caio Assumpção Rezzadori - N.USP: 11810481
João Marcelo R. Júnior - N.USP: 8531118

1 Introdução

O projeto está dividido em 3 partes:

- Implementação: Funcionamento de cada busca com suas respectivas complexidades e explicação das alterações feitas nos templates fornecidas.
- Tabelas de resultados e análise empírica: Análise dos resultados obtidos, tabelas contendo as médias e desvios-padrão de cada algoritmo.
- Dificuldades e conclusão: Comentários das dificuldades encontrados na resolução e conclusão sobre os resultados.

O projeto baseia-se na construção de algoritmos de buscas. Com os templates propostos, implementamos diferentes versões da busca sequencial (Parte 1) e da busca por espalhamento/hashing (Parte 2).

2 Implementação

1ª parte

Item a

O primeiro algoritmo implementado foi a busca sequencial simples. De longe, o método de busca mais fácil e claro de ser implementado. O mesmo é feito através da verificação de cada uma das posições contidas no vetor, até o valor buscado ser encontrado ou a última posição ter sido lida. Como o algoritmo faz n comparações no pior caso (elemento não existe ou está na última posição da lista), sendo n o tamanho do vetor, a complexidade é $O(n)$ no pior caso.

Item b

O segundo algoritmo, por sua vez, é a busca sequencial com realocação por meio do método mover-para-frente. Após realizar uma busca bem sucedida, o valor é colocado

na posição inicial do vetor, e todos os valores precedentes a ele são movidos uma posição à frente de suas posições originais. Isso é feito com o intuito de melhorar futuras buscas por valores que já tinham sido requisitados, como um histórico do navegador fornecer um site frequentemente visitado. Contudo, uma das falhas do método mover-para-frente está nos valores antigos que não foram buscados novamente, fazendo com que o algoritmo fique muito semelhante a uma busca sequencial simples para esses casos. Em instâncias de vários deslocamentos para a primeira posição, o custo de mover os valores anteriores também pode aumentar o tempo de execução do código, tendo uma complexidade $O(2n)$ no pior caso (elemento está na última posição da lista), uma vez que são necessárias n comparações até achar o elemento buscado, e depois mais n operações para mover todos os valores da lista uma posição para frente ao inserir o elemento encontrado na primeira posição.

Item c

O terceiro algoritmo é a busca sequencial com realocação por meio do método de transposição. Semelhante ao algoritmo anterior, o valor buscado troca de posição, mas agora com o elemento anterior de onde foi encontrado, ao invés de ser colocado logo na primeira posição. Isso facilita a longo prazo na busca de um mesmo elemento, uma vez que ele vai ficando mais próximo do início da lista cada vez que é buscado, o que diminui o número de comparações necessárias para encontrá-lo a cada nova busca. Todavia, a complexidade no pior caso ainda é $O(n)$.

Item d

Por último, o quarto algoritmo é a busca sequencial utilizando índice primário. Sua natureza de criar uma nova tabela contendo índices e chaves do vetor original torna a busca rápida, algo muito eficiente para grandes listas de valores. Ao invés de pesquisar diretamente no vetor, a busca é feita no vetor de índice, que deve ser menor e fácil de percorrer, e este devolve a localização exata do valor buscado no vetor principal. Porém, por causa da criação de uma nova lista, esse algoritmo possui um custo de espaço maior do que os algoritmos anteriores.

Parte 2

Os próximos algoritmos se referem às variações da técnica de espalhamento/*hashing* para o tratamento de colisões. A ideia principal em tal método de busca é a criação de uma tabela auxiliar que guarda todas as chaves da lista possíveis de serem buscadas em

posições definidas por uma função *hash* h aplicada sobre o valor da chave a ser inserida. Após as inserções, para encontrar uma chave k, basta acessar a posição $h(k)$ da tabela. Tal técnica possui complexidade $O(1)$, uma vez que o cálculo da posição é instantâneo por meio da função *hash*.

Todavia, é muito difícil achar funções que mapeiem qualquer chave k em posições únicas na tabela *hash*, o que significa que é muito comum duas chaves de valores diferentes serem mapeadas na mesma posição, caracterizando uma colisão. Os próximos itens lidam com tal problema por jeitos diferentes.

Item a

A primeira técnica se chama *rehash* com *overflow progressivo*. Tal método calcula a posição da chave por uma certa função *hash* e, ao encontrar elementos ocupando a posição calculada (colisão), o algoritmo vai para a próxima casa da tabela e avalia se é possível inserir o elemento tratado, no caso de inserção; ou se o elemento buscado está na nova posição, no caso da operação de busca.

Para construir a tabela *hash*, foi utilizada uma estrutura que guarda uma variável ponteiro do tipo *unsigned*, caracterizando uma lista sequencial dinâmica de valores inteiros.

Além disso, foram utilizadas as seguintes funções *hash*:

$$h_{div}(x,i) = ((x \mod B) + i) \mod B$$

$$h_{mul}(x,i) = (fmod(x \cdot A, 1) \cdot B + i) \mod B$$

onde B é um número inteiro, A é um número real, mod representa o resto da divisão inteira do número à esquerda dele pelo da direita, e $fmod$ representa a parte fracionária da divisão do primeiro parâmetro pelo segundo.

Item b

A segunda técnica se chama *rehash* por *hash duplo*. Nele, utiliza-se cria-se uma função *hash* única a partir de outras duas. O registro utilizado para a construção da tabela auxiliar foi o mesmo do Item a, e a função utilizada foi:

$$h(x,i) = (h_{mul}(x,0) + i \cdot h_{div}(x,0)) \mod B$$

Item c

Como última técnica para tratar as colisões, foi implementada o *rehash* por *encadeamento em lista linear não ordenada*. Esta utiliza de uma lista encadeada para armazenar

todo elemento com uma mesma chave *hash*, onde cada elemento da lista aponta para o elemento alocado em seguida. Isto é, cada posição da tabela *hash* possui uma lista, e cada elemento dessa lista é um elemento que pertence à posição da tabela. Em outras palavras, se dois elementos possuem uma mesma chave (mesma posição na tabela), o primeiro elemento é o primeiro a ser apontado pela lista, que passa a apontar para uma nova região da memória onde o próximo elemento da mesma posição ficará.

Para encontrar a posição de cada elemento, foram utilizadas as funções *hash* a seguir:

$$h_{div}(x) = x \mod B$$

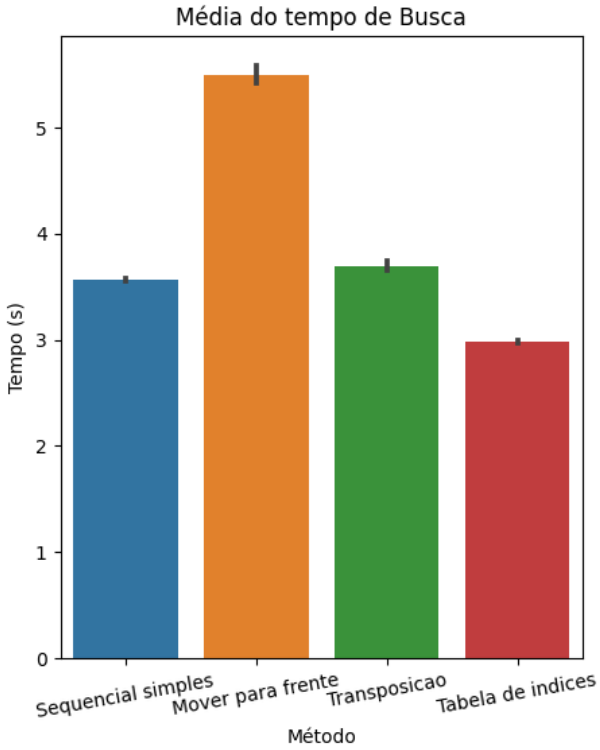
$$h_{mul}(x) = fmod(x \cdot A, 1) \cdot B$$

Vale ressaltar também que, para este último método, foi implementado um TAD para a estrutura de dados utilizada (Lista simplesmente encadeada), cujo nome do arquivo é *listaEncadeada.c*.

3 Tabelas de Resultado

Parte 1

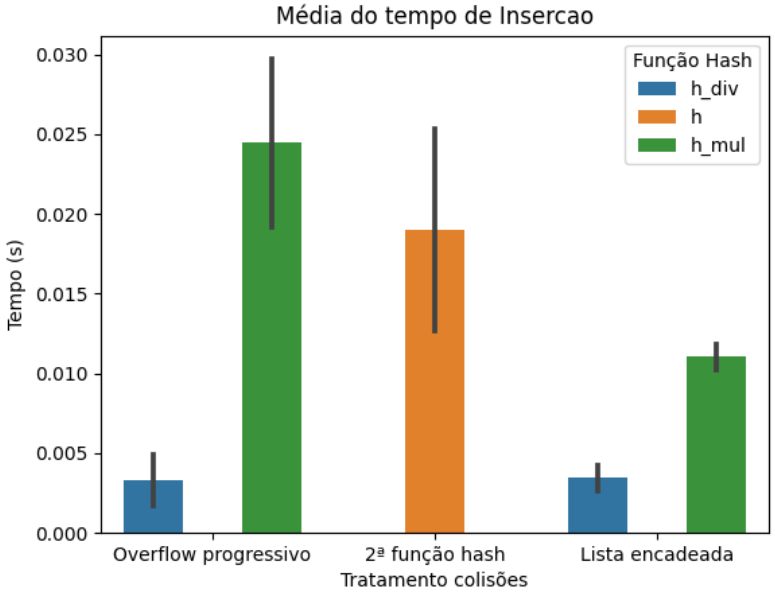
	Tempo (s)		
Método	Nº de execuções	Média	Desvio-Padrão
Sequencial simples	4	3.5714	0.0108
Mover para frente	4	5.5017	0.0847
Transposição	4	3.6953	0.0447
Tabela de índices	4	2.9797	0.0140

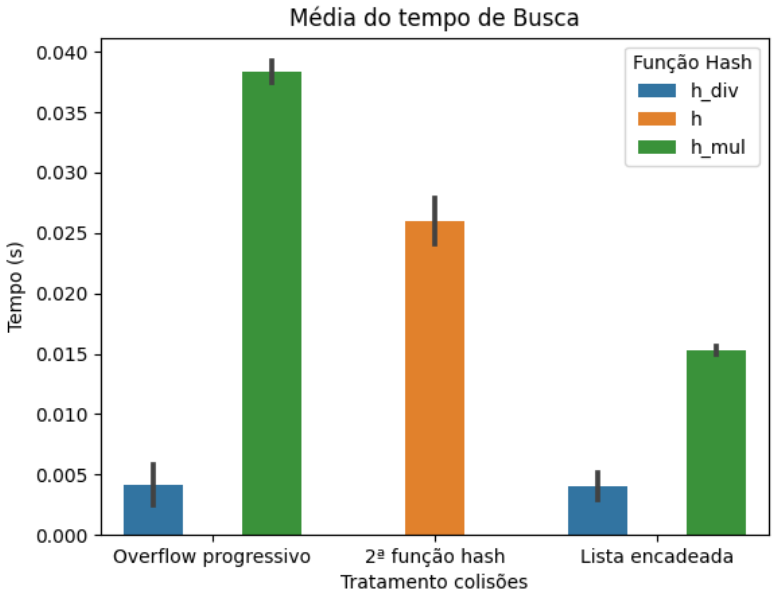


Parte 2

Tabela de resultados

			Tempo (s)		
Operação	Tratamento colisões	Função Hash	Nº de execuções	Média	Desvio-Padrão
Inserção	Overflow progressivo	h_div	4	0.0033	0.0016
		h_mul	4	0.0245	0.0052
	2ª função hash	h	4	0.0190	0.0063
	Lista encadeada	h_div	4	0.0035	0.0008
		h_mul	4	0.0110	0.0008
Busca	Overflow progressivo	h_div	4	0.0041	0.0017
		h_mul	4	0.0383	0.0009
	2ª função hash	h	4	0.0260	0.0019
	Lista encadeada	h_div	4	0.0040	0.0011
		h_mul	4	0.0153	0.0004





4 Dificuldades e Conclusão

Pelos resultados obtidos na Parte 1, concluímos que o método de mover-para-frente mostrou-se o menos eficiente, pois possui um tempo de execução médio alto e inconsistente, dado seus parâmetros de comparação. Como esperado, a busca sequencial simples obteve o menor desvio padrão por sua consistência "básica" ao executar a busca, além de uma média relativamente alta em tempo de execução. A busca por transposição demonstrou uma média semelhante à da busca sequencial simples, apenas com um desvio padrão aumentado por sempre alterar a posição dos elementos, algo que pode vir a prejudicar a consistências da busca. Por fim, a busca por tabela de índices concedeu ótimos resultados, tanto com um bom tempo de execução médio, quanto com uma baixa variação no tempo de busca, mas vale o comentário de que foi o único método que requisita de espaço de memória adicional.

Nos resultados referente a Parte 2, o Overflow progressivo apresenta média e desvio padrão semelhante à lista encadeada para o caso h_div. Contudo, utilizando a função h_mul, é notável uma menor alteração desses parâmetros no uso de lista encadeada. Por último, a 2ª função hash é mais próxima do overflow progressivo, sendo uma melhor opção que a mesma ao considerar h_mul.