Árvore Rubro Negra

Sávio Rodrigues

[1]Engenharia da Computação
Campus Divinópolis CEFET-MG

1. Introdução

Árvore rubro negra, também conhecida como vermelho-preto ou red-back é um tipo de árvore balanceada que foi originalmente criada por Rudolf Bayer em 1972. A estrutura usa um esquema de coloração dos nós para manter o balanceamento da árvore.

Em uma aŕvore Rubro-Negra, os nós possuem identificação de cor através de um bit, contudo dispõe da mesma complexidade computacional que a árvore AVL. Entretanto, árvores rubro-negras possuem operações de remoção e inserção mais rápida, ao mesmo tempo que é mais lenta ao buscar algum dado. Após o uso de métodos de inserção ou remoção, é feita a análise das cores a fim de encontrar desbalanceamento e realizar as rotações. A estrutura possui além do bit extra os seguintes campos: Valor, Filho esquerdo e direito.

Uma árvore Vermelha-Preta precisa obedecer as seguintes propriedades:

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 os seus filhos são pretos.
5. Para cada nó, todos os caminhos do nó para folhas descendentes contém o mesmo número de nós pretos.

Obedecendo as propriedades enumeradas, é possível concluir que em um caminho de raiz até uma sub-árvore vazia, não pode existir dois rubros consecutivos. A altura de um nó nas árvores rubro-negras é calculada através da quantidade de nós pretos que o caminho entre o nó e sua folha descendentes possui (excluindo os nós nulos).

2. Balanceamento

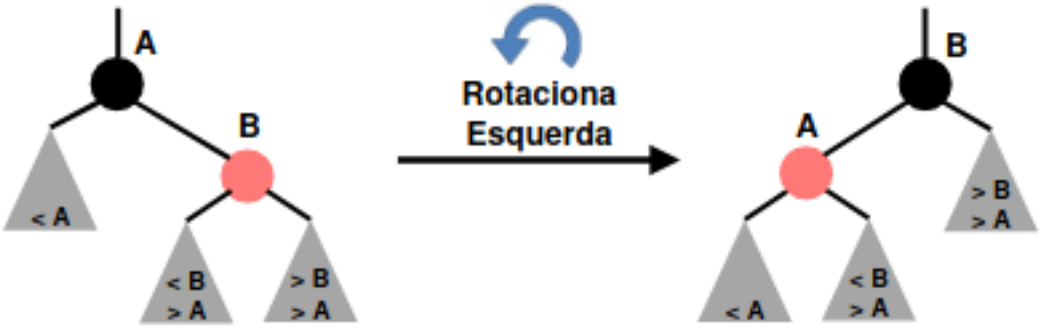
O balanceamento é feito por meio de rotações e ajuste de cores a cada inserção e remoção. Nesse sentido, o intuito é obedecer as propriedades de uma árvore vermelho-preta, mantendo o equilíbrio e possuindo um custo máximo de *O(log N)*.

2.1. Rotações

Diferente de árvores AVL que possui quatro funções de rebalancear, a rubro-negra possui apenas duas funções de rotação, sendo elas: rotação para a direita e rotação para a esquerda.

2.1.1. Rotação para a esquerda

A função de rotação para a esquerda recebe um nó A com B como filho **direito**. Depois move B para o lugar de A, A se torna o filho **esquerdo** de B e por fim, B recebe a cor de A, A fica vermelho.

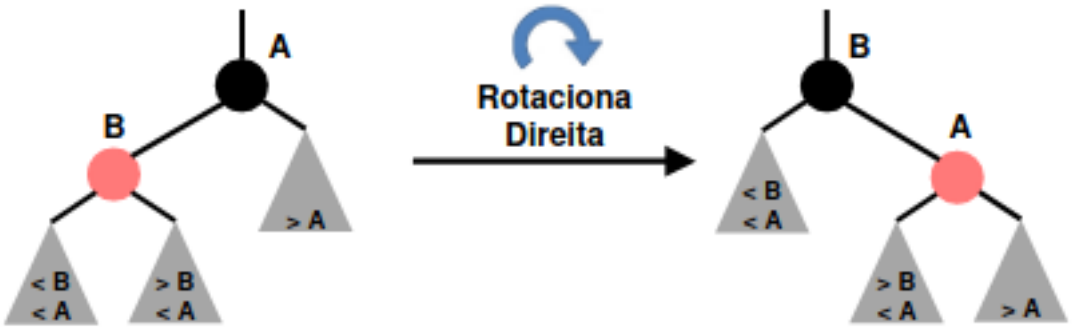



```
void rotacaoSimplesEsquerda(Tree **t){
    Tree *aux;
    aux = (*t)->dir;
    (*t)->dir = aux->esq;
    aux->esq = (*t);
    aux->cor = (*t)->cor;
    (*t)->cor = 1;
    (*t) = aux;
}
```

Obs: A fim de estabelecer uma relação entre as duas imagens, define-se: *t = A, aux = B e cor sendo um valor inteiro: 1 (vemelho), 0 (preto)

2.1.2. Rotação para a direita

A função de rotação para a esquerda recebe um nó A com B como filho **esquerdo**. Depois move B para o lugar de A, A se torna o filho **direito** de B e por fim, B recebe a cor de A, A fica vermelho.



```c
void rotacaoSimplesDireita(Tree **t){
    Tree *aux;
    aux = (*t)->esq;
    (*t)->esq = aux->dir;
    aux->dir = (*t);
    aux->cor = (*t)->cor;
    (*t)->cor = 1
    (*t) = aux;
}
```

Obs: A fim de estabelecer uma relação entre as duas imagens, define-se: *t = A, aux = B e cor sendo um valor inteiro: 1 (vemelho), 0 (preto)

3. Inserção

A inserção é bastante similar à árvore AVL, porém é necessário inserir respeitando as propriedades da estrutura rubro-negra, sendo:

- Se a raiz é igual a **NULL**, insira um nó **Preto**
- Se a raiz é diferente de **NULL**, o nó inserido possui cor **Vermelho**

Se o valor a ser inserido é maior que a raiz vá para a sub-árvore á direita, se for menor, vá para esquerda. Esse procedimento repetirá recursivamente até que se chegue nó folha, o qual se tornará pai do novo nó.

Uma vez inserido o novo elemento, é necessário percorrer o caminho contrário analisando se não houve a violação de nenhuma propriedade da árvore. Identificado algum erro, é necessário realizar rotações para reestabelecer o balanceamento da árvore. Há 3 tipos de violações das propriedades na inserção:

- Filho da direita é **Vermelho** e o filho da esquerda é **Preto**: Resolvido com uma rotação á esquerda.
- Filho da esquerda é **Vermelho** e o filho á esquerda é **Vermelho**: Resolvido com uma rotação á direita.
- Ambos os filhos são vermelhos: Resolvido com a troca de cores (ambos os filhos se tronam pretos e o pai vermelho)

A seguir têm-se a implementação do código de inserção da árvore Rubro-Negra.

```c
void insert(Tree **t, Record r){
  if(*t == NULL){
    *t = (Tree*)malloc(sizeof(Tree));
    (*t)->esq    = NULL;
    (*t)->dir    = NULL;
    (*t)->cor = 1;
    (*t)->reg    = r;
    actions++;
    printf("[%d]%d adicionado.\n", actions, r.key);
  } else {
    if(r.key == (*t)->reg.key){
        printf("Valor Duplicado!\n");
        return;
    }else {
      if(r.key < (*t)->reg.key){
        insert(&(*t)->esq, r);

      }else{
        insert(&(*t)->dir, r);
      }
    }
    if(AcessarCor(&(*t)->dir) == 1 && AcessarCor(&(*t)->esq) == 0){
      actions++;
      rotacaoSimplesEsquerda(t);
      printf("[%d]Rotação para a esquerda.\n", actions);
    }
    if(AcessarCor(&(*t)->dir) == 1 && AcessarCor(&(*t)->esq->esq) == 1){
      rotacaoSimplesDireita(t);
      printf("[%d]Rotação para a direita.\n", actions);
    }
    if(AcessarCor(&(*t)->dir) == 1 && AcessarCor(&(*t)->esq) == 1){
      trocaCor(t);
      printf("[%d]Troca de cor (%d <- %d -> %d).\n", actions, (*t)->esq->reg.key,(*t)->reg.key,(*t)->dir->reg.key);
    }
  }
}
```

4. Remoção

Para remover algum dado, é necessário percorrer a árvore no intuito de encontrar o nó que será removido. Se ele existir a remoção pode ser realizada, podendo variar em 3 tipos:

- Nó folha (Sem filhos).
- Nó com 1 filho.
- Nó com 2 filhos.

Uma vez removido o elemento, é preciso percorrer o caminho contrário analisando se não houve a violação de nenhuma propriedade da árvore. Identificado algum erro, é necessário realizar rotações para reestabelecer o balanceamento da árvore.

- Filho da direita é **Vermelho**: Resolvido com uma rotação á esquerda.
- Filho da esquerda e neto da esquerda são **Vermelhos**: Resolvido com uma rotação á direita.
- Ambos os filhos são vermelhos: Resolvido com a troca de cores (ambos os filhos se tronam pretos e o pai vermelho)

A implementação do código de remoção e suas funções auxiliares são ilustrado a seguir:

```c
void rebalanceTree(Tree **t){
    //printf("PAssou aqui\n");
    //nó vermelho é sempre filho à esquerda
    if(AcessarCor(&(*t)->dir)==1)
        rotacaoSimplesEsquerda(t);
    //Filho da esquerda e neto da esquerda são vermelhos
    if( (*t)->esq != NULL && AcessarCor(&(*t)->esq) == 1 && AcessarCor(&(*t)->esq->esq) == 1)
        rotacaoSimplesDireita(t);
    //2 Filhos Vermelhos
    if(AcessarCor(&(*t)->esq)== 1 && AcessarCor(&(*t)->dir)== 1)
        trocaCor(t);
}
void mover2EsqRED(Tree **t){
    trocaCor(t);
    if(AcessarCor(&(*t)->dir->esq) == 1){
        rotacaoSimplesDireita(&(*t)->dir);
        rotacaoSimplesEsquerda(t);
        trocaCor(t);
    }
}
void mover2DirRED(Tree **t){
    trocaCor(t);
    if(AcessarCor(&(*t)->esq->esq) == 1){
        rotacaoSimplesDireita(t);
        trocaCor(t);
    }
}

void removerMenor(Tree **t){
    if((*t)->esq == NULL){
        free(t);
        return;
    }
    if(AcessarCor(&(*t)->esq) == 0 && AcessarCor(&(*t)->esq->esq) == 0)
        mover2EsqRED(t);

    removerMenor(&(*t)->esq);
    rebalanceTree(t);
}
```

```c
void remover(Tree **t, Record r){
    if(r.key < (*t)->reg.key){
        if(AcessarCor(&(*t)->esq) == 0 && AcessarCor(&(*t)->esq->esq)== 0)
            mover2EsqRED(t);
        remover(&(*t)->esq,r);
    }else{
        if(AcessarCor(&(*t)->esq) == 1)
            rotacaoSimplesDireita(t);

        if(r.key == (*t)->reg.key && ((*t)->dir == NULL)){
            free(*t);
            return;
        }
        if(AcessarCor(&(*t)->dir) == 0 && AcessarCor(&(*t)->dir->esq) == 0){
            mover2DirRED(t);
        }
        if(r.key == (*t)->reg.key){
            Tree *aux = procurarMenor(&(*t)->dir);
            (*t)->reg.key = aux->reg.key;
            removerMenor(&(*t)->dir);
        }else{
            remover(&(*t)->dir, r);
        }
    }
    rebalanceTree(t);
}
```

A função *remover()* procura recursivamente com o auxilio de funções, o nó a ser removido. Quando encontrado, é realizado um *free()* e um balanceamento é feito através das funções *move2EsqRED()* e *move2DirRED()*, a fim de organizar a arvore novamente e executando as rotações necessárias.

5. Execução

A implementação contém as seguintes linguagens de programação: C e C++ para a codificação do código fonte e criar/importar bibliotecas respectivamente. Nesse sentido o compilador é o GCC e a linha de execução do algoritmo é

gcc Main -o Main.c

Atenção: Configurações do sistema operacional utilizado pra a criação do código: Ubuntu 20.04.2 LTS, versão do GCC: gcc 9.3.0.

- A primeira parte da execução representa ações do código sobre a árvore.

```
ELEMENTOS A SEREM ADICIONADOS: 11  2  1  5  4  7  8  13  15  14
[1]11 adicionado.
[2]2 adicionado.
[3]1 adicionado.
[4]5 adicionado.
[4]Troca de cor (1 <- 2 -> 5).
[5]4 adicionado.
[5]Troca de cor (1 <- 2 -> 5).
[6]7 adicionado.
[6]Troca de cor (4 <- 5 -> 7).
[7]Rotação para a esquerda.
[8]8 adicionado.
[9]Rotação para a esquerda.
[10]13 adicionado.
[10]Rotação para a direita.
[11]15 adicionado.
[12]Rotação para a esquerda.
[13]Rotação para a esquerda.
[14]14 adicionado.
[15]Rotação para a esquerda.
[16]Rotação para a esquerda.
```

- A segunda parte da execução apresenta os elementos e suas respectivas cores e a arvore após inserções.

```
        ORDEM CRESCENTE
1 Preto 2 Vermelho 4 Preto 5 Preto 7 Vermelho 8 Preto 11 Vermelho 13 Vermelho 14 Vermelho 15 Preto
---------------

         15

                  14

                                    13

                           11

                                    8

                                             7

5

                  4

         2

                  1

-------------------
Raiz : 5 (Preto)        Esquerda : 2(Vermelho)  Direita : 15 (Preto)
```

- A terceira parte da execução apresenta a remoção de um elemento folha (7).

```
        APÓS REMOÇÃO 7(Folha):
 1 Preto 2 Vermelho 4 Preto 5 Preto 8 Preto 11 Vermelho 13 Vermelho 14 Vermelho 15 Preto
 -------------------

          15

                   14

                                     13

                            11

                                     8

 5

                   4

          2

                   1

 -------------------
 Raiz: 5         Esquerda: 2     Direita: 15
 5
 15
```

- A quarta parte da execução apresenta a remoção dos elementos intermediários (14, 11, 13).

```
         APÓS REMOÇÃO 14 11 13:
 1 Preto 2 Vermelho 4 Preto 5 Preto 8 Preto 15 Preto
 --------------------

           15

                     8

 5

                     4

           2

                     1
```

6. Referências

https://www.ime.usp.br/ song/mac5710/slides/08rb.pdf
http://www.facom.ufu.br/ backes/gsi011/Aula12-ArvoreRB.pdf
https://pt.wikipedia.org/wiki/
T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT Press McGraw-Hill, 2001.
https://programacaodescomplicada.wordpress.com/