UFAM

FT – FACULDADE DE TECNOLOGIA
ENGENHARIA DA COMPUTAÇÃO

JEAN CLEISON BRAGA GUIMARÃES – 21601227

TRABALHO FINAL: ÁRVORE RUBRO-NEGRA INSERÇÃO, REMOÇÃO E BUSCA

MANAUS, AM
2019

JEAN CLEISON BRAGA GUIMARÃES

TRABALHO FINAL: ÁRVORE RUBRO-NEGRA INSERÇÃO, REMOÇÃO E BUSCA

O trabalho foi solicitado pelo professor de Algoritmos e Estrutura de Dados II, Edson Nascimento para obtenção de nota final por parte dos alunos no primeiro semestre de 2019.

MANAUS, AM
2019

Introdução

Uma árvore rubro-negra é um tipo de árvore binária de busca balanceada, uma estrutura de dados usada em ciência da computação, tipicamente para implementar vetores associativos. A estrutura original foi inventada em 1972, por Rudolf Bayer que a chamou de "Árvores Binárias B Simétricas", mas adquiriu este nome moderno em um artigo de 1978 escrito por Leonidas J. Guibas e Robert Sedgewick. 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.

Nas árvores rubro-negras, os nós folhas não são relevantes e não contém dados. Estas folhas não precisam ser mantidas em memória de computador - basta apenas um ponteiro para nulo para identificá-las — mas algumas operações em árvores rubro-negras são simplificadas se os nós folha forem explicitados. Para economizar memória, algumas vezes um nó sentinela interpreta o papel de todos os nós folha; todas as referências dos nós internos para os nós folha apontam para o nó sentinela.

Além dos requisitos ordinários impostos pelas árvores de busca binárias, as árvores rubro-negras têm 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 contém 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 à altura permite que árvores rubro-negras sejam eficientes no pior caso, diferentemente de árvore de busca binária.

Para perceber por que essas propriedades garantem isto, basta observar que nenhum caminho pode ter dois nós vermelhos sucessivos, devido à propriedade 4. O caminho mais curto possível tem todos os nós pretos, e o caminho mais longo possível alterna

entre nós vermelhos e pretos. Desde que todos os caminhos máximos têm o mesmo número de nós pretos, pela propriedade 5, isto mostra que nenhum caminho é mais do que duas vezes mais longo que qualquer outro caminho.

Em muitas das representações de estruturas de dados em árvore, é possível para um nó pai ter só um filho, e os nós folha contêm dados. É possível representar árvores rubro-negras neste paradigma, mas ele modifica várias propriedades e deixa os algoritmos mais complexos. Por essa razão, este artigo usa "folhas nulas", que não contêm nenhum dado e simplesmente servem para indicar onde a árvore termina, como mostrado acima. Esses nós muitas vezes são omitidos dos desenhos, resultando em uma árvore que parece contradizer os princípios acima mencionados, mas que de fato não faz. Uma consequência disto é que todo nó interno (não-folha) têm dois filhos, embora um ou ambos filhos possam ser folhas nulas.

Alguns explicam uma árvore rubro-negra como uma árvore de busca binária cujas bordas, em vez de nós, são coloridas em vermelho ou preto, mas isto não faz nenhuma diferença. A cor de um nó na terminologia deste artigo corresponde à cor da borda que une o nó ao seu pai, exceto que o nó de raiz é sempre preto (propriedade 2) ao passo que a borda correspondente não existe.

Implementação

- O programa foi construído para fazer inserção, remoção e busca em uma árvore rubro-negra.

Inserção

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.

O que acontece a seguir depende da cor dos nós vizinhos. O termo nó tio será usado para referenciar o irmão de um nó pái, como em uma árvore genealógica. Note que a:

- Propriedade 3 (todos as folhas são pretas) sempre se mantém;
- Propriedade 4 (ambos os filhos de um nó vermelho são pretos) é tratada somente pela adição de um nó vermelho, repintura de um nó preto como vermelho ou uma rotação;
- Propriedade 5 (todos os caminhos de um dado nó até suas folhas contêm o mesmo número de nós pretos) é tratada apenas pela adição de um nó preto, repintura de um nó vermelho em preto ou uma rotação.

Remoção

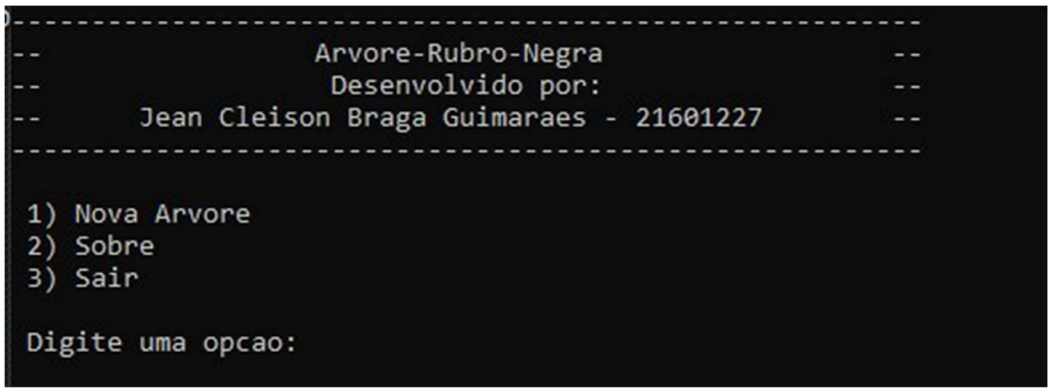
A remoção nas árvores rubro-negras se inicia com uma etapa de busca e remoção como nas árvores binárias de busca convencionais. Então se alguma propriedade vermelho-preta for violada, a árvore deve ser rebalanceada.

Caso a remoção efetiva seja de um nó vermelho, esta é realizada sem problemas, pois todas as propriedades da árvore se manterão intactas. Se o nó a ser removido for preto, a quantidade de nós pretos em pelo menos um dos caminhos da árvore foi alterado, o que implica que algumas operações de rotação e/ou alteração de cor sejam feitas para manter o balanceamento da árvore.

Para ocorrer a remoção de um nó em arvore preta e vermelha é necessário que esse nó tenha no máximo um filho que não seja folha, a maior preocupação na remoção está em não quebrar as regras básicas que fazem uma arvore ser preta e vermelha e falando principalmente sobre a regra da altura preta. O primeiro caso verifica se o pai do nó não é nulo, se for vai para o segundo caso. No segundo caso, se o nó e seu pai forem pretos e seu irmão for vermelho o pai deve ser pintado de vermelho e o irmão de preto e então se o nó for filho esquerdo, faz a rotação à esquerda de seu pai e vai pro próximo caso, se for filho direito, rotaciona o pai à direita e vai pro próximo caso. Nesse caso, se o pai do nó, o irmão, o filho esquerdo e direito do irmão forem todos pretos, pinta o irmão de vermelho e volte para o primeiro caso com o pai do nó, se não forem vai pro próximo caso. No quarto caso, se o irmão e o filho esquerdo e direito do irmão forem pretos e o pai do nó for vermelho, deve pintar o irmão de vermelho e o pai do nó de preto, se não deve prosseguir para o próximo caso. No quinto caso, se o nó for filho esquerdo e o filho direito do irmão for preto deverá pintar o irmão de vermelho e o filho esquerdo do irmão de preto e aí sim rotacionar à direita o irmão, mas se o nó for filho direito deverá pintar o irmão de vermelho e o filho direito do irmão de preto e então rotacionar para esquerda o irmão, indo para o último caso. Ao chegar no último caso deverá pintar o pai do nó de preto, caso o nó seja filho esquerdo, pinta o filho direito do irmão do nó de preto e rotaciona o pai do nó para a esquerda, se o nó for filho direito, pinta o filho esquerdo do irmão de preto e rotaciona o pai para direita. Ao sair do encadeamento de casos poderá ser feita a remoção do nó naturalmente. O algoritmo de remoção é O(log n).

Execução do Algoritmo

O programa, ao ser executado, está encarregado de imprimir na tela um menu, que por sua vez, irá mostrar as opções disponíveis ao usuário.

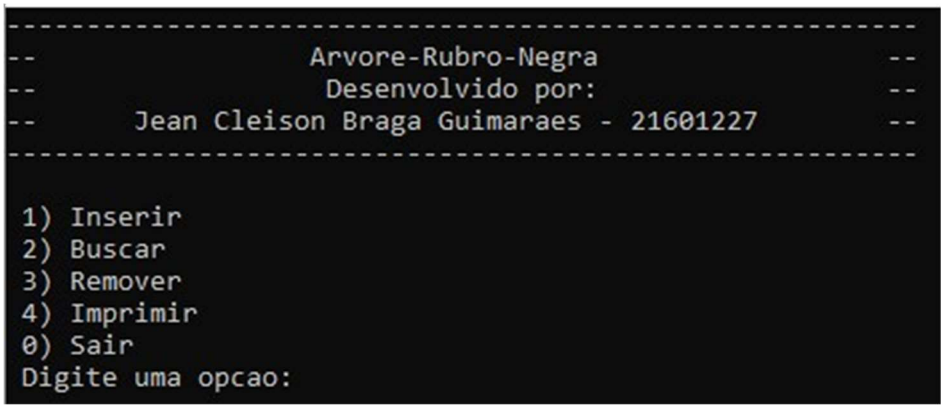


A seguir há a explicação de cada opção:

1 - Nova Arvore:

- Inicia o programa com a construção de uma árvore rubro-negra.

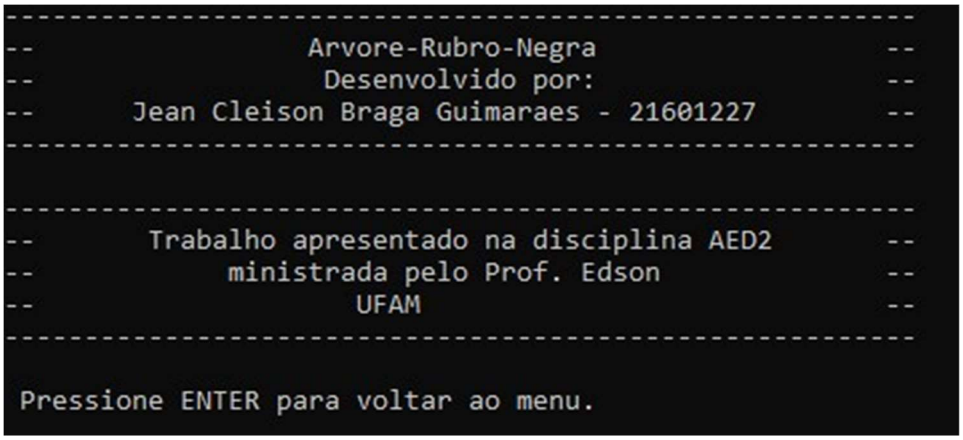
Tela da execução:



2 - Sobre:

- Mostra uma pequena informação sobre o programa.

Tela de execução:



3 - Sair:

- Finaliza o programa.

Estrutura do Algoritmo

Além dos algoritmos relativos à arvore B, como os de busca, inserção, impressão, rotação, foram criadas estruturas e algoritmos afim de aperfeiçoar o sistema. Segue abaixo as principais estruturas e funções utilizadas.

Estrutura da Árvore

```c
typedef struct no {
    int key;
    struct no * pai, * esquerda, * direita;
    int cor;
} t_no;
```

Inserção

```c
t_no * insertFixUp(t_no * raiz, t_no * z) {
    while (z != raiz && z->pai->cor == RED) {
        if (z->pai == z->pai->pai->esquerda) {
            t_no * y = z->pai->pai->direita;
            if (y->cor == RED) { //CASO 1
                z->pai->cor = BLACK;
                y->cor = BLACK;
                z->pai->pai->cor = RED;
                z = z->pai->pai;
            } else {
                if (z == z->pai->direita) { //CASO 2
                    z = z->pai;
                    raiz = left_rotate(raiz, z);
                }
                //CASO 3
                z->pai->cor = BLACK;
                z->pai->pai->cor = RED;
                raiz = right_rotate(raiz, z->pai->pai);
            }
        } else { //MESMA COISA DO IF INVERTENDO ESQUERDA E DIREITA
            if (z->pai == z->pai->pai->direita) {
                t_no * y = z->pai->pai->esquerda;
                if (y->cor == RED) {

                    z->pai->cor = BLACK;
                    y->cor = BLACK;
                    z->pai->pai->cor = RED;
                    z = z->pai->pai;
                } else {
                    if (z == z->pai->esquerda) {
                        z = z->pai;
                        raiz = right_rotate(raiz, z);
                    }
                    z->pai->cor = BLACK;
                    z->pai->pai->cor = RED;
                    raiz = left_rotate(raiz, z->pai->pai);
                }
            }
        }
    }
    raiz->cor = BLACK;
    return raiz;
}
```

```c
t_no * insert(t_no * raiz, int key) {
    t_no * z = cria_no(key);
    t_no * y = nil;
    t_no * x = raiz;

    //Caso não exista nenhum elemento
    if(raiz == nil) {
        raiz = cria_no(z->key);
        raiz->cor = BLACK;
        return raiz;
    }

    //faz a busca para inserir o nó
    while (x != nil) {
        y = x;
        if (z->key < x->key) {
            x = x->esquerda;
        } else {
            x = x->direita;
        }
    }
    z->pai = y;
    if (y == nil) {
        raiz = z;
    } else if (z->key < y->key) {
        y->esquerda = z;
    } else {
        y->direita = z;
    }

    z->esquerda = nil;
    z->direita = nil;
    z->cor = RED;
    //chama a função insertFixUp
    //para verificar e restaurar as propriedades
    //caso tenham sido quebradas
    raiz = insertFixUp(raiz, z);
    return raiz;
}
```

Remoção

```
171  //procura o sucessor de um no
172  t_no * sucessor(t_no * z) {
173      t_no * aux = z->direita;
174      while (aux->esquerda != nil) {
175          aux = aux->esquerda;
176      }
177      return aux;
178  }
179
180  t_no * deleteFixUp(t_no * raiz, t_no * x) {
181      t_no * w;
182
183      while (x != raiz && x->cor == BLACK) {
184          if (x == x->pai->esquerda) {
185              w = x->pai->direita;
186              if (w->cor == RED) { //CASO 1
187                  w->cor = BLACK;
188                  x->pai->cor = RED;
189                  raiz = left_rotate(raiz, x->pai);
190                  w = x->pai->direita;
191              }
192              //CASO 2
193              if (w->esquerda->cor == BLACK && w->direita->cor == BLACK) {
194                  w->cor = RED;
195                  x = x->pai;
196              } else {
197                  if (w->direita->cor == BLACK) { //CASO 3
198                      w->esquerda->cor = BLACK;
199                      w->cor = RED;
200                      raiz = right_rotate(raiz, w);
201                      w = x->pai->direita;
202                  }
203                  //CASO 4
204                  w->cor = x->pai->cor;
205                  x->pai->cor = BLACK;
206                  w->direita->cor = BLACK;
207                  raiz = left_rotate(raiz, x->pai);
208                  x = raiz;
209              }
210          } else {
211              w = x->pai->esquerda;
212              if (w->cor == RED) {
213                  w->cor = BLACK;
214                  x->pai->cor = RED;
215                  raiz = right_rotate(raiz, x->pai);
216                  w = x->pai->esquerda;
217              }
218              if (w->direita->cor == BLACK && w->esquerda->cor == BLACK) {
219                  w->cor = RED;
220                  x = x->pai;
221              } else {
222                  if (w->esquerda->cor == BLACK) {
223                      w->direita->cor = BLACK;
224                      w->cor = RED;
225                      raiz = left_rotate(raiz, w);
226                      w = x->pai->esquerda;
227                  }
228                  w->cor = x->pai->cor;
229                  x->pai->cor = BLACK;
230                  w->esquerda->cor = BLACK;
231                  raiz = right_rotate(raiz, x->pai);
232                  x = raiz;
233              }
234          }
235      }
236      x->cor = BLACK;
237      return raiz;
238  }
```

```
257  t_no * remover(t_no * raiz, int key) {
258      t_no * z = search(raiz, key);
259      if(z == NULL) {
260          printf("No nao encontrado\n");
261          return raiz;
262      }
263      t_no * y;
264      t_no * x;
265      if (z->esquerda == nil || z->direita == nil) {
266          y = z;
267      } else {
268          y = sucessor(z);
269      }
270
271      if (y->esquerda != nil) {
272          x = y->esquerda;
273      } else {
274          x = y->direita;
275      }
276      x->pai = y->pai;
277      if (y->pai == nil) {
278          raiz = x;
279
280      } else {
281          if (y == y->pai->esquerda) {
282              y->pai->esquerda = x;
283          } else {
284              y->pai->direita = x;
285          }
286      }
287      if (y != z) {
288          z->key = y->key;
289      }
290      if (y->cor == BLACK) {
291          raiz = deleteFixUp(raiz, x);
292      }
293      return raiz;
294  }
295
```

Busca

```
240  //Procura um no
241  t_no * search(t_no * raiz, int key) {
242      t_no * aux = raiz;
243
244      while (aux != nil) {
245          if (key < aux->key) {
246              aux = aux->esquerda;
247          } else if (key > aux->key) {
248              aux = aux->direita;
249          } else if (key == aux->key) {
250              return aux;
251          }
252      }
253
254      return NULL;
255  }
256
```

Referências Bibliográficas

1. https://pt.wikipedia.org/wiki/%C3%81rvore_rubro-negra
2. https://www.ime.usp.br/~song/mac5710/slides/08rb.pdf
3. https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/st-redblack.html
4. http://www.ufjf.br/jairo_souza/files/2012/11/5-Indexa%C3%A7%C3%A3o-Arvore-Vermelho-Preta.pdf
5. https://www.ic.unicamp.br/~rocha/teaching/2014s1/mc202/aulas/aula-arvores-rubro-negras.pdf