UNIVERSIDADE ESTADUAL DO CEARÁ
Curso de Bacharelado em Ciência da Computação
Disciplina: Estrutura de Dados II Prof. Camila Colares 2019.2

Implementação da Árvore Rubro-Negra

Beatriz Bianca Moreira Vasconcelos
beatriz.vasconcelos@aluno.uece.br

Matheus Vieira de Araújo
matheusvieira.11@hotmail.com

Samuel Alcântara Fontenele Rocha
samuel.rocha@aluno.uece.br

Resumo

A Árvore Rubro-Negra é um algoritmo baseado na melhoria do retorno de um a busca em conjunto rico de operações e baseia-se no uso da recursividade para ser funcional. Em uma árvore balanceada, pode-se atingir a melhoria de pior cenário em log(n), em uma árvore de N nós, sendo um dos algoritmos mais eficientes dentre as estruturas de dados que conhecemos.

1 Introdução

Projetada em 1972, pelo professor emérito na Universidade Técnica de Munique, a Árvore rubro negra, ou Árvores B Binária Simétrica (auto-balanceada), popularizou-se nos meados de 1978 por Leonidas J. Guibas e Robert Sedgewick. A árvore rubro-negra consiste em um tipo especial de árvore binária balanceada, para organizar dados que possam ser comparáveis. Nessas comparações, os nós folhas são irrelevantes e não contém dados, porém, necessita-se manter um ponteiro não-nulo para identificá-las. Denomina-se um nó como Nó Sentinela, a fim de economizar memória, que interpretará o papel de todos os nós folha; ou seja, todas as referências

dos nós internos para os nós folham apontam para o nó sentinela. Algumas operações em árvores rubro-negras são simplificadas se os nós folham forem explicitados. Árvore rubro-negras, como todas as árvores de busca binárias, permitem travessia ordenada de elementos de maneira eficiente, ao acessar o pai de qualquer nó.

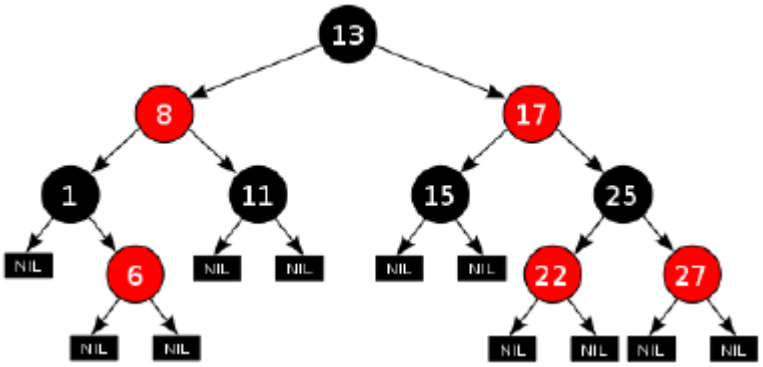


Figura 1 - Exemplificação da Árvore Rubro-Negra

Neste trabalho, realizamos a implementação de uma árvore rubro-negra usando a linguagem C. O código é dividido em 3 partes:

I - ArvRN.c - Contém o menu e a função *main( )*

II - libArvRN.h - Contém a interface da biblioteca criada

III - libArvRN.c - Contém a implementação da biblioteca criada

O trabalho, por sua vez, é dividido em 5 partes: Introdução, Descrição do algoritmo e complexidade, Resultados dos experimentos computacionais, Conclusão e Referências.

2 Descrição do algoritmo e complexidade

Uma árvore de pesquisa binária é uma árvore vermelho-preto se satisfaz às seguintes propriedades vermelho-preto:

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (NULL) é preta.
4. Se um nó é vermelho, então ambos os seus filhos são pretos.
5. Para cada nó, todos os caminhos desde um nó até as folhas descendente contêm o mesmo número de nós pretos.

Utilizamos uma única sentinela para representar NULL. Para uma árvore vermelho-preto T, a sentinela ***nodonull*** é um objeto com os mesmos campos que um nó comum na árvore e, cada nó da árvore contém: cor, chave, pai, esq, dir.

Chamamos o número de nós pretos em qualquer caminho desde um nó x, sem incluir esse nó, até uma folha, de **altura de preto do nó**. Pela propriedade 5, a noção de altura de preto é bem definida, pois todos os caminhos descendentes a partir do nó têm o mesmo número de nós pretos. Definimos a altura de preto de uma árvore vermelho-preto como a altura de preto de sua raiz.

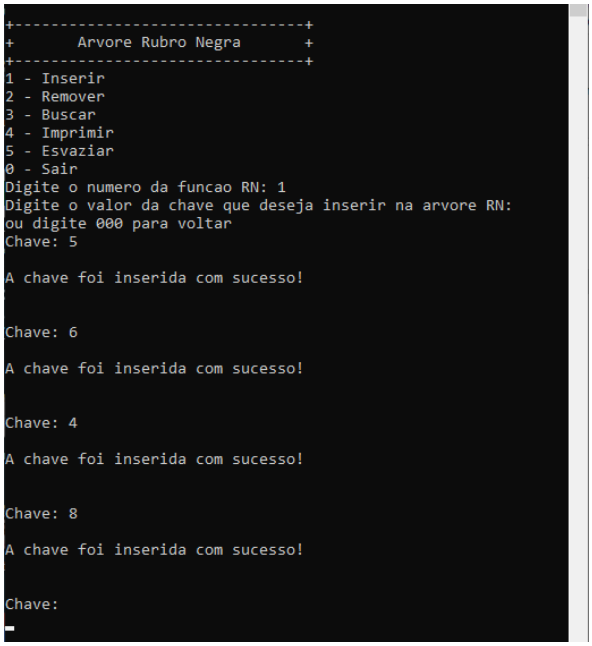
Organizamos a estrutura do programa com um arquivo .C separado com todas as funções e criações de ponteiro necessárias, para facilitar o entendimento do código.

Iniciando pela função de inicializar a árvore com a raiz Null e cor Black, para guardar o primeiro nó da árvore utilizamos um “ponteiro para ponteiro”, o qual pode guardar o endereço de um “ponteiro”. Dessa forma podemos mudar quem é a “raiz” da árvore, caso necessário.

Na função “Main()”, aplicamos um Switch case para referenciar cada função enviando o elemento desejado para a ação selecionada, por exemplo, chamamos a função “Inserir” referenciando o valor que já desejo inserir e a função me retorna à confirmação de sucesso ou fracasso no procedimento. Desta forma sem precisar toda vez que inserir, gerar uma atualização da árvore e poluir o GUI do programa.

Para a estrutura da árvore ser intuitiva, foi um dos principais problemas na hora da construção do programa, tentamos criar uma interface em Java, porém demandou muito tempo, então optamos por trabalhar na interface do console, um dos motivos por procurar sempre a simplicidade, entretanto a funcionalidade coerente, foi visto que o uso dos ponteiros facilita muito nas trocas de elementos da árvore

Inserir: Para inserir fizemos de uma forma simples, através de um valor por vez, assim podemos evitar vários problemas com inconsistência na inserção, além de poupar tempo para posteriores contratempos na hora de desenvolver.



Para inserir, chama-se a função “arvRN_insere()” com seus paramentros.

```
arvRN_inicializa();
op = arvRN_menu ();
while (op != 0) {
    switch (op) {
        case 1: //Inserir
            printf ("Digite o valor da chave que deseja inserir na arvore RN: \nou digite 000 para voltar\n");
            do{
            printf ("Chave: ");
            scanf ("%d", &chave);
            printf ("\n");
            if(chave==000)
                break;
            raiz =  arvRN_insere (raiz, chave);
            printf ("A chave foi inserida com sucesso!\n\n\n");

            }while (chave ==(int)chave);
```

Ao requisitar a função inserir, é feita toda uma consulta de nodos pai para saber se já existe, e assim poder criar e inserir o nodo, sua chave e seus ponteiros para filhos direita e esquerda também. Caso o nodo pai não exista, ele insere o novo nodo como pai, seus filhos como Null e ele se torna a raiz da arvore pois ela estaria vazia.

```
/*--------------------------------------------------------------------------*/

/*--------------------------------------------------------------------------*/
/* Insere uma chave na arvore RN                                            */
/*--------------------------------------------------------------------------*/
Apontador arvRN_insere (Apontador p, int k) {
    Apontador novo = arvRN_criaNodo (k, nodonull, nodonull, nodonull);

    Apontador x = p;
    Apontador paix = nodonull;

    while (x != nodonull){ /*busca o pai do nodo novo*/
        paix = x;
        if (k < x->chave)
            x = x->esq;
        else
            x = x->dir;
    }
    novo->pai = paix;
    if (paix == nodonull) /*arvore vazia*/
        p = novo;
    else if (k < paix->chave)
        paix->esq = novo;
    else
        paix->dir = novo;
    novo->cor = RED;
    p =  arvRN_arrumaInsere (p, novo);
    return p;
}
```

Após inserir é chamado a função para arrumar a arvore, “arvRN_arrumaInsere()”, a qual ira analisar todos os casos da situação daquele nodo e poder selecionar sua cor correta e alterar seus ponteiros.

Remover: Talvez a parte mais difícil do código, tivemos bastante dificuldade para conseguir entender o funcionamento para fazer o desenvolvimento. Da forma que fizemos a função remover, é realizada a consulta verificando se o valor existe na arvore, se sim, a função “arvRN_remove()” realiza as consultas de filhos. Através dela existem varias condições que condicionam a ação de remover para a forma correta da arvore, por exemplo:

```
/*--------------------------------------------------------------------------*/
/* Remove um nodo da arvore RN                                              */
/*--------------------------------------------------------------------------*/
void arvRN_remove (Apontador r, Apontador nodok) {
    Apontador nodoRem, filho;
    if (nodok == nodonull)
        return;
    if (nodok->esq == nodonull || nodok->dir == nodonull)
    /* se nodok tem 0 ou 1 filho, remove nodok */
        nodoRem = nodok;
    else                              /* senão remove o Sucessor */
        nodoRem = arvRN_sucessor (nodok); /* neste caso o nodoRem não tem filho esq */
```

Essa condição para remover o elemento desejado verifica se o elemento buscado existe na arvore, e logo após já verifica se seus filhos são nulos para assim poder remover.

Buscar: A busca foi a parte mais simples, caso o elemento a ser buscado fosse encontrado na árvore, a função retorna o próprio elemento achado como resposta.

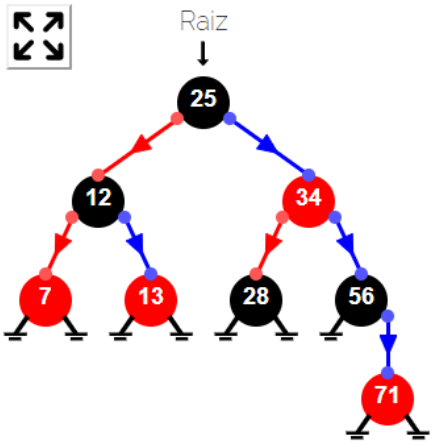
Imprimir: Como foi dito acima, a função imprimir separada das demais funções serviu como uma forma de organizar tanto a estrutura do código como também para deixar o console de execução mais limpo, podendo analisar de forma fácil cada processo.

As operações sobre árvores de pesquisa **arvRN_insere** e **arvRN_remove**, quando executadas sobre uma árvore vermelho-preto com n chaves demoram o tempo O (lg n). Considerando-se que elas modificam a árvore, o resultado pode violar as propriedades vermelho-preto enumeradas acima. Para restabelecer essas propriedades, devemos mudar as cores de alguns nós na árvore e também mudar a estrutura de ponteiros, mudamos a estrutura de ponteiros através de rotação.

3 Resultados dos experimentos computacionais

Entrada: 34, 25, 12, 7, 56, 13, 28, 71

Saída esperada:



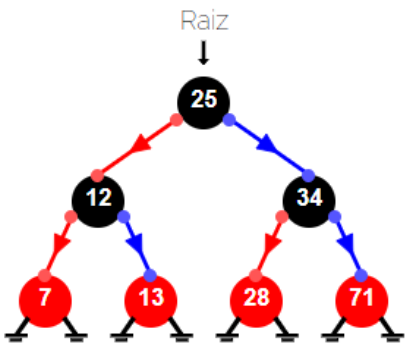
Saída:

```
+--------------------------------+
+            Imprimir            +
+       Arvore Rubro Negra       +
+--------------------------------+
1 - Pre-ordem
2 - Em ordem
3 - Pos-ordem
4 - Voltar
Digite o numero da funcao RN: 3
Chave: 7 | Cor: RED
Chave: 13 | Cor: RED
Chave: 12 | Cor: BLACK
Chave: 28 | Cor: BLACK
Chave: 71 | Cor: RED
Chave: 56 | Cor: BLACK
Chave: 34 | Cor: RED
Chave: 25 | Cor: BLACK
```

A saída foi igual a saída esperada!

Realizando a remoção: 56

Saída esperada:



Saida:

```
+--------------------------------+
+            Imprimir            +
+       Arvore Rubro Negra       +
+--------------------------------+
1 - Pre-ordem
2 - Em ordem
3 - Pos-ordem
4 - Voltar
Digite o numero da funcao RN: 3
Chave: 7 | Cor: RED
Chave: 13 | Cor: RED
Chave: 12 | Cor: BLACK
Chave: 28 | Cor: BLACK
Chave: 71 | Cor: RED
Chave: 34 | Cor: RED
Chave: 25 | Cor: BLACK
```

Novamente obtivemos sucesso no resultado

4 Conclusão

Árvore Rubro-Negra (ARN) é uma árvore binária balanceada que usa um método de busca com quatro premissas, nas quais dividem a árvore composta por pais negros, filhos vermelhos e folhas descartáveis. Vimos neste trabalho inserção, remoção, busca e impressão da ARN, também vimos que sua complexidade é O (log n), sendo uma das mais velozes estruturas de dado. Verificamos também que ponteiros e recursividade são essenciais para melhor otimização do algoritmo. Por fim, concluímos que a Árvore Rubro-Negra é uma estrutura que cumpre seu papel de forma eficiente e eficaz.

Referências

[ED] Aula 105 - Árvore Rubro Negra – Playlist completa
https://www.youtube.com/watch?v=DaWNuijRRFY&list=PL8iN9FQ7_jt6XjYc0H01AVJoCePsiSNEq

ÁRVORE RUBRO- NEGRA. Prof. André Backes

https://docplayer.com.br/37848956-Arvore-rubro-negra-prof-andre-backes-tambem-conhecida-como-arvore-vermelhopreto.html

Simulação de Árvore Rubro Negra

https://portaldoprofessor.fct.unesp.br/projetos/cadilag/apps/structs/arv_rn.php