Árvore Rubro-Negra

Arthur Barreto Godoi e Ricardo Quer Do Nascimento Filho
Departamento de Informática
Universidade Federal do Paraná – UFPR
Curitiba, Brasil

***Resumo*—Implementação de uma árvore binária rubro-negra e suas propriedades.**

I. Introdução

Este relatório analisa a implementação de uma **árvore rubro-negra em linguagem C.** As árvores rubro-negras são árvores binárias de busca balanceadas, bastante utilizadas em programas que exigem operações eficientes de inserção, remoção e busca. Sua estrutura garante que o caminho mais longo da raiz até uma folha seja, no máximo, duas vezes o caminho mais curto, proporcionando **complexidade de tempo O(logn)**.

II. Estrutura do código

A implementação apresentada organiza as funções de manipulação de nós e manutenção das propriedades da árvore rubro-negra em dois arquivos: um **cabeçalho (librb.h)** que define as estruturas e funções, e um arquivo de **código fonte (librb.c)** que contém suas implementações. Além disso, há um **programa principal (main.c)** para interagir com a estrutura.

1) **Estrutura da Árvore Rubro-Negra:**
   a) **rb tree t():** Representa a árvore em si, com um ponteiro para o nó raiz.
   b) **rb node t():** Representa um nó, armazenando sua chave, cor, letra associada, ponteiros para o nó pai e filhos (esquerdo e direito).
   c) **Color:** Uma enumeração Color define as cores dos nós: RED e BLACK.
2) **Inicialização e Destruição:**
   a) **create rb tree():** Cria e inicializa uma nova árvore.
   b) **create rb node():** Aloca e inicializa um novo nó com valores específicos.
   c) **delete rb tree() e delete rb node():** Liberam a memória alocada para a árvore e seus nós.
3) **Inserção:**
   a) **insert rb node():** Insere um nó com chave e valor especificados. Garante que chaves duplicadas não sejam inseridas.
   b) **fix rb tree():** Balanceia a árvore após uma inserção, garantindo que suas propriedades sejam mantidas.
4) **Remoção:**
   a) **remove rb node():** Remove um nó com uma chave especificada, garantindo que a árvore permaneça balanceada.
   b) **balance removal():** Resolve os desequilíbrios após a remoção de um nó.
5) **Busca:**
   a) **search rb():** Busca a chave na árvore e retorna o caractere associado.
   b) **search rb node():** Encontra e retorna o nó correspondente à chave.
   c) **find predecessor():** Encontra o maior nó na subárvore esquerda de um nó especificado.
6) **Operações de Balanceamento:**
   a) **rotate left() e rotate right():** Executam rotações para manter as propriedades da árvore rubro-negra.
7) **Travessia e Impressão:**
   a) **print rb tree() e print rb inorder():** Percorrem e imprimem os nós da árvore em ordem, exibindo a chave, nível e cor.
8) **Arquivo Principal:** O programa principal permite a interação com a árvore através de comandos:
   a) **i (chave):** Insere a chave na árvore.
   b) **r (chave):** remove a chave da árvore. Após processar os comandos, a árvore é impressa em ordem e liberada da memória.

III. Conclusão

Com base na implementação e análise realizadas, pode-se concluir que as árvores rubro-negras são estruturas de dados altamente eficientes e práticas para manter os dados organizados e balanceados. Sua principal característica é o balanceamento automático, o que garante que as operações de busca, inserção e remoção ocorram em tempo O(logn), independentemente da sequência de operações realizadas. Em resumo, as árvores rubro-negras são uma escolha excelente para aplicações que exigem eficiência em tempo de execução e manipulação dinâmica de dados. Esta implementação prática reforçou os conceitos vistos em aula, demonstrando que, embora sua construção e manutenção sejam desafiadoras, os benefícios compensam o esforço, tornando-as uma ferramenta indispensável em várias áreas da computação.