Trabalho final da disciplina de Estrutura de Dados 2 (EDA2)
Alunos:
Lucas Gabriel
Leonardo de Souza
Santiago Cardoso

O objetivo deste trabalho consiste em analisar a complexidade algorítmica das operações de adição e remoção de nós (considerando o balanceamento) em árvores AVL, rubro-negra e B(ordens 1, 5 e 10). Essa complexidade será trabalhada em um alcance de até 10.000 elementos, com um espaçamento de 100. Ou seja, o custo de inserção e remoção será calculado para cada adição de 100 elementos no nosso conjunto de valores, da seguinte maneira:

(Elementos x Esforço) Para adição em uma AVL

Quantidade de elementos	esforço médio para adição(AVL)
100	24.80
200	27.53
300	29.40
400	31.20
…	…
10000	44.63

Obs: Essa tabela utiliza valores de exemplo e será formada para todas as árvores, levando em consideração tanto a adição quanto a remoção, logo teremos 2 tabelas para cada árvore.

Para conseguir representar um caso médio de inserção e remoção, todos os códigos foram trabalhados com base na geração aleatória de valores, utilizando para tal a função rand() do C, e para obter uma validade estatística consideramos uma média aritmética do esforço de 10 conjuntos, então para cada quantidade X de elementos, é feito 10 vezes o cálculo de esforço para essa quantidade X elementos, e através deles obtemos um esforço médio.

AVL:

Uma árvore AVL (Adelson-Velsky e Landis) é uma estrutura de dados na forma de árvore binária de busca balanceada. Ela mantém uma propriedade adicional para garantir que a altura das subárvores esquerda e direita de cada nó difira no máximo em 1, o que ajuda a manter o balanceamento da árvore.

Para garantir essa propriedade de balanceamento, a árvore AVL utiliza rotações simples e duplas em seus nós durante as operações de inserção e remoção. Essas rotações ajudam a reequilibrar a árvore sempre que uma operação pode quebrar a condição de balanceamento.

No início do código é criada uma variável global: “int esforco = 0;” a qual representará o esforço computacional exercido. Esse esforço é incrementado em 1, a cada:

- criação de um novo nó.
- rotação direita ou esquerda.
- chamada de função para calcular o fator de balanceamento.
- inserção.
- buscas dentro da inserção para encontrar a posição adequada do nó(direita ou esquerda) a partir da raiz.
- comparação “if” e “else” para determinar qual rotação realizar.
- nó deletado
- buscas a partir da raiz para encontrar o nó que deve ser deletado
- balanceamento da árvore após deleções

Na função main o usuário escolhe se deseja realizar o cálculo de esforço médio para a inserção ou para remoção. Os valores aleatórios são gerados e inseridos na árvore, e após calcular o esforço médio para inserção ou remoção, esses valores são alocados em um vetor “float esforcos_medio[ ]” o qual é responsável por guardar todos os esforços médios dos 100 aos 10.000 elementos. E então esses valores são colocados em um arquivo correspondente à escolha do usuário, caso o usuário tenha escolhido inserção, então os valores serão guardados em um arquivo chamado “avl_insercao.txt”. Caso a escolha tenha sido remoção, então os valores serão guardados em “avl_remocao.txt”.

Para esse trabalho foram realizadas as 2 operações, tanto de adição quanto remoção, e então os 2 arquivos foram preenchidos. O arquivo txt está estruturado conforme a tabela “(Elementos x Esforço) Para adição em uma AVL”, com seus valores separados por um espaço em branco. Esses arquivos futuramente servirão de auxílio para a geração do gráfico comparativo entre os algoritmos de árvores binárias.

Árvore Rubro-Negra:

Uma árvore Rubro-Negra é uma estrutura de dados na forma de árvore binária. Ela mantém propriedades adicionais para garantir seu funcionamento, cada um dos nós tem uma cor, vermelho ou preto, os requisito padrões de árvores de busca binárias, além de seus requisitos adicionais:

1. Um nó é vermelho ou preto.
2. A raiz é preta.
3. Todas as folhas (NULL) são pretas.
4. Ambos os filhos de todos os nós vermelhos são pretos.
5. Todo caminho de um dado nó para qualquer de seus nós folhas descendentes contém o mesmo número de nós pretos.

Para garantir essa propriedade de balanceamento, a árvore rubro-negra utiliza rotações e algoritmos de balanceamento que envolvem também a recoloração, em seus nós durante as operações de inserção e remoção. Essas rotações e balanceamentos ajudam a reequilibrar a árvore sempre que uma operação pode quebrar a condição de balanceamento.

No início do código são criadas variáveis globais: “int arvore_quant = 100;”, “int cont_insere = 0;” e “int cont_remove = 0;”, as quais, representaram respectivamente a quantidade de nós na árvore, o custo de inserção e o custo de remoção, que são incrementados da seguinte forma:

arvore_quant:

- incrementada de 100 em 100, até chegar em 10000, sempre que completamos a criação, inserção e remoção de uma árvore.

cont_insere:

- criação da árvore
- criação de um nó
- rotação à esquerda
- rotação à direita
- balanceamento
- inserção

cont_remove:

- transplantar nós
- encontrar o mínimo
- balanceamento da remoção
- remoção

Na função main abrimos os arquivos que armazenam os dados de custo de inserção e remoção, e inicializamos as variáveis “int custo_medio_insercao = 0;”, “int custo_medio_remocao = 0;” e “int r, i = 0;”, as quais, respectivamente, armazenam a média dos custos das operações de inserção e remoção, “r” representa o loop de realização dos algoritmos, repetidos 10x para garantir uma média que representa a realidade, e “i” representa os loops dentro de “r”. Abrimos um while de casos “while (arvore_quant < 10001) {” incrementamos de 100 em 100 o número de nós na árvore e calculamos a média de custos de inserção e remoção, além de escrevermos nos arquivos anteriormente abertos os valores, e por fim zeramos as variáveis de médias para o próximo conjunto de casos. Dentro do loop “r” criamos um vetor “int random_numeros[arvore_quant];” que possui números indo de 1 até arvore_quant (número de nós na árvore), após isso, realizamos o shuffle do vetor com a função “void random_num(int numeros[]);”, inicializamos a árvore rubro-negra e entramos em um loop que insere os arvore_quant nós aleatórios, dentro desse loop, contamos o custo de inserção do último nó inserido naquele conjunto de casos, e somamos o custo daquela operação na variável de média, por conseguinte,

fazemos o mesmo para a remoção, realizamos um novo shuffle, entretanto em um vetor com todos os nós, e retiramos o nó na posição 0 do vetor de dentro da árvore para calcular a média da remoção.

Para esse trabalho foram realizadas as 2 operações, tanto de adição quanto remoção, e então os 2 arquivos foram preenchidos. O arquivo txt está estruturado conforme a tabela “(Elementos x Esforço) Para adição em uma AVL”, com seus valores separados por um espaço em branco. Esses arquivos futuramente servirão de auxílio para a geração do gráfico comparativo entre os algoritmos de árvores binárias.

Árvore B:

Uma árvore B é uma estrutura de dados em árvore amplamente utilizada em computação, especialmente em sistemas de armazenamento e bancos de dados, para organizar e buscar informações de maneira eficiente.

Caracteriza-se por ter as seguintes propriedades:

- **Nós Internos e Folhas**: Um nó interno pode ter um número variável de chaves e um número fixo de filhos. As folhas estão no mesmo nível e contêm as informações.
- **Ordem:** A árvore B é definida por uma ordem fixa. Isso significa que cada nó interno pode ter no máximo um certo número de chaves e um certo número de filhos, determinados pela ordem da árvore.
- **Balanceamento:** A árvore B mantém um balanceamento entre suas ramificações, garantindo que todas as folhas estejam na mesma profundidade. Isso contribui para operações de busca, inserção e remoção eficientes, com um tempo de execução relativamente estável.
- **Busca Eficiente:** Devido à sua estrutura balanceada, as operações de busca em uma árvore B têm complexidade de tempo logarítmica, o que significa que são rápidas mesmo para grandes conjuntos de dados.

Operações Básicas da Árvore B:

1. criaArvore(int ordem): Cria uma nova árvore B com a ordem especificada.
2. criaNo(ArvoreB* ArvoreB): Cria um novo nó para a árvore B.
3. percorreArvore(No* no): Realiza um percurso em ordem na árvore B, imprimindo as chaves.
4. localizaChave(ArvoreB* ArvoreB, int chave): Localiza uma chave específica na árvore B.

5. adicionaChave(ArvoreB* ArvoreB, int chave): Adiciona uma chave à árvore B, realizando as operações de divisão do nó quando necessário.
6. removerChave(ArvoreB* ArvoreB, int chave): Remove uma chave da árvore B, realizando as operações de rearranjo ou fusão de nós quando necessário.

Operações de Divisão e Fusão:

7. transbordo(ArvoreB* ArvoreB, No* no): Verifica se um nó excedeu sua capacidade máxima.
8. divideNo(ArvoreB* ArvoreB, No* no): Divide um nó em dois quando excede a capacidade máxima.
9. rearranjaNo(ArvoreB* ArvoreB, No* no, int indice): Realiza ajustes após uma remoção, redistribuindo chaves ou fundindo nós.
10. mergeNos(ArvoreB* ArvoreB, No* no, int indice): Funde dois nós quando a redistribuição não é possível após uma remoção.

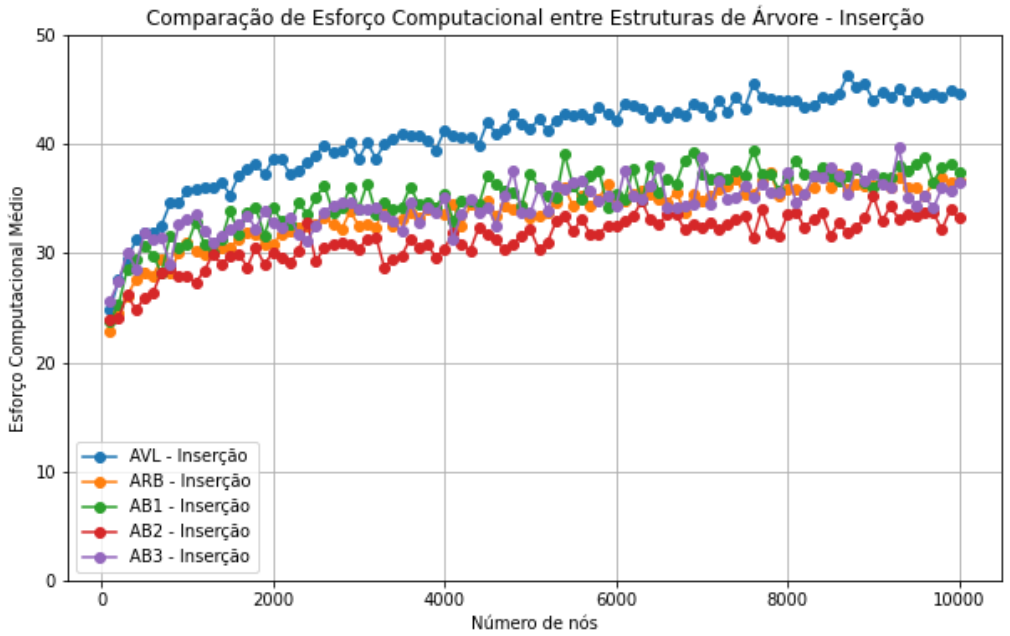
Funções para Medição de Desempenho:

avgcase1(), avgcase5(), avgcase10(): Geram dados de desempenho médio para operações de inserção e remoção em árvores B de ordem 1, 5 e 10, respectivamente. Os resultados são armazenados em arquivos.

Gráficos comparativo de esforços final AVL x RED_BLACK x AB**:**

Utilizamos a biblioteca **matplotlib** do python, para ler os arquivos .txt de inserção e remoção para cada árvore, e gerar um gráfico comparativo dos esforços de cada algoritmo.

Esforço comparativo de **inserção:**



Esforço comparativo de **remoção:**