Notas de Aula 7 - Algoritmos e Estrutura de Dados 2 (AE43CP) – Árvores rubro-negras
Prof. Jefferson T. Oliva

Recapitulação sobre árvores binárias de busca (ABB)
- Nó raiz
- Subárvores
- ABB: possui propriedades de ordenação
- n elementos → n – 1 ramificações
- altura mínima: log n
- altura máxima: n
- Representação de uma ABB: estrutura com um campo para o item e dois ponteiros, uma subárvore direita e a outra, esquerda
- Operações: criar, liberar, buscar, inserir e remover, imprimir (pré-, in- e pós-fix)

As ABB são implementadas para agilizar o acesso à informação. Para isso, a árvore deve ser balanceada de modo que garanta que o acesso seja feito no máximo em tempo na ordem de O(log n).

Quando uma árvore está balanceada, o número de comparações em uma operação de busca é minimizada: no máximo O(log n).

Em outras palavras, a altura da árvore é mantida baixa em relação à quantidade de itens após as operações de inserção e remoção. Assim, uma árvore balanceada de altura h pode conter, no máximo, 2^{h + 1} – 1 elementos.

Diz-se que uma árvore é balanceada se a altura entre as subárvores diferem no máximo em 1.

Entretanto, sucessivas inserções de itens podem acarretar no aumento da complexidade de tempo para O(n), que é o caso em que a árvore cresce em apenas uma direção, lembrando uma lista encadeada. Também, manter árvores balanceadas é uma tarefa complexa.

Vimos em uma aula anterior, algumas propriedades de árvores, como altura, profundidade e balanceamento, sendo esta última abordado mais profundamente nessa aula.

Assim como as árvores AVL (eram vistas na disciplina de algoritmos 2 da grade antiga), as árvores rubro-negras (ARB) também são árvores binárias de busca (ABB).

ARBs também são conhecidas como vermelha-pretas ou Red-Black Trees e também tem como característica o balanceamento de seus nós.

Em árvores RB, cada nó possui os seguintes campos:
- cor (1 bit): pode ser vermelho (0) ou preto (1).
- key (e.g. inteiro): indica o valor de uma chave.
- left, right: ponteiros que apontam para as subárvores esquerda e direita.
- pai: ponteiro que aponta para o nó pai. Quando esse ponteiro é nulo, significa que o nó é raiz.

Ver exemplo no slide 6.

As ABBs que vimos na última aula têm uma semelhança com listas encadeadas simples quando percorremos essas estruturas a partir da raiz ou cabeça: quando o ponteiro auxiliar aponta para a esquerda, direita ou próximo, não é possível voltar atrás. As árvores ARB, assim como listas duplamente encadeadas, permitem “idas” e “voltas” em cada nó.

A ideia é restringir a forma como os nós podem ser coloridos em qualquer caminho da raiz até as folhas de tal forma que nenhum caminho possa ser mais que duas vezes mais longo que outro caminho, deste modo á arvore fica aproximadamente balanceada.

Uma árvore vermelha-preta é: Uma árvore binária de busca que faz a remoção e a inserção de forma inteligente, para assegurar que a árvore seja mantida balanceada.

Essa árvore é complexa, mas possui bom custo de tempo para a execução de suas operações e é eficiente na prática: pode-se buscar, inserir, e remover em tempo na ordem de O(log n).

Uma árvore vermelha-preta com n nós tem altura máxima de 2log(n + 1). O motivo será visto mais para frente.

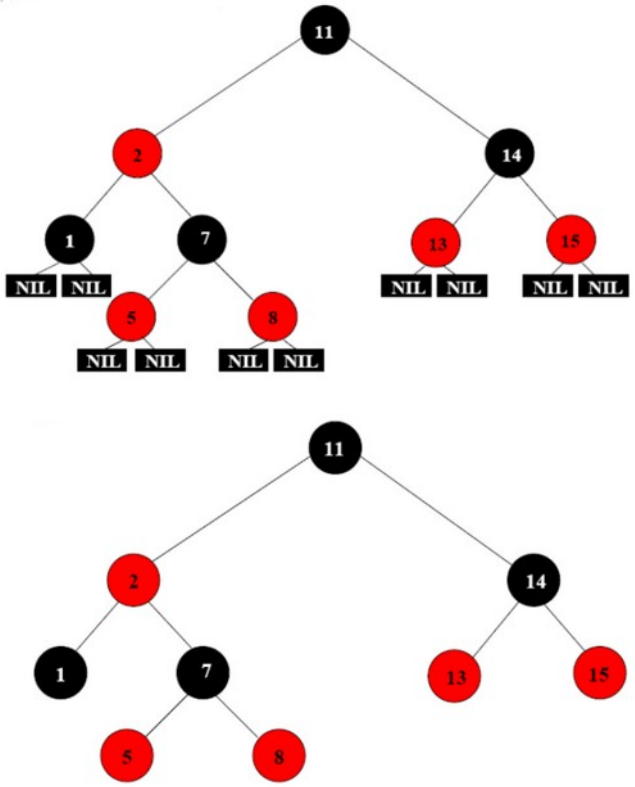
O balanceamento garante que essas árvores possuam complexidade na ordem de log n em suas operações.

Propriedades

1 - Todo nó é vermelho ou preto.
2 - A raiz é preta. A raiz pode sempre ser alterada de vermelho para preto, mas não o contrário.
3 - Toda folha (NULL) é preta. Elas não são relevantes e não contém dados, ou seja, não precisam ser mantidas na memória, bastando apenas um ponteiro NULL para identificá-las.
4 - Se um nó é vermelho, então os seus filhos são pretos. Uma condição (a partir das propriedades), para a qual, no caminho da raiz até uma sub-árvore vazia não pode existir dois nós vermelhos consecutivos. Apesar disso, podem haver dois ou mais nós pretos consecutivos. Por isso, uma subárvore pode ter o o dobro do tamanho em relação a outra.
5 - Para cada nó, todos os caminhos simples do nó até folhas descendentes contêm o mesmo número de nós pretos.

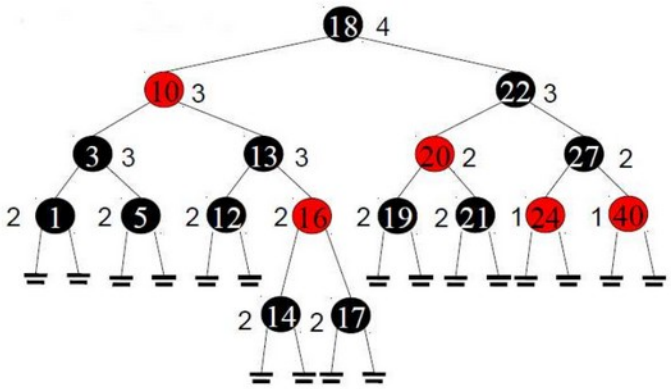
Um nó que satisfaz as propriedades anteriores é denominado equilibrado, caso contrário é dito desequilibrado: Em uma ARB, todos os nós estão equilibrados.

Algumas das formas de representação de ARB:



Cada vez que uma operação (inserção ou remoção) for realizada na árvore, o conjunto de propriedades é testado. Assim, caso alguma propriedade seja violada, são realizadas rotações e/ou ajustes de cores, de forma que a árvore permaneça "balanceada".

Restringindo o modo como os nós são coloridos desde a raiz até uma folha, assegura-se que nenhum caminho será maior que duas vezes o comprimento de qualquer outro. Em outras palavras, uma ARB é equilibrada se a sub-árvore esquerda possui a mesma altura preta em relação à direita.
Altura preta: número de nós pretos encontrados até qualquer nó folha descendente



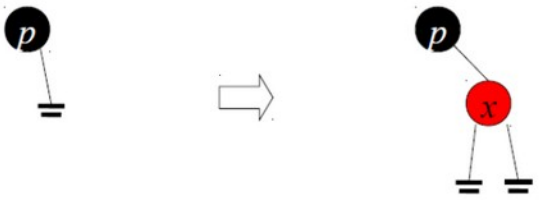
Inserção

Assim como em uma ABB qualquer, operação de inserção em uma árvore vermelha-preta começa por uma busca da posição onde o novo nó deve ser inserido. Em outras palavras, a inserção em uma ARB ocorre da mesma forma em relação à ABB, mas podendo ter rebalanceamento, assim como na AVL.

As operações inserção e de remoção são mais complicadas nas árvores vermelha-pretas em relação à ABB, pois podem violar alguma propriedade de ARB.

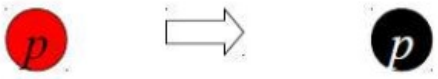
Essas operações podem ser implementadas de forma bastante parecida com as respectivas operações nas árvores binárias de busca, mas com adição de algumas funcionalidades, como mudanças de cor e rotações.

Nesse contexto, em uma ARB, um nó é inserido sempre na cor vermelha: caso a inserção seja feita em uma árvore vazia, basta alterar a cor do nó para preto. Se o nó fosse inserido na cor preta, invalidaria a propriedade 5 (para cada nó, todos os caminhos do nó para as folhas descendentes contém o mesmo número de nós PRETOS).

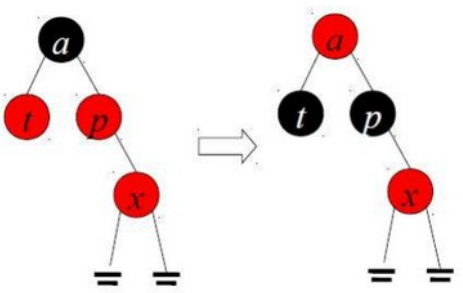


Caso o nó inserido possui um pai preto, não é necessário fazer alguma operação adicional, pois a árvore encontra-se balanceada.

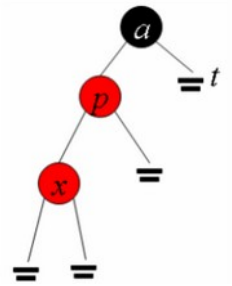
Desse modo, em cada inserção, um conjunto de propriedades é testado. Caso a ARB não satisfazer essas propriedades, deverão ser realizadas rotações e/ou ajustes de cores para o balanceamento da árvore.
Caso 1: se esta inserção é feita em uma árvore vazia, basta alterar a cor do nó para preto, satisfazendo a propriedade de que a raiz deve ser preta.



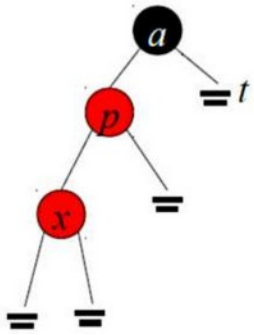
Caso 2: Ao inserir x e respectivo tio é vermelho, é necessário fazer a re-coloração de a, t e p.
- Se o pai de a é vermelho, o rebalanceamento deve ser feito novamente
- Se a é raiz, então ela deve ser mantida preta

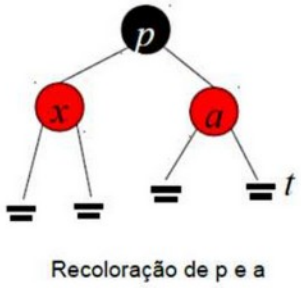


Caso 3: Suponha que o tio do nó inserido é preto
- Para manter a propriedade 4 (se um nó é vermelho, então seus filhos são pretos) é preciso fazer rotações envolvendo a, t, p e x. Apenas mudar a cor de x para preto violaria a propriedade de os caminhos pretos possuem o mesmo tamanho.
- Há 4 sub-casos que correspondem às 4 rotações possíveis



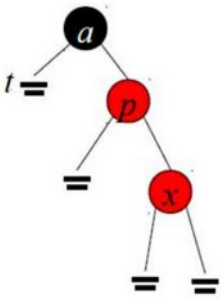
Caso 3a: Rotação à Direita – x tem irmão preto, é filho esquerdo de p, e neto esquerdo de a

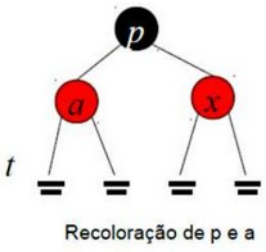




a → l aponta para p → r, p → r aponta para a

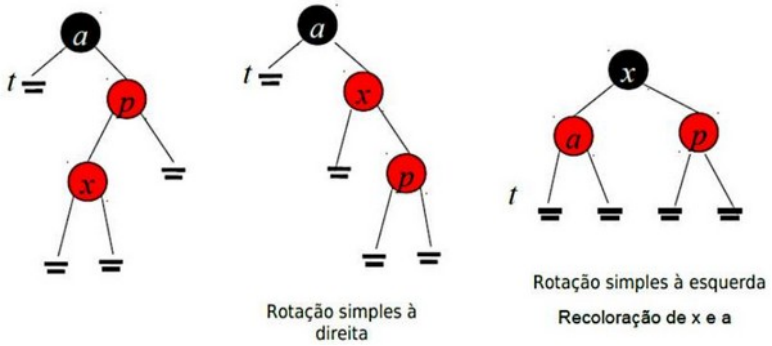
Caso 3b: Rotação à Esquerda – x tem irmão preto, é filho direito de p, e neto direito de a. Como nessa árvore não tem fator de balanceamento (como em árvores AVL), para definir as rotações temos que verificar a subárvore que define grau de parentesco entre os nós.





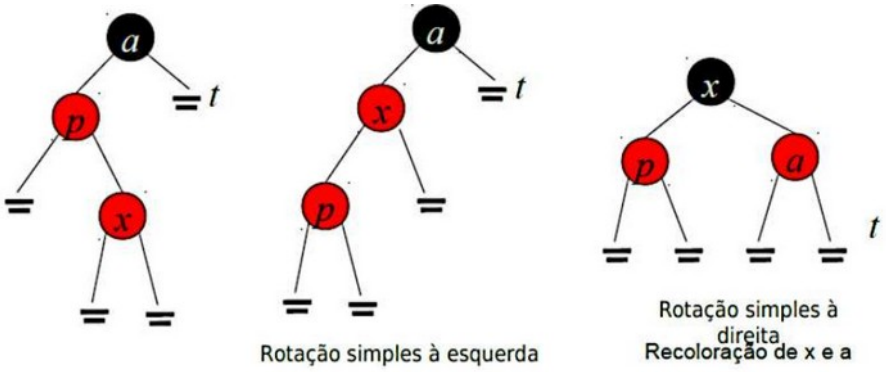
a → r aponta para p → l, p → l aponta para a

Caso 3c: rotação dupla esquerda – x tem irmão preto, é filho esquerdo de p, e neto direito de a
- Pode ser visto como um caso 3a seguido do caso 3b



1 - p → l aponta para x → r, x → r aponta para p
2 - a → r aponta para x → l, x → l aponta para a

Caso 3d: rotação dupla direita – x tem irmão preto, é filho direito de p, e neto esquerdo de a (forma um triângulo).
- Pode ser visto como um caso 3b seguido do caso 3a



1 - p → r aponta para x → l, x → l aponta para p
2 - a → l aponta para x → r, x → r aponta para a

Exemplos: slides de 27 a 35

Remoção

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

Caso a remoção efetiva seja de um nó vermelho, todas as propriedades da árvore ainda 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 em que algumas operações de rotação e/ou alteração de cor sejam feitas para manter o balanceamento da árvore.

Complexidade

A árvore vermelha-preta necessita de, no máximo, 1 rotação para o rebalanceamento após a inserção, pode ter log(n) trocas de cores.

Na remoção, a árvore vermelha-preta faz no máximo 1 rotação (simples ou dupla), enquanto a AVL pode fazer log(n) rotações.

A árvore VP tem melhor pior caso que a árvore AVL
- Isso faz com que a árvore seja utilizada em aplicações de tempo real (críticas)
- AVL é uma estrutura mais balanceada que a VP
- Pode ser usada para construir blocos em outras estruturas de dados que precisam de garantias no pior caso. Por exemplo, estruturas de dados em geometria computacional podem ser baseadas em árvores vermelha-preta.

Implementação da Operação de Inserção

Arquivo .h

```
typedef struct NodeVP NodeVP;

NodeVP * criar_AVP(int key);

int liberar_AVP(NodeVP *tree);

NodeVP* pesquisar(NodeVP *tree, int key);

NodeVP * inserir(NodeVP *tree, int key);
```

Arquivo .c

```
#include <stdio.h>
#include <stdlib.h>
#include "RB_tree.h"

struct NodeVP{
   int key;
   int color;
   NodeVP *father;
   NodeVP *left;
   NodeVP *right;
};


NodeVP* criar_AVP(int key){
   NodeVP* tree = (NodeVP*) malloc(sizeof(NodeVP));
   tree->key = key;
   tree->color = 0;
   tree->left = NULL;
   tree->right = NULL;
   tree->father = NULL;

   return tree;
}


int liberar(NodeVP *tree){
   if (tree != NULL){
      free(tree);

      return 1;
   }
   return 0;
}


NodeVP* pesquisar(NodeVP *tree, int key){
   if (tree != NULL){
      if (key == tree->key)
         return tree;
      else if (key < tree->key)
         return pesquisar(tree->left, key);
      else
         return pesquisar(tree->right, key);
   }

   return NULL;
}


NodeVP* obter_avo(NodeVP *no){
   if ((no != NULL) && (no->father != NULL)){
      return no->father->father;
   }

   return NULL;
}
```

```
NodeVP* obter_tio(NodeVP *no){
   NodeVP* avo = obter_avo(no);
   NodeVP* aux = NULL;


   if (avo != NULL){
      if (avo->left == no->father){
         aux = avo->right;
      }else
         aux = avo->left;
   }

   return aux;
}


NodeVP * rotacionar_dir(NodeVP *x){
   NodeVP *p = x->father; // pai de x
   NodeVP *a = p->father; // avô de x

   p->father = a->father;

   if (a->father != NULL)
      a->father->left = p;

   a->father = p;
   a->left = p->right;
   p->right = a;

   x = p;

   return x;
}


NodeVP * rotacionar_esq(NodeVP *x){
   NodeVP *p = x->father; // pai de x
   NodeVP * a = p->father; // avô de x

   p->father = a->father;

   if (a->father != NULL)
      a->father->right = p;

   a->father = p;
   a->right = p->left;
   p->left = a;

   x = p;

   return x;
}
```

```
void rotacionar_dir_esq(NodeVP *x){
   NodeVP *p = x->father; // pai de x
   NodeVP *a = p->father; // avô de x
   // Rotação à direita
   x->father = p->father;
   a->right = x;
   p->father = x;
   p->left = x->right;
   x->right = p;
   // Rotação à esquerda
    rotacionar_esq(p);
}


void rotacionar_esq_dir(NodeVP *x){
   NodeVP *p = x->father; // pai de x
   NodeVP * a = p->father; // avô de x

   // Rotação à esquerda
   x->father = p->father;
   a->left = x;
   p->father = x;
   p->right = x->left;
   x->left = p;

   // Rotação à direita
    rotacionar_dir(p);
}
```

```
NodeVP * balancear(NodeVP *x){
   NodeVP *pai, *tio;

   if (x->father == NULL){ // caso 1
      x->color = 1;
      //printf("caso 1\n");
   }else{
      pai = x->father;

      if (pai->color == 0){
         tio = obter_tio(x);

         if ((tio != NULL) && (tio->color == 0)){// caso 2
            pai->color = tio->color = 1;
            x = obter_avo(x);

            //printf("caso 2\n");

                  if (x->father != NULL)
                 x->color = 0;
         }else{ // caso 3

            if (pai->left == x){
               if (pai->father->right == pai)
                  x = rotacionar_dir_esq(x);       // caso 3c
               else
                  x = rotacionar_dir(x);       // caso 3a
            }else{ // caso 3
               if (pai->father->left == pai)
               x = rotacionar_esq_dir(x);      // caso 3d
               else
                  x = rotacionar_esq(x);       // caso 3b
            }

            x->color = 1;
            x->left->color = 0;
            x->right->color = 0;
         }
      }
   }

   return x;
}
```

```
NodeVP * inserir(NodeVP *tree, int key){
   NodeVP *auxP, *auxF;

   if (tree == NULL){
      tree = criar_AVP(key);
      tree->color = 1;
   }else{
      auxP = auxF = tree;

      while (auxF != NULL){
         auxP = auxF;

         if (key < auxF->key)
            auxF = auxF->left;
         else
            auxF = auxF->right;
      }

      auxF= criar_AVP(key);
      auxF->father = auxP;

      if (auxF->key < auxP->key)
         auxP->left = auxF;
      else
         auxP->right = auxF;

      while ((auxF->father != NULL)  && (auxF->color == 0) && (auxF->father->color == 0))
         balancear(auxF);

      tree = auxF;
   }

   return tree;
}
```

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Marin, L. O. Árvores Vermelho-Preta (Rubro-Negra) (RB-Tree). AE23CP - Algoritmos e Estrutura de Dados II. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.

Song, S. W. Árvore Rubro-Negra. MAC 508 - Estrutura de Dados. Slides. Ciência da Computação. IME/USP/São Paulo, 2008.