Arvore Rubro Negra,
Aplicação: sistema de busca inteligente

Lucas Alves Da Costa.

[1]*LucasAlves16.LA@gmail.com*

[2]Instituto federal de educação, ciência e tecnologia-Goiás
(IFG)
Matricula: *20151070130183.*

Resumo. *Este trabalho tem como proposito mostrar uma aplicação que foi criada utilizando a estrutura de dados;* ***Arvore rubro-negra****. a linguagem utilizada foi a linguagem C, cada valor inserido será calculado uma posição especifica para o mesmo, podendo assim busca-lo ou remove-lo posteriormente obedecendo regras e limitações da estrutura. tudo feito será em tempo de execução pois a aplicação se limita a uma possivel conexão com banco de dados ou sistema de arquivos de texto.*

1. Arvore rubro-negra

Uma árvore rubro-negra ou arvore red-black é um tipo de árvore binária de busca balanceada, ela é complexa, mas tem um bom pior-caso de tempo de execução para suas operações e é eficiente na prática: pode-se buscar, inserir, e remover em tempo **O(log n)**, onde **n** é o número total de elementos da árvore. De maneira simplificada, uma árvore rubro-negra é uma árvore de busca binária que insere e remove de forma inteligente, para assegurar que a árvore permaneça aproximadamente balanceada, cada nó tem um atributo de cor, vermelho ou preto. Além dos requisitos ordinários impostos pelas árvores de busca binárias, as árvores rubro-negras tem os seguintes requisitos adicionais:
1. Um nó é vermelho ou preto.
2. A raiz é preta. (Esta regra é usada em algumas definições. Como a raiz pode sempre ser alterada de vermelho para preto, mas não sendo válido o oposto, esta regra tem pouco efeito na análise.)
3. Todas as folhas(nil) são pretas.
4. Ambos os filhos de todos os nós vermelhos são pretos.
5. Todo caminho de um dado nó para qualquer de seus nós folhas descendentes contem o mesmo número de nós pretos.

Essas regras asseguram uma propriedade crítica das árvores rubro-negras: que o caminho mais longo da raiz a qualquer folha não seja mais do que duas vezes o caminho mais curto da raiz a qualquer outra folha naquela árvore. O resultado é que a árvore é aproximadamente balanceada. Como as operações de inserção, remoção, e busca de valores necessitam de tempo de pior caso proporcional à altura da árvore, este limite proporcional a altura permite que árvores rubro-negras sejam eficientes no pior caso, diferentemente de árvore de busca binária. A cada vez que uma operação é realizada na árvore, testa-se este conjunto de propriedades e são efetuadas rotações e ajuste de cores até que a árvore satisfaça todas estas regras. Uma rotação é uma operação realizada na árvore para garantir

seu balanceamento. A rotação na árvore rubro-negra pode ser feita à direita e à esquerda, onde são alterados os ponteiros dos nós rotacionados. Ao inserir-se um elemento em uma árvore rubro-negra, esta é comparada com os elementos e alocada em sua posição conforme a regra 2. Ao inserir-se um elemento ele é sempre da cor vermelha (exceto se for o nó raiz). A seguir a árvore analisa o antecessor da folha. Se este for vermelho será necessário alterar as cores para garantir a regra 4. Uma inserção começa pela adição de um nó de forma semelhante a uma inserção em árvore binária, pintando-se o nó em vermelho. Diferentemente de uma árvore binária onde sempre adicionamos uma folha, na árvore rubro-negra as folhas não contém informação, então inserimos um nó vermelho na parte interna da árvore, com dois nós filhos pretos, no lugar de uma folha preta.

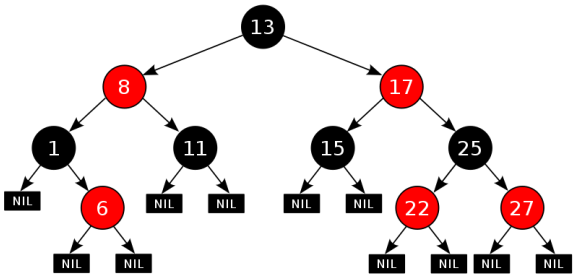

Figura 1- Arvore rubro-negra

Implementação: A aplicação foi feita visando os conceitos de arvore rubro-negra, podendo inserir, remover ou buscar por um elemento, sendo possível também a impressão dos valores inseridos e a destruição da árvore. Abaixo etá a parte da aplicação onde a inserção é implementada.

```
1
2  int insereArvLLRB(ArvLLRB* raiz, int valor, string string){
3      int resp;
4      *raiz= insereNO((NO*) *raiz,valor,&resp, string);
5          if((*raiz) != NULL){
6              (*raiz)->cor=BLACK;
7          }
8           return resp;
9  }
10
11 struct NO* insereNO(struct NO* H, int valor, int *resp, string
       str){
12     if(H == NULL){
13     struct NO *novo;
14     novo = (struct NO*)malloc(sizeof(struct NO));
15       if(novo == NULL){
16         *resp = 0;
17         return;
18       }
19       novo->string.chave = str.chave;
20       novo->string.texto = str.texto;
21       novo->inf = valor;
22       novo->cor = RED;
23       novo->dir =NULL;
```

```
      novo->esq =NULL;
      *resp = 1;
      return novo;
     // free(novo);
    }
     if(valor == H->inf)
        *resp = 0;
     else{
           if(valor < H->inf)
              H->esq = insereNO(H->esq,valor,resp,str);
           else
               H->dir = insereNO(H->dir,valor,resp,str);

     }
        if(cor(H->dir) == RED && cor(H->esq) == BLACK){
          H = rotacionaEsquerda(H);
          //printf("\nRotaciona_Esquerda!");
        }
         if(cor(H->esq) == RED && cor(H->esq->esq) == RED){
            H = rotacionaDireita(H);
            //printf("\nRotaciona_Direita!");
         }
          if(cor(H->esq) == RED && cor(H->dir) == RED){
             trocarCor(H);
             printf("\ncor trocada!!");
          }
            return H;
 }
```

2. Inserção

A função de inserção é semelhante as usadas pelas outras arvores de busca, a diferença está no balanceamento e na troca de cores, já que a árvore é balanceada assim que um valor é inserido. Ao voltar da recursão da inserção é verificado as propriedades da árvore e rotações e ou troca de cores são feitas caso alguma das regras ou propriedades da árvore sejam violadas. Note que usamos duas funções para inserção, mas apenas uma realmente insere, a função **insereARVLLRB()** serve apenas para gerenciamento da inserção. Note também que na assinatura das funções é passado como argumento uma estrutura do tipo ***String*** *string* onde é guardado uma palavra chave e um texto correspondente a essa palavra, o intuito aqui é inserir na arvore além de um valor chave, também uma estrutura contendo informações em texto. esse valor chave é inserido de acordo com o a chave de texto que foi inserida, isso é feito através de uma função que calcula todos os caracteres da palavra chave que foi digitada pelo usuário, isso irá se resultar em um numero inteiro, esse numero que será guardado como chave de busca na estrutura.

Veja abaixo como é feito o calculo da chave de busca de uma palavra chave qualquer.

```

void menu(){
    int opc,i,valor;
    string string;
```

```
5       char str[50],str2[200];
6       printf("Escolha:\n1-iserir\n2-remover\n3-imprimir Arvore\n4-
           simular busca\n5-|Destruir Arvore|\nZERO to EXIT\n=> ");
7       scanf("%d",&opc);
8
9        switch(opc){
10       case(0):
11        printf("\n\t\t<<<|Aplicacao encerrada|>>>");
12        exit(1);
13       break;
14
15          case(1):
16               system("clear");
17               printf("\n---=== inserir ===---");
18               setbuf(stdin,NULL);
19               printf("\nDigite a palavra chave: ");
20               scanf("%[^\n]s",str);
21                 getchar();
22               printf("Digite a inf. desta chave: ");
23               scanf("%[^\n]s",str2);
24
25               valor = key_insert(str);
26               string.chave = str;
27               string.texto = str2;
28
29               i = insereArvLLRB(&raiza,valor,string);
30
31               if(i == 1) printf("\n\t\t insert in key ->%d!\n",
                    valor);
32               else printf("\n\tNao foi possivel inserir o valor
                    :[%d]\n",valor);
33               menu();
34          break;
```

Esta parte se refere apenas ao menu onde é inserido primeiramente a palavra chave, apôs isso é calculada uma chave de inserção para a mesma.próximo trecho de código refere-se a função que calcula a chave para essa palavra, texto ou letra digitada.

```
1   int key_insert(char *string){
2       char str[5];
3        int i;
4        int key =0;
5
6               if(strlen(string) <= 3){
7                strncpy(str,string,2);
8                key = str[0]+str[1]-str[2];
9               }else{
10                  for(i=0; i<strlen(string); i++){
11                      key += string[i];
12                      if(i == strlen(string)-1)
```

```
                    key -= string[i];
                }

            }

    printf("Valor: %d\n",key);
    int size = strlen(string);
    printf("tamanho: %d\n",size);

    for(i=0; i<size; i++){
       printf("%c = %d\n",string[i],string[i]*1);
    }
    return key;
}
```

No mesmo nó da arvore que foi inserido a chave gerada pela função; **key insert** é inserido também a estrutura de texto correspondente que foi digitada pelo usuário, ou seja, a palavra chave e o texto correspondente a ela. Visando essa ideia já podemos entender o funcionamento da aplicação que nada mais é do que um sistema de busca onde digitamos por uma palavra especifica como: *Gato* e o resultado da busca seria: *Gato é um felino doméstico*, para melhor intendermos, pense no sistema de pesquisa da **Google**, quando pesquisamos por alguma palavra chave temos por muitas vezes como resultado várias informações, essa seria a ideia por traz desta aplicação.
Logo a baixo está a parte onde podemos simular uma busca, considerando que a árvore já esteja preenchida com algumas informações.

```
  case(4):
     printf("Pesquise por algo: ");
             setbuf(stdin,NULL);
      scanf("%[^\n]s",str);
        valor = key_insert(str);
        NO *aux = consultaArvLLRB(raiza, valor);
        if(aux != NULL){
            printf("\n\t\tBusca:(%s)\nResultado:\n\t[%s]\n",aux
               ->string.chave,aux->string.texto);
        }else
            printf("\nNada encontrado!!\n\n");
             menu();
    break;
```

Note que inserimos apenas a palavra chave, e a partir desta palavra é calculada a sua posição de destino. Caso esse valor exista dentro da arvore é retornado o texto ou informação referente a quela palavra(chave de busca) digitada anteriormente.

3. Remoção

A remoção assim como a inserção, também é uma parte complexa. Quando um valor é removido da arvore a mesma pode ficar desbalanceada ou alguma(s) das propriedades descritas anteriormente podem ser violada(s), para esse caso são feitas rotações e trocas de cores entre os nós para que a arvore continue aproximadamente balanceada e

obedecendo os critérios estabelecidos para obter uma arvore rubro-negra. A lógica utilizada na remoção é semelhante a usada nas outras arvores binarias de busca como a *AVL*, a diferença está no balanceamento que é feito juntamente com a remoção de um valor, isso assegura que todas as propriedades da arvore continuem válidas até que outro valor possa ser removido ou até mesmo inserido. Veja abaixo a função de remoção utilizada na aplicação.

```
 int removeArvLLRB(ArvLLRB* raiz, int valor){
    if(consulta(*raiz,valor)){
        struct NO* h = *raiz;

         (*raiz)= removeNO(h,valor);
          if(*raiz != NULL)
             (*raiz)->cor = BLACK;
         return 1;
    }else{
          return 0;
    }
}

struct NO* removeNO(struct NO* H, int valor){
    if(valor < H->inf){
        if(cor(H->esq) == BLACK && cor(H->esq->esq) == BLACK)
         H = move2EsqRed(H);

        H->esq = removeNO(H->esq,valor);
    }else{
         if(cor(H->esq)== RED)
           H = rotacionaDireita(H);

         if(valor == H->inf && H->dir == NULL){
            free(H);
            return NULL;
         }

          if(cor(H->dir)== BLACK && cor(H->dir->esq) == BLACK)
             H = move2DirRed(H);

          if(valor == H->inf){
            struct NO* x = procuraMenor(H->dir);
            H->inf = x->inf;
            H->dir = removerMenor(H->dir);
          }else
               H->dir = removeNO(H->dir,valor);
    }
    return balancear(H);
}
```

Primeiro verifica-se se o valor a ser removido está na arvore, para isso é utilizada uma função de consulta para verificar a existência do valor a ser removido. caso o valor não

seja encontrado na arvore não será possível a remoção do mesmo. A remoção aqui funciona no mesmo intuito da inserção, bastando digitar a palavra chave do nó que será removido. essa palavra chave será calculada para um valor numérico com a função **insert key** que por sua vez será usado para remover o nó que contém esse valor que foi calculado a partir da palavra chave, caso exista. Segue a baixo como o valor é removido.

```
1   case(2):
2                       printf("\n====== REMOVER ======");
3                        printf("\nDigite a palavra chave do
                            conteudo: ");
4                        setbuf(stdin,NULL);
5                        scanf("%[^\n]s",str);
6                        valor = key_insert(str);
7                        i = removeArvLLRB(&raiza,valor);
8
9                        if(i==1) printf("\n\tvalor da chave->[%d]
                            removido!!\n",valor);
10                       else printf("\n\t nao foi possivel remover
                             o valor[%d]\t--Verifique se o valor
                            existe na arvore!\n",valor);
11                       menu();
12                  break;
```

As funções de impressão e destruição são as mesmas utilizadas em qualquer outro tipo de arvore binaria de busca, por este motivo as considerei irrelevantes para este relatório.

4. Considerações finais

Foi mostrado neste relatório o funcionamento de uma aplicação baseada na estrutura de dados; *Arvore rubro-negra*, optei por essa estrutura pois posso inserir e remover valores de forma inteligente, ao invés de inserir elementos de texto em forma caracteres, faço a inserção utilizando esse mesmo elemento de texto convertido para um valor numérico inteiro. Cada caractere tem um valor numérico inteiro que o representa na tabela ASCI, calculando esses valores fica mais fácil fazer comparações de chaves para inserir, remover ou busca-los em uma arvore.

5. Referências

SlideShare
Comunidade do Hardware
Red Black Trees
Wikipedia/Arvores rubro-negras
Programação descomplicada
Red Black Tree MIT/GNU
Árvore Rubro-Negra Siang Wun Song - Universidade de São Paulo - IME/USP