Relatório De Estruturas De Dados

Aluno: Marcos Dalessandro Cavalcante Lima (MATRÍCULA: 20209016090)
Campus Senador Helvídio Nunes de Barros - Picos
Professor(a): Prof(a) Juliana Oliveira Carvalho
Disciplina: Estruturas de Dados 2 - 2023.1
C.H.: 60 h Créditos: 2.2.0 Período: 2023.1
28 de julho de 2023

1 Resumo

O presente trabalho apresenta de forma breve as estruturas de dados do tipo Árvore 2-3(2-3) e Árvore Rubro-Negra(LLRB) estudadas durante a disciplina de estruturas de dados 2. Compreende também o trabalho proposto pela professora Juliana Oliveira Carvalho, de forma detalhada as funções desenvolvidas.

2 Introdução

Esta seção tem como objetivo introduzir o leitor ao trabalho. Árvores são estruturas de dados hierárquicas amplamente utilizadas na ciência da computação. Elas são compostas por nós interconectados, onde cada nó possui uma referência para um conjunto de outros nós chamados de filhos. A árvore possui um nó especial chamado raiz, que é o ponto de partida da estrutura, e os nós sem filhos são chamados de folhas. Este trabalho tem como principais objetivos, apresentar o leito o funcionamento desses dois tipos básicos de árvores e seu desempenho com grandes quantidades de dados. É de suma importância pontuar que não existe melhor estrutura, existe a que resolve melhor o teu problema. Os algoritmos foram executados em um notebook Lenovo Ideapad Gaming 3i, uma maquina capaz de processar a base de dados utilizada sem tanto esforço.

3 Árvores

Com o decorrer da seção será apresentado os tipos de árvores que foram utilizadas durante o presente trabalho. Existem vários tipos de árvores, cada uma com suas características específicas. Porem o presente trabalho foca apenas em duas delas:

3.1 Árvore 2-3

A árvore 2-3 é uma estrutura de dados de árvore balanceada com nós internos contendo dois ou três filhos. Ela organiza dados de forma hierárquica e eficiente, permitindo operações de busca, inserção e exclusão em tempo logarítmico na altura da árvore, proporcionando desempenho eficiente para um grande número de elementos.

Cada nó interno pode conter até dois elementos e até três filhos, dependendo do número de elementos presentes. Existem três tipos de nós na árvore 2-3: nó 2-nó, nó 3-nó e nó raiz.

O nó 2-nó contém apenas um elemento e tem dois filhos. O nó 3-nó contém dois elementos e possui três filhos. O nó raiz contém os elementos no topo da árvore.

As regras principais são:

Todos os elementos estão organizados em ordem ascendente nos nós. Todos os nós folha estão no mesmo nível, ou seja, têm a mesma profundidade. Se um nó tem dois filhos, os elementos dos filhos estão em ordem estritamente crescente. Se um nó tem três filhos, os elementos do primeiro filho são menores que o elemento do nó, e os elementos do terceiro filho são maiores que o elemento do nó. A inserção é cuidadosamente realizada para manter as regras da árvore. Quando um nó está cheio e um novo elemento precisa ser inserido, o nó é dividido em dois e o elemento intermediário é promovido para o nível superior. Esse processo é chamado de "divisão"ou "split". A divisão pode ser propagada para cima na árvore, caso o pai do nó dividido também esteja cheio, mantendo o balanceamento e eficiência da árvore após a inserção.

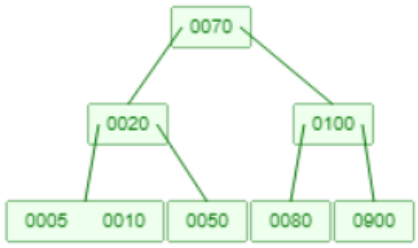

Figura 1 – Exemplo de ABB.

Da mesma forma, a exclusão de elementos é realizada com cuidado para garantir que as regras da árvore não sejam violadas. Se um nó fica com apenas um elemento após a exclusão, ele pode ser "fundido"com um irmão que também tenha apenas um elemento.

A árvore 2-3 oferece benefícios significativos em termos de equilíbrio e tempo de execução de operações, tornando-a uma estrutura de dados eficiente para gerenciar grandes conjuntos de dados em muitos cenários. No entanto, vale ressaltar que, embora a árvore 2-3 seja uma estrutura importante, existem outras árvores balanceadas com diferentes características que também são amplamente utilizadas na ciência da computação, como a árvore AVL e a árvore rubro-negra.

3.2 Árvore Rubro-Negra (Left Leaning Red and Black Tree)

A Árvore Rubro-Negra Pendente para Esquerda é uma variação da árvore rubro-negra, uma estrutura de dados binária de busca balanceada. Proposta por Sedgewick em 2008, essa variação tem como objetivo simplificar as operações de inserção e exclusão, mantendo as principais propriedades da árvore rubro-negra.

Na árvore rubro-negra, cada nó é colorido como vermelho ou preto, e possui as seguintes propriedades: todo nó vermelho possui apenas filhos pretos, a raiz da árvore é sempre preta, todas as folhas nulas (filhos dos nós folhas) são consideradas pretas, e para cada nó, qualquer caminho da raiz até uma folha nula contém o mesmo número de nós pretos.

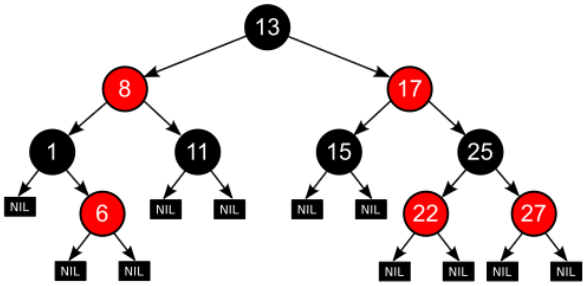

Figura 2 – Exemplo de Rubro Negra Padrão.

A Árvore Rubro-Negra Pendente para Esquerda simplifica a árvore rubro-negra tradicional ao adotar algumas modificações: os nós filhos à esquerda são mais profundos, garantindo que sempre tenham uma profundidade maior em comparação com os nós filhos à direita. Ao inserir um novo elemento, ele será sempre colocado à esquerda de seu pai, tornando a árvore pendente para esse lado. Além disso, a propriedade das folhas nulas é mantida, assim como a propriedade da profundidade negra, onde o número de nós pretos ao longo de qualquer caminho da raiz até uma folha nula permanece o mesmo.

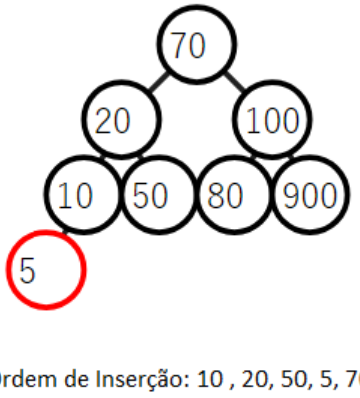

Figura 3 – Exemplo de LLRB.

Essas modificações simplificam a lógica de inserção e exclusão na Árvore Rubro-Negra Pendente para Esquerda, enquanto ainda oferecem uma árvore binária de busca balanceada

com desempenho eficiente para operações de busca, inserção e exclusão em tempo logarítmico na altura da árvore.

4 Problema Proposto

Esta seção tem como objetivo apresentar ao leitor o problema proposto para implementação de Árvores 2-3 e LLRB. O problema propõe a escrita de um programa na linguagem C que realiza uma referência cruzada através da construção de uma árvore vermelha-preta. O programa recebe um arquivo texto como entrada, onde todas as palavras são armazenadas na árvore, juntamente com os números das linhas em que cada palavra foi utilizada. Os números das linhas são associados aos nós da árvore por meio de uma lista encadeada.

O processo inicia com a leitura do arquivo de entrada, onde as palavras são separadas e inseridas na árvore vermelha-preta. Cada nó da árvore contém a palavra como chave e uma lista encadeada para armazenar os números das linhas em que a palavra foi usada. Caso a palavra já exista na árvore, o programa insere o número da linha na lista associada ao nó correspondente.

Após o processamento do arquivo de entrada, o programa imprime todas as palavras em ordem alfabética, juntamente com os números das linhas em que cada palavra foi utilizada. Para isso, realiza-se um percurso in-order na árvore vermelha-preta.

O programa também oferece a funcionalidade de busca de uma palavra inserida pelo usuário. Ao receber a palavra de busca, o programa verifica sua existência na árvore e, caso encontrada, imprime os números das linhas onde foi utilizada.

Além disso, o programa permite que o usuário exclua ou acrescente uma palavra em uma linha específica. Para realizar a exclusão, o programa encontra a palavra na árvore, localiza o nó correspondente e remove o número da linha da lista associada a esse nó. Já para a inserção, o programa verifica se a palavra já existe na árvore e, caso contrário, a insere na posição adequada com um novo nó e a lista de números de linha associada a esse nó.

Com essa implementação, o programa proporciona uma referência cruzada eficiente para o arquivo de texto fornecido, permitindo consultas rápidas e a manipulação das palavras e linhas de forma interativa pelo usuário.

4.1 Funções:

Esta seção tem como objetivo apresentar as funções que foram criadas e usadas pra resolver os problemas

4.2 LLRB

void rEsquerda(ArvRN **RaizArv); - Função de rotação para a esquerda, recebe a raiz por referência;
void rDireita(ArvRN **RaizArv); - Função de rotação para a direita, recebe a raiz por re-

ferência;
void trocaCor(ArvRN **RaizArv); - Função de troca de cor, recebe a raiz por referência;
void balancearLLRB(ArvRN **RaizArv); - Função de balanceamento, recebe a raiz por referência;
char verificaCor(ArvRN *RaizArv); - Função auxiliar para verificar a cor do nó;
void EntradasDados(ArvRN **RaizArv, char *valor, int repetidas); - Função que cuida da parte de alocar memória para a inserção, recebe a Raiz pro referencia, palavra e quantidade de interações;
void inserir(ArvRN **RaizArv, char *valor, int repetidas); - Função de inserção de fato, apóos ter alocado a memória essa função acha o lugar para inserir o nó de fato;
void ImprimeArvoreInOrdemLLRB(ArvRN *RaizArv); - Função auxiliar para imprimir a árvore na ordem, recebe a Raiz da árvore;
void ImprimirListaLinhas(ArvRN *RaizArv); - Função auxiliar para imprimir a lista de palavras repetidas;
void BuscarPalavra(ArvRN *RaizArv, char *valor); - Função de busca na árvore, recebe a árvore e mostra onde está o nó buscado.

4.3 2-3

void LeituraDadosArq(Arv23** RaizArv23, InfoPalavra** PalavraSobe); - Função que lê as palavras que estão sendo inseridas e chama a próxima função que de fato vai alocar espaço e inserir na árvore, recebe a raiz da árvore por referencia e a palavra que sobe;
Arv23* InsereArv23(Arv23** RaizArv23, Arv23* Pai, InfoPalavra** PalavraSobe, char* PalavraInfo, int LinhaPalavra); - Função que acha o local onde aquela palavra deve ser inserida e decide se precisa quebar o nó ou não. Recebe a raiz da árvore, o pai do nó atual, a palavra que sobe, a palavra a ser inserido e a linha que essa palavra está sendo inserida;
void AdicionaInfoNo(Arv23 **RaizArv23, InfoPalavra* PalavraInfo, Arv23* Filho); Arv23* CriaNo23(InfoPalavra* PalavraInfo, Arv23* Esq, Arv23* Centro, Arv23* Dir); Arv23* QuebraNo(Arv23** No, InfoPalavra** Palavra, InfoPalavra* PalavraInfo, Arv23* Filho); - Função que quebra o nó quando preciso, caso um nó tenha as duas informações cheias, recebe a raiz da arvore, a palavra a ser inserida, as palavras do nó, os filhos e retorna a árvore com um nó alocado;
InfoPalavra* CriaInfoPalavra(char* Palavra); - Função responsável pro alocar o nó para a inserir a palavra, recebe a palavra e retorna o nó alocado;
int VerificaFolha(Arv23* NoArv23); - Vifica que se o nó passado é um nó folha, recebe o nó da árvore e retorna um interio que funciona no código como um valor booleano;
InfoPalavra* BuscaInfo23(Arv23* RaizArv23, char* Palavra); - Função de busca para a palavra desejada, recebe a raiz da árvore e a palavra que se deseja buscar, retorna onde a palavra se encontra na árvore;
void ImprimeInfo(InfoPalavra* InfoNo); - Função auxiliar que mostra onde a informação esta e qual a linha que ela se encontra na lista, recebe o nó;

void ImprimeNo23(Arv23* No); - Imprime o nó desejado, recebe a nó; void ImprimeArv23(Arv23 *RaizArv23, int nivel); - Imprime a árvore por completo, recebe a árvore e os níveis;
void RemovePalavra23(Arv23** RaizArv23, Arv23** Pai, char* Palavra, int LinhaPalavra); - Função responsável por remover a palavra da árvore e da lista, recebe a raiz da árvore por referência, o pai do nó, a linha que a palavra se encontra e a propria palavra;
void RemoveMaiorInfoEsq(Arv23** RaizArv23, Arv23** PaiMaior, Arv23** MaiorInfoRemove, int LocalInfo); - Função auxiliar para remoção, acha o nó mais a esquerda e substitui como nó desejado para remoção, recebe a raiz da árvore por referência, o pai do maior nó, a localização da palavra e a maior info presente no nó;
void RedistribuiArv23(Arv23** RaizArv23, Arv23** Pai); - Função auxiliar a remoção, tem como objetivo reajustar a árvore garantindo que nenhum propriedade seja quebrada durante as remoções, recebe a árvore por referencia e o pai do nó por referencia.

4.4 Lista Encadeada Simples

void InserirLista(rLinhas **NoListaLinhas, int NumLinha); - Função que insere na Lista encadeada, recebe a posição e o número da linha, e faz a inserção na lista;
rLinhas* CriaLista(int NumLinha); - Função que cria a lista encadeada, recebe o numero da linha e retorna o espaço alocado para a inserção;
void ImprimirLista(rLinhas *NoListaLinhas); - Função para impressão da lista encadeada, recebe apenas a lista.

4.5 Problema

Será realizado o mesmo trabalho do problema anterior:
Observação 1:para os experimentos cada execução deve inserir os mesmos códigos de cursos em ordem diferente (pode utilizar comando para embaralhar os códigos e assim as árvores ficarem bem aleatórias e poder verificar a diferença entre inserções).
Observação 2: Lembre-se que não pode haver impressão entre o tempo inicial e o tempo final, pois impressão consome muito tempo.
Observação 3: Para validar o tempo de busca, o mesmo deve ser repetido 30 vezes. Faça uma média para obter o resultado final.

No primeiro teste, foi decidido a quantidade de dados que seria utilizado para este experimento, chegamos a conclusão que 240000 valores seriam uma boa quantidade de dados para o presente trabalho. Para a unidade de medida de tempo, tivemos a ideia de utilizar mili-segundos, pois o números ficariam mais visíveis. Esse foi o primeiro problema enfrentado durante o presente trabalho, como iriamos selecionar esses valores, sendo eles sem assento gráfico e palavras comuns, que usamos no nosso dia a dia.

Então achamos um repositório com a exata quantidade de palavras desejadas, tivemos outro problema, as palavras estavam em ordem alfabética, logo criamos um script simples em Python que nos ajudava a pegar todas as palavras do repositório e embaralha-las em um arquivo de texto.

Durante a busca foi acordado que como medição de tempo seriam utilizados 100000 palavras e feitas a média de busca que será apresentada na próxima seção.

4.5.1 Tempo de Busca de 100000 Elementos

LLRB:
Média de Busca: Palavra: 0.001939 ms
2-3:
Média de Busca: Palavra: 0.004970 ms