UNIVERSIDADE FEDERAL DO MARANHÃO

DEPARTAMENTO DE INFORMÁTICA

CIÊNCIA DA COMPUTAÇÃO

ESTRUTURAS DE DADOS 2

SÃO LUÍS

2024.2

Guilherme Barrio Nascimento

Relatório do segundo trabalho prático

Trabalho prático apresentado ao componente curricular de Estrutura de dados 2, requisitado pelo Prof. D.r João Dallyson Sousa de Almeida

SÃO LUÍS

2024.2

Resumo

Nesse relatório, debateremos sobre a implementação de um pequeno projeto, que tem por objetivo comparar o desempenho de estruturas de dados

1. Introdução:

Ao longo desse Relatório, irei descrever detalhes sobre a implementação das estruturas solicitadas, desafios nessa implementação, e discutir o desempenho das estruturas no geral.

Informações sobre número de atribuições, movimentações, e tempo de execução em milissegundos, serão apresentadas apenas na implementação, reservando o espaço do relatório apenas para a complexidade e comentários adicionais.

2. Análise da complexidade:

2.1 Árvore AVL (TreeAVL):

- Inserção (insert): O(log n)

A árvore AVL mantém o balanceamento após cada inserção, garantindo que a altura da árvore seja logarítmica em relação ao número de elementos, resultando em um tempo de busca no pior caso de O(log n).

- Remoção (remove): O(log n)

Assim como a inserção, a remoção também mantém o balanceamento da árvore, garantindo a altura logarítmica.

- Busca (find): O(log n)

A busca é eficiente devido à altura da árvore, que é da Ordem O(log n).

- Rotação (rotateWithLeftChild, rotateWithRightChild): O(1)

As rotações são operações locais que não dependem do tamanho da árvore, resultando em complexidade constante.

2.2 Árvore Rubro-Negra (TreeRB):

- Inserção (insert): O(log n)

A árvore Rubro-Negra mantém o balanceamento aproximado, garantindo uma altura máxima de 2 * log(n + 1), que no pior caso resulta em um tempo de O(log n).

- Remoção (remove): O(log n)

A remoção também mantém complexidade O(log n).

- Busca (find): O(log n)

A busca é da ordem O(log n) devido à altura balanceada da árvore.

- Rotações (rotateLeft, rotateRight): O(1)

As rotações são operações que não dependem do tamanho da entrada, tendo complexidade linear.

2.3 Tabela Hash com Tentativa Linear (HashTentativaLinear):

- Inserção (put): O(1) em média, O(n) no pior caso.

No caso médio, a inserção é constante, mas no pior caso, em que ocorrem n colisões na mesma posição, pode ser linear.

- Busca (get): O(1) em média, O(n) no pior caso.

Em uma tabela hash com poucas colisões, a busca levaria tempo constante, mas no pior caso, os n elementos da tabela colidiram na mesma posição, sendo necessário n comparações para achar o elemento buscado.

- Remoção (delete): O(1) em média, O(n) no pior caso.

Busca é parte do processo de inserção, então, o pior caso da busca é o mesmo da remoção.

- Redimensionamento (resize): O(n)

Redimensionar a tabela requer rehashing de todos os elementos, ou seja, n reinserções numa nova tabela hash de tamanho M.

2.4 Lista Encadeada (LinkedList):

- Inserção (add): O(1) para inserção no final/começo, O(n) para inserção no meio.

Inserir no final ou no começo é constante, já que a depender da implementação, seria mantida uma referência para o elemento auxiliar para nossa inserção, mas inserir no meio requer percorrer a lista na ordem de O(n) elementos.

- Busca (contains): O(n)

A busca requer percorrer a lista linearmente.

- Remoção (remove): O(n)

A remoção requer percorrer a lista (complexidade linear) para encontrar o elemento.

3. Testes:

Os testes foram realizados comparando o desempenho das estruturas: Lista Encadeada, Árvore AVL, Árvore Rubro-Negra e Tabela Hash. em cima de três operações solicitadas:

1. Busca de elementos do conjunto A presentes em B.
2. Inserção de elementos do conjunto A em B, se o elemento não estiver já em B.
3. Remoção de elementos do conjunto A que estão presentes em B.

Os arquivos de entrada seguiram três padrões:

- ArquivoMenorPequeno.txt e ArquivoMaiorPequeno.txt:

(Conjuntos pequenos de dados – unidades de milhares)

- ArquivoMenorMedio.txt e ArquivoMaiorMedio.txt:

(Conjuntos médios de dados – centenas de milhares)

- ArquivoMenorGrande.txt e ArquivoMaiorGrande.txt:

(Conjuntos grandes de dados – unidades de milhões)

3.1 Resultados:

3.1.1 Lista Encadeada:

- Demonstrou desempenho inferior em buscas e remoções devido à complexidade O(n).

- Inserções foram rápidas apenas no final da lista.

3.1.2 Árvore AVL:

- Excelente desempenho em buscas, inserções e remoções, com complexidade O(log n).

- Manteve o balanceamento mesmo após várias operações.

3.1.3 Árvore Rubro-Negra:

- Similar à AVL, com desempenho O(log n) em todas as operações.

- Menos rotações que a AVL, mas com altura maior.

3.1.4 Tabela Hash:

- Excelente desempenho, em média O(1) - constante para buscas, inserções e remoções.

- Degradação no pior caso O(n) - linear devido a colisões.

4. Conclusão:

A implementação das estruturas de dados foi bem-sucedida, com todas as operações funcionando conforme o esperado.

A Árvore AVL e a Árvore Rubro-Negra demonstraram desempenho superior em cenários de buscas e operações de manutenção, ou de retorno dos elementos em ordem, enquanto a Tabela Hash foi a mais eficiente em média para operações de inserção e busca.

A Lista Encadeada, embora simples, mostrou-se inadequada para grandes volumes de dados devido à sua complexidade linear.

Algumas partes da implementação causaram problemas, tais como o Balanceamento das árvores, o tratamento de colisões na tabela hash, e os testes com o grande volume de dados.

As operações de balanceamento para árvores, especialmente após remoções, apresentaram diversos detalhes de implementação, que tornaram sua aplicação dificultosa, embora relativamente tranquila.

As colisões na tabela hash exigira alteração na forma como fiz o chaveamento de strings, especialmente lidando com o grande volume de dados dos últimos testes.

De forma geral, o Trabalho demonstrou em quais cenários cada estrutura tem melhor desempenho, e quais tipos de detalhes na implementação eu devo me atentar, a fim de manter a eficiência esperada de cada estrutura de dados.

Enquanto árvores balanceadas são ideais para operações dinâmicas com buscas frequentes, a tabela hash é a melhor para cenários onde a inserção e busca são predominantes, e poucas colisões ocorrem.

A lista encadeada, embora simples, é mais adequada para volumes pequenos de dados ou quando a ordem de inserção é importante.