RELATÓRIO ATIVIDADE RA2 - ÁRVORES BINÁRIA / AVL

Alunos: Gustavo Cesar Regnel

Clovis de Mello   
  
SOBRE O CÓDIGO:

Implementamos uma árvore binária comum (Arvore.java) e uma árvore AVL (ArvoreAVL.java);

Criamos uma classe Main para cada uma das árvores;

Ambas utilizam a classe de Nó (Node.java);

Ambas as árvores possuem as funções de inserção, busca e remoção;

Árvore Binária Comum(ArvoreBin):

Implementamos os seguintes métodos nesta classe:

inserir() -> método para inserir um nó com o dado na árvore, se a árvore estiver vazia, um novo nó é criado e definido como raiz. Se a raiz atual (ou subárvore) for nula, cria um novo nó. Se o dado for menor que o dado no nó atual, insere na subárvore esquerda. Se o dado for maior ou igual ao dado no nó atual, insere na subárvore direita, no final, retorna o nó raiz atualizado após a inserção.

removerNo() -> Método para remover um nó com um dado específico da árvore, se a raiz atual for nula, não há nada a fazer, Se o dado for menor que o dado no nó atual, remove da subárvore esquerda, se o dado for maior que o dado no nó atual, remove da subárvore direita, se o nó possui mais de uma instância do dado, decrementa a contagem e se o nó tem um filho ou nenhum, remove diretamente. No final, retorna o nó raiz atualizado.

percorrerPreOrdem(), percorrerNodos, getSucessor -> Adicionamos esses métodos para percorrer a árvore, para buscar o sucessor de um nó e também retornar os nós visitados como uma string, assim, montamos uma String de ‘’travessia’’.

Também adicionamos os métodos de imprimir() e de buscar().

Árvore AVL

Implementamos os seguintes métodos nesta classe:

Primeiramente, utilizamos um método calcularAltura(), como o nome já diz, calcula a altura de um nó na árvore AVL. Essa altura representa a distância entre o nó e as folhas mais distantes da árvore, calcula as alturas da sub árvore esquerda e direita do nó e retorna a altura máxima mais 1.

Métodos de rotação -> rotarDir() e rotarEsq() que recebem um nó inicial com parâmetro.

calcularBalanceamento () -> Calcula o fator de balanceamento de um nó. O fator de balanceamento é a diferença entre a altura da subárvore esquerda e a altura da subárvore direita do nó.

inserir() -> Adiciona um nó com a chave especificada na árvore AVL, mantendo o equilíbrio. Usamos a recursão para encontrar a posição correta para a inserção. Quando um nó é inserido, verifica o fator de balanceamento e aplica rotações, se necessário, para manter o equilíbrio da árvore.

buscar e buscarIgual() -> Busca na árvore um nó com a chave especificada. Já no buscarIgual, será realizada uma busca na árvore de um nó com a chave especificada e uma contagem de 1, desse jeito, lidamos com casos onde a árvore permite chaves duplicadas com contagem.

deletar() e getProx() -> Nesses métodos um nó com a chave especificada da árvore é removido, mantendo o equilíbrio. Após isso, será realizada uma verificação de balanceamento, para aplicar rotações caso necessário, depois, pegamos o próximo/sucessor da árvore.

procurar() -> Realiza uma busca na árvore e retorna o nó com a chave especificada.

Classes de Execução (Main)

Nessas duas classes, implementamos as lógicas para o funcionamento de cada uma das árvores.

São definidos os valores que serão adicionados as árvores.

Também é perguntado ao usuário quais valores ele deseja remover.

Também usamos InicioTempo e FimTempo para medir o tempo de duração de cada execução em MS.

NOSSA ANÁLISE DOS RESULTADOS

Ambos os testes foram realizados em um Desktop Windows

Utilizamos o JConsole - avalizamos o uso da CPU, alocação da memória, uso de threads e de classes  
  
Método de Inserção:

Árvore AVL

100 elementos: CPU 0.1% / Heap Memory 14,9MB / 17 Threads / 4.444 Classes

500 elementos: CPU 0.1% / Heap Memory 108MB / 15 Threads / 2504 Classes

1000 elementos: CPU 0.0% / Heap Memory 110MB / 17 Threads / 2502 Classes

10000 elementos: CPU 0.1% / Heap Memory 108MB / 16 Threads / 2503 Classes

20000 elementos: CPU 0.1% / Heap Memory 111MB / 16 Threads / 2499 Classes

Árvore Comum

100 elementos: CPU 0.1% / Heap Memory 110MB / 17 Threads / 2505 Classes

500 elementos: CPU 0.2% / Heap Memory 110MB / 16 Threads / 2504 Classes

1000 elementos: CPU 0.1% / Heap Memory 108MB / 16 Threads / 2500 Classes

10000 elementos: CPU 0.1% -> 0.6% / Heap Memory 113MB/ 16 Threads / 2504 Classes

20000 elementos: CPU 0,1% / Heap Memory 115MB / 17 Threads / 2511 Classes

Conclusão sobre a inserção ->

Com base nos resultados dos testes de desempenho, observamos que a implementação da Árvore AVL mostrou uma tendência geral de melhor eficiência em termos de uso de CPU e consumo de memória, especialmente ao lidar com um grande número de elementos. Embora ambos os tipos de árvores tenham demonstrado estabilidade em relação ao número de classes e threads, a Árvore AVL se destacou pela capacidade de manter um desempenho mais consistente e estável, mesmo com o aumento substancial na carga de dados. Isso sugere que, para operações que demandam uma quantidade considerável de elementos, a Árvore AVL pode oferecer uma solução mais eficiente e robusta em comparação com a Árvore Comum. No entanto, é importante considerar outros fatores específicos do contexto de aplicação ao decidir qual estrutura utilizar, como o tipo de operações realizadas e as restrições de recursos específicas do ambiente de execução  
  
  
Método de remoção

Árvore AVL

100 elementos: CPU 0.1% / Heap Memory 270MB / 76 Threads / 543589 Classes

500 elementos: CPU 0.0% / Heap Memory 14MB / 16 Threads / 4443 Classes

1000 elementos: CPU 0.1% / Heap Memory 58MB / 15 Threads / 2499 Classes

10000 elementos: CPU 0.0% / Heap Memory 61MB / 17 Threads / 2506 Classes

20000 elementos: CPU 0.0% / Heap Memory 63MB / 14 Threads / 2510 Classes

Árvore Comum

100 elementos: CPU 0.3% / Heap Memory 61MB / 16 Threads / 2499 Classes

500 elementos: CPU 0.0% / Heap Memory 62MB / 15 Threads / 2505 Classes

1000 elementos: CPU 0.0% / Heap Memory 64MB / 14 Threads / 2509 Classes

10000 elementos: CPU 0.0% / Heap Memory 62MB / 16 Threads / 2503 Classes

20000 elementos: CPU 0.0% / Heap Memory 297MB / 15 Threads / 2500 Classes

RESULTADOS

Apos a implementacao das arvores AVL e arvores Binarias, notamos que as duas tem jeitos diferentes de lidar com a organizacao dos dados nas estruturas de arvores. A arvore AVL uma árvore binária de busca balanceada, é projetada para manter seu equilíbrio através de rotações de nós, garantindo um desempenho estável em operações de inserção, remoção e busca. Por outro lado, a árvore binária é mais simples e carece de estabilidade automática, o que resulta em variações de desempenho que dependem da distribuição dos dados. As operações em árvores binárias podem ser mais eficazes em casos específicos, mas também podem degradar-se em situações com distribuição desigual de dados. A decisão entre estas estruturas será baseada em requisitos específicos de aplicação, as árvores AVL possuem um desempenho mais equilibrado, enquanto as árvores binárias são mais apropriadas para situações que requerem design simples e um alto grau de espaço de memória.

Sobre inserções, remoções e buscas, as árvores AVL são mais consistentes, o que garante um tempo de execução previsível, mesmo em cenários difíceis. As rotações de equilíbrio são iniciadas automaticamente, o que ajudará a manter a estabilidade no desempenho. Por outro lado, as árvoresbinárias têm um propósito específico em mente, mas não garantem um desempenho consistente em todos os cenários, a estrutura não é uniforme. Com isso, a decisão entre essas estruturas deve considerar a natureza dos dados e os requisitos específicos da aplicação, buscando um compromisso entre desempenho e complexidade de implementação.