**Árvore Rubro-negra - Matheus Gastal Magalhães e Antônio Ressurreição Filho**

**Objetivo:**

Com as aulas de Algoritmos 3, aprendemos sobre árvores de busca binária e, dentro dessa classe, árvores rubro-negras, que possuem particularidades no método de implementação. Nosso objetivo era desenvolver em C toda a manipulação de árvores rubro-negras, desde structs, criação de nós NIL, criação de nós com chaves, inserção, deleção e saída em ordem. Dentro do objetivo principal, também nos atentamos às boas práticas de programação e clareza nas ideias, usando como base para o código o livro *Algoritmos - Teoria e Prática* de Cormen, Leiserson, Rivest e Stein. As condições básicas para uma árvore ser rubro-negra são: cada nó é vermelho ou preto, o nó raiz é preto, todas as folhas (NIL) são pretas, se um nó é vermelho, então seus filhos são pretos, e para cada nó, todos os caminhos de um nó até seus descendentes possuem o mesmo número de nós pretos.

**Desenvolvimento do projeto:**

Inicialmente, desenvolvemos as funções e structs mais simples, como a definição de nós normais, nós NIL, árvore e criação de todas as estruturas necessárias. Além disso, definimos de prontidão o nivelamento da árvore para o print final e também a função que imprime em ordem. Após preparar o terreno para realmente começarmos a implementar as funções complexas da árvore rubro-negra, começamos pela inserção.

Em um primeiro momento, a inserção é bem simples: seguindo as propriedades de árvores de busca binária, inserimos o nó com chave menor à esquerda de seu pai e o maior à direita. Porém, logo após terminarmos a função de inserção, chamamos a função para verificar as propriedades da árvore rubro-negra. Todo o funcionamento da função se baseia em uma sequência de decisões que precisam ser tomadas conforme a situação encontrada. Primeiro, verificamos se o pai do novo nó é vermelho, pois, se ele for preto, não há problema e nada precisa ser feito. Se o pai for vermelho, olhamos de qual lado ele está em relação ao avô, porque isso vai determinar o caminho que seguimos. Se o pai for filho da esquerda, nossa atenção se volta para o tio, que é o filho direito do avô. Agora, precisamos decidir: se o tio também for vermelho, simplesmente trocamos as cores do pai e do tio para preto e pintamos o avô de vermelho, depois subimos na árvore e recomeçamos as verificações a partir do avô. Porém, se o tio for preto ou não existir, é preciso reorganizar a árvore: se o novo nó for filho direito, fazemos primeiro uma rotação à esquerda no pai para acertar a posição. Depois disso, pintamos o pai de preto, o avô de vermelho, e fazemos uma rotação à direita no avô para corrigir completamente. Se o pai estiver à direita do avô, seguimos uma linha de raciocínio parecida, só que agora o tio analisado é o filho esquerdo do avô, e as rotações acontecem em sentidos trocados. Ao final de todas essas decisões, garantimos que a raiz da árvore seja preta, mantendo as propriedades que definem uma árvore rubro-negra.

Logo após finalizarmos a inserção, partimos para a parte mais desafiadora do projeto, que é a remoção dos nós. Isso se deve a alguns motivos: achar o nó baseado somente na chave dele, percorrendo a árvore até encontrá-lo, e desafios para tomar a decisão correta em cada caso, pois, na remoção, também usamos a função transplante, e entender completamente seu funcionamento foi um pouco confuso. Fizemos na remoção a função que busca o nó baseado na chave e retorna o ponteiro, e a função de transplante, que basicamente retira um nó da árvore e direciona os ponteiros a fim de não perder a estrutura. Inicialmente, a

remoção não é complicada: primeiro verificamos a ausência de um dos filhos, se isso for o caso, fazemos o transplante com o nó diferente do inexistente e o nó a ser removido. Caso o nó a ser removido possua os dois filhos, precisamos fazer uma análise um pouco mais profunda, achando outro nó para ocupar seu lugar, definido no enunciado para utilizarmos o antecessor. Depois de encontrar esse nó, salvamos sua cor original para usar mais tarde. Então decidimos: se o sucessor encontrado for filho direto do nó que estamos removendo, apenas ajustamos seu filho para apontar de volta para ele, mantendo a ligação. Caso contrário, fazemos um transplante para mover o sucessor para cima, colocamos seu filho esquerdo no lugar dele, e depois conectamos corretamente os filhos do nó removido ao sucessor. Depois que o sucessor já ocupa o lugar do nó removido, copiamos a cor do nó antigo para o sucessor, para manter a árvore correta. Por fim, se o sucessor que saiu de sua posição original era preto, chamamos a função para arrumar a árvore, já que remover um nó preto sempre desbalanceia as propriedades da árvore rubro-negra. Nessa função, enquanto o nó que estamos analisando for preto e não for a raiz, precisamos pensar nos próximos passos. Primeiro, olhamos se o nó é filho da esquerda ou da direita do seu pai, porque isso define de que lado vamos agir. Se o nó estiver à esquerda, olhamos para seu irmão, que é o filho direito do pai. Se esse irmão for vermelho, trocamos as cores dele e do pai, fazemos uma rotação para a esquerda no pai, e então atualizamos o irmão, porque a situação mudou. Depois disso, se o irmão tem ambos os filhos pretos, pintamos o irmão de vermelho e subimos para o pai, repetindo o processo. Mas, se o irmão tiver pelo menos um filho vermelho, tomamos uma nova decisão: se o filho direito do irmão for preto, primeiro fazemos uma rotação para a direita no irmão, para preparar o cenário, e depois seguimos com a rotação para a esquerda no pai. Durante essas rotações, também ajustamos as cores para restaurar as propriedades da árvore. Se o nó estivesse à direita do pai, o raciocínio é o mesmo, apenas espelhado: analisamos o irmão que está à esquerda e trocamos as rotações de lado. Quando finalmente saímos do loop, garantimos que o nó atual fique pintado de preto, fechando a correção e mantendo todas as propriedades da árvore rubro-negra.

Finalmente, o desenvolvimento do main foi simples: iniciamos os nós NIL, alocamos memória para a árvore e desenvolvemos o sistema de input baseado no enunciado, com "i" sendo o código para inserir e "r" para remover. Após isso, fazemos o nivelamento da árvore e a impressão em ordem, com a chave, o nível e a cor.

**Conclusão:**

Implementar a estrutura de árvores rubro-negras foi um desafio muito interessante, principalmente devido às vastas condições em processos de inserção e remoção. Dentro desse trabalho, podemos entender não só conceitos de árvores rubro-negras, mas também a genialidade por trás de árvores de busca binária e sua implementação. Além disso, visualizamos a diferença entre NULL e NIL dentro dos nós, e como os detalhes dentro de estruturas mudam toda sua implementação. Agradecemos ao professor pelo suporte dado durante o desenvolvimento do trabalho.