# Algoritmos e Estruturas de Dados I - 2º semestre de 2013 – Turma 02

# Segundo Exercício Programa - Árvores Rubro-Negras – 22/12/2013


# Profa. Ariane Machado Lima


## 1 Introdução

Seja *T* uma árvore binária de busca. Associe a cada um de seus nós uma cor: *rubra* ou *negra*.

Dizemos que *T* é uma *árvore rubro-negra* se existe uma coloração dos nós de T tal que todo vértice *v* em *T* satisfaz:

a) Se *v* é um nó externo (folha), a cor de *v* é negro;

b) Todos os caminhos de *v* para seus descendentes nós externos têm o mesmo número de nós com cor negra;

c) Se *v* é rubro e *v* não é raiz da árvore, então o pai de *v* é negro.

Quando um nó satisfaz as condições a) a c) dizemos que ele está *equilibrado*, caso contrário dizemos que ele está *desequilibrado*.

Não são colocados valores nos nós folhas. Ou seja, toda a informação "útil" de um árvore rubro-negra estão nos nós internos.

Exemplo:

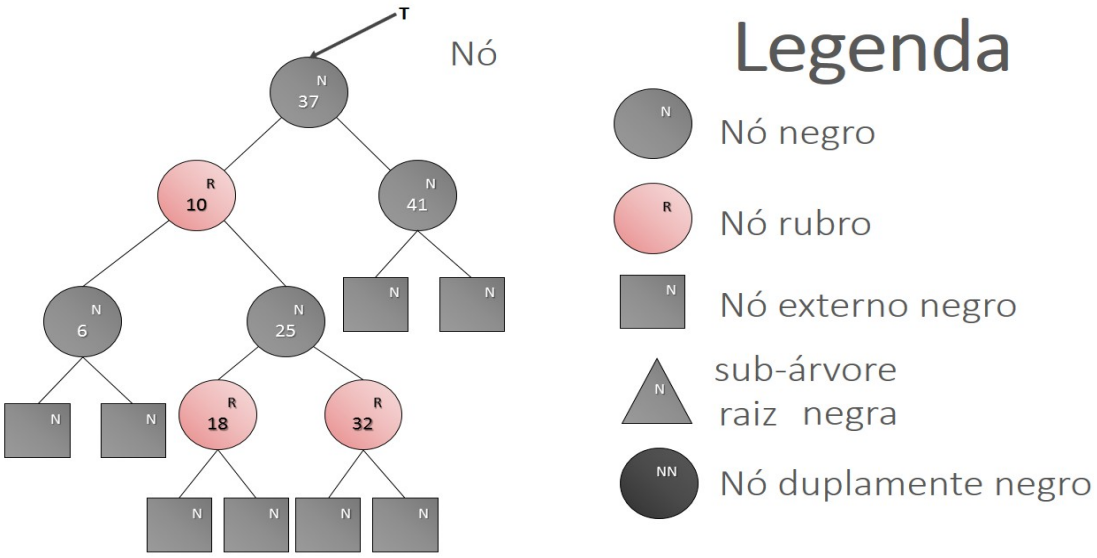


Fig 1: Exemplo de árvore rubro-negra. Nós duplo-negros serão explicados adiante.

OBS: Prova-se que:

Se *T* é uma árvore rubro-negra com *n* nós internos então H(t) ≤ 2 log2 (n+1).

Em C, árvores rubro-negras podem ser implementadas dinamicamente com a seguinte estrutura:

```
typedef int TIPOCHAVE;
typedef enum{esquerdo, direito} LADO;
typedef enum{rubro, negro} COR;

typedef struct aux{
  TIPOCHAVE chave;
  struct aux *esq;
  struct aux *dir;
  struct aux *pai;  //para facilitar e tornar mais rápida a remoção
  COR cor;
} NO;

typedef NO* PNO;
```

Para evitar consumo desnecessário de memória, os nós folhas (que são todos iguais) devem ser representados por um único nó especial, chamado *externo*, de cor negra. Ou seja, sempre que um filho de um nó seria NULL em uma árvore tradicional, aqui apontará para *externo*. Assim, uma árvore vazia possui raiz = *externo*.

Além disso, vamos considerar que a raiz deve ser sempre negra, pois isto simplificará os algoritmos de inserção e remoção. Isto é feito sem perda de generalidade, uma vez que qualquer árvore rubro-negra que tenha raiz rubra pode ter essa raiz transformada em negra mantendo todos os nós equilibrados.

## 2 Inserção em Árvores Rubro-Negras

Seja *T* uma árvore rubro-negra e *q* um nó a ser inserido em *T*.

O algoritmo de inserção numa árvore binária de busca substitui um dos nós externos de T pelo novo nó interno q (com dois filhos nós externos negros), resultando numa nova árvore T1. Inicialmente insira *q* com a cor rubra.

Para saber se T1 é rubro-negra basta verificar as condições (a), (b) e (c).

(a) É verdadeira,

(b) Também é verdadeira, pois definimos cor(q)=rubra.

É preciso verificar a condição (c).

Sejam v=pai de q e w=avô de q em T1. As seguintes possibilidades podem ocorrer:

**Caso 1:** v é negro

A condição (c) é verdadeira e T1 é rubro-negra.

**Caso 2:** v é rubro

O nó inserido q não satisfaz (c), e portanto é preciso equilibrá-lo.

OBS: Se v não é a raiz da árvore, então w é negro (pois w é pai de v).

Vamos examinar a cor do nó t, irmão de v. Podemos ter os seguintes casos:

**Caso 2.1:** t é rubro (Figura 2)

Neste caso, v e t devem se tornar negros e w deve se tornar rubro.

OBS: O nó w = avô de q pode ter ficado desequilibrado se seu pai (r) tiver a cor rubra. Ou seja, deve-se repetir o processo considerando q ←w e redefinindo v, w e t como pai, avô e tio de q, respectivamente.

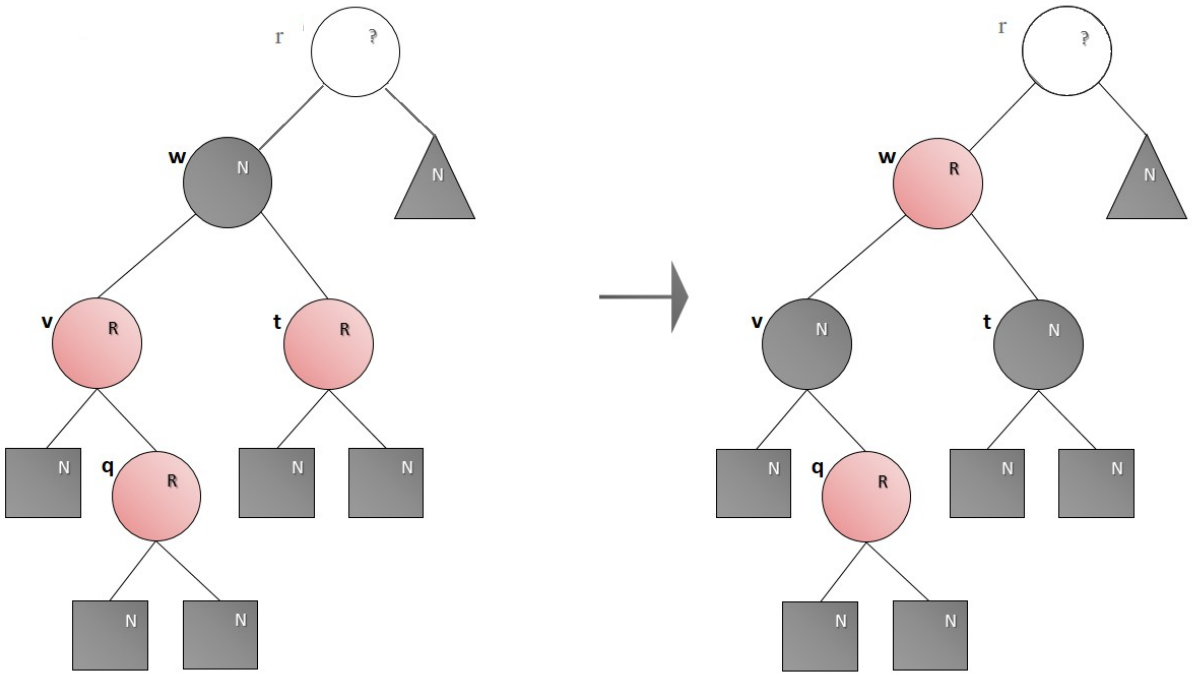


Fig 2: Inserção quando quando v = pai(q) é rubro e t = irmão(v) é rubro.

**Caso 2.2:** t é negro

Neste caso, utilizam-se as *operações de rotações*.

OBS: Basta uma operação de rotação, seguida de uma troca de cores conveniente de dois nós para equilibrar q e transformar T1 em uma árvore rubro-negra.

Os seguintes casos podem ocorrer:

**Caso 2.2.1:** q é filho esquerdo de v e v é filho **esquerdo** de w (Figura 3).

Neste caso é preciso fazer uma rotação à direita em w, como mostra a Figura 3. Além disso, v deve se tornar negro e w deve se tornar rubro.

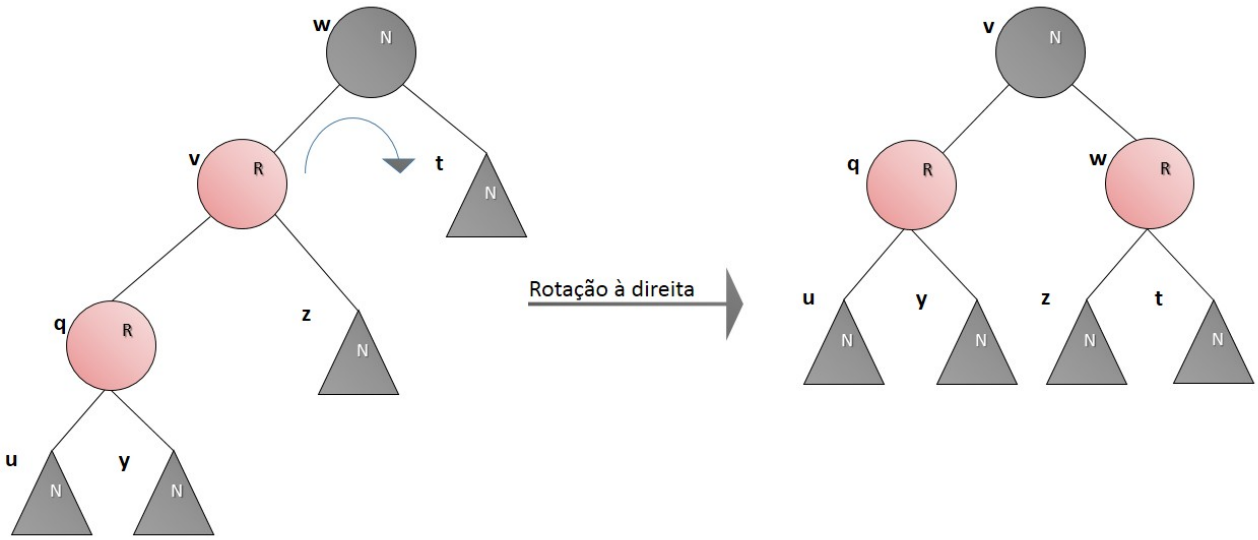


Fig 3: Inserção quando quando v = pai(q) é rubro, t = irmão(v) é negro, q = filho_esq (v) e **v = filho_esq(w)**. Os triângulos negros com um rótulo x representam subárvores rubro-negras tendo o nó negro x como raiz .

**Caso 2.2.2:** q é filho esquerdo de v e v é filho **direito** de w (Figura 4).

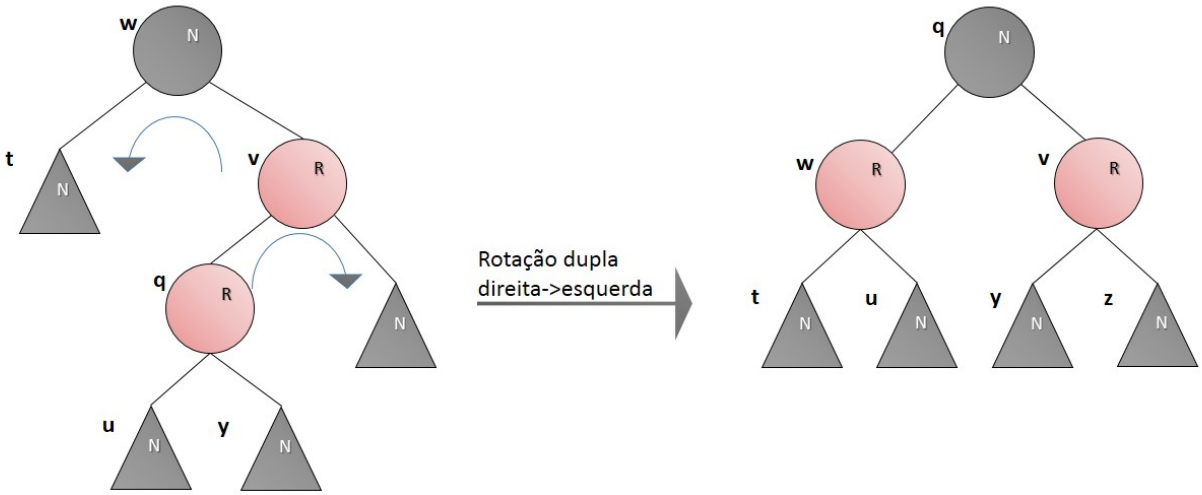


Fig 4: Inserção quando quando v = pai(q) é rubro, t = irmão(v) é negro, q = filho_esq (v) e **v = filho_dir(w)**. Os triângulos negros com um rótulo x representam subárvores rubro-negras tendo o nó negro x como raiz .

Neste caso é preciso fazer uma rotação dupla: uma rotação à direita em v e outra rotação à esquerda em w, como mostra a Figura 4. Além disso, q deve se tornar negro e w deve se tornar rubro.

**Caso 2.2.3:** q é filho direito de v e v é filho direito de w (Figura 5).

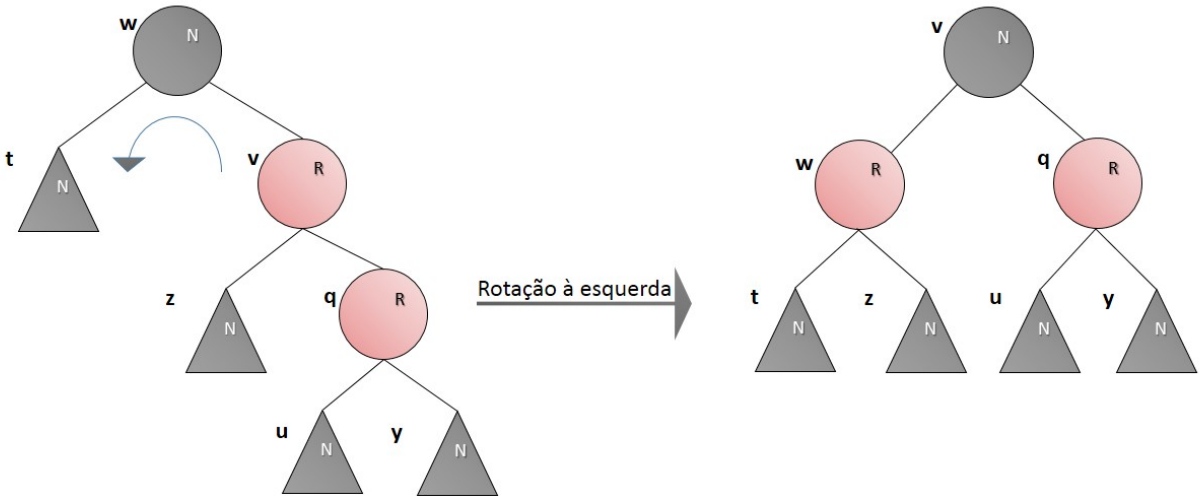


Fig 5: Inserção quando quando v = pai(q) é rubro, t = irmão(v) é negro, q = filho_dir (v) e v = filho_dir(w). Os triângulos negros com um rótulo x representam subárvores rubro-negras tendo o nó negro x como raiz.

Neste caso é preciso fazer uma rotação à esquerda em w, como mostra a Figura 5. Além disso, v deve se tornar negro e w deve se tornar rubro.

**Caso 2.2.4:** q é filho direito de v e v é filho esquerdo de w (Figura 6).

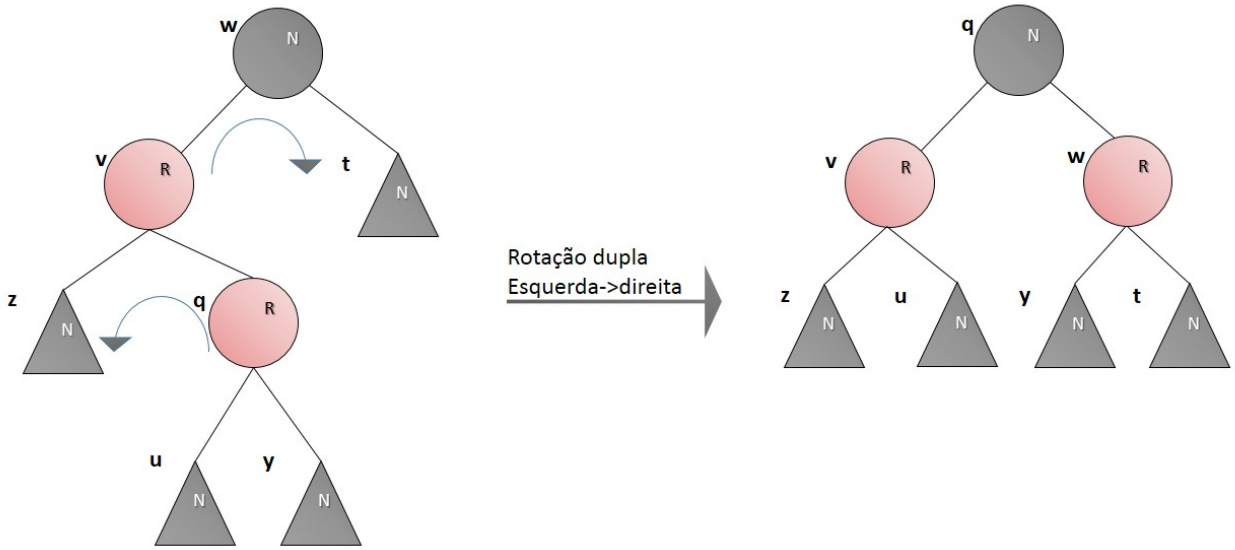


Fig 6: Inserção quando quando v = pai(q) é rubro, t = irmão(v) é negro, q = filho_dir (v) e v = filho_esq(w). Os triângulos negros com um rótulo x representam subárvores rubro-negras tendo o nó negro x como raiz.

Neste caso é preciso fazer uma rotação dupla: uma rotação à esquerda em v e outra rotação à direita em w, como mostra a Figura 6. Além disso, q deve se tornar negro e w deve se tornar rubro.

## 3 Implementação da Inserção em Árvores Rubro-Negras

A inserção é realizada pela função recursiva `inserir_RN(raiz, x, atual, pai, avo, controle)`, na qual `raiz` é o ponteiro para a raiz da árvore, `x` é o valor da chave a ser inserido na árvore, os ponteiros `atual`, `pai` e `avo` apontam, respectivamente, para o nó corrente da busca, seu pai e seu avô, e `controle` controla a chamada da função `rotacionar` (descrita a seguir), que só deve ser chamada quando `controle` = 1. Caso contrário, se `controle` = 0, então `rotacionar` não deve ser chamada, mas `controle` deve receber valor 1. Essa função `inserir_RN` primeiro faz uma busca para localizar a posição na árvore onde o novo nó com chave x será inserido e então realiza a inserção propriamente dita de forma que a árvore resultante continue sendo rubro-negra. Essa função deve ser implementada recursivamente, sendo a busca pela posição de inserção implementada dentro desta função (sem chamar a função buscar_no, utilizada na remoção), a fim de identificar corretamente o nó atual, pai e avo COMO se o campo pai não existisse. Note, porém, que a inserção deve atualizar o campo pai dos nós modificados. A primeira chamada desta função (chamada externa da recursão) é `inserir_RN(raiz, x, raiz, NULL, NULL, 1)`. A função de inserção deve retornar `true` se inserir com sucesso e `false` caso contrário (se já existir um nó com a chave x).

Para verificar e acertar o equilíbrio de um nó, a função de inserção deve chamar a função `rotacionar(raiz, filho, atual, pai, avo, controle)`, na qual `raiz` é o ponteiro para a raiz da árvore, `atual`, `pai`, `avo` e `controle` são como antes e `filho` aponta para o filho de `atual` no processo da busca (na primeira chamada de `rotacionar`, `filho` aponta para o nó inserido). Dentro desta função, quando o caso 2.1 for identificado, `controle` = 0, indicando que `rotacionar` somente será executada novamente quando o nó atual for o avô do atual. Caso contrário `controle` = 2, o que significa que `rotacionar` não será mais executada até o final do processo de inserção de `x`. Na primeira chamada de `rotacionar`, `filho` aponta para o novo nó incluído. A função de rotação não deve retornar nada.

Observação: os ponteiros `filho`, `atual`, `pai` e `avo` desta seção correspondem aos ponteiros q, v, w e r da seção 2.

Não esqueçam de atualizar o campo pai de cada nó!

## 4 Complexidade do algoritmo InsereRubroNegra

O caso 2.1 pode ocorrer tantas vezes quanto a metade do comprimento do caminho do nó inserido até a raiz, isto é, $O(\log_2 n)$. Cada um dos demais casos pode ocorrer no máximo uma vez e o custo de suas rotações não alteram a complexidade. Assim a complexidade no pior caso é $O(\log_2 n)$.

## 5 Remoção em Árvores Rubro-Negras

Seja *T* uma árvore rubro-negra. Queremos remover da árvore T um nó contendo a chave de x, mantendo-a rubro-negra. Para isso, devem ser utilizados os algoritmos de busca e de remoção de nós de

árvores binárias de busca. O algoritmo de remoção, porém, deve ser implementado com algumas adaptações:

- Lembre-se de que todo nó "útil" da árvore possui 2 filhos, sendo 0, 1 ou 2 deles o nó externo. Apenas nós diferentes do externo são removidos. Assim, os 3 casos de remoção em árvores binárias de busca devem ser adaptados para a árvore rubro-negra que possui o nó externo no lugar de NULL. Assim, deve-se considerar se um nó possui 0, 1 ou 2 filhos DIFERENTES DE EXTERNO.
- Seja y um apontador para o nó que vai ser realmente removido (lembre-se que, caso o nó com chave `chave` possua dois filhos, um de seus descendentes é que será removido. Neste caso, ESCOLHA PARA REMOVER O MENOR DESCENDENTE DIREITO DO NÓ com chave `chave` - isso é importante para a correção dos EPs!!!);
- Faça q apontar para o único filho (não externo) de y ou para o nó externo (caso os dois filhos de y sejam externos);
- Remova y;
- Se a cor de y era rubro, não há mais nada a fazer (a árvore continua rubro-negra). Se a cor de y era negro, então a propriedade b) das árvores rubro-negras foi violada, pois todo caminho de um nó até um nó externo descendente seu, que passava pelo nó y, agora tem um negro a menos. Neste caso, é nessário executar um algoritmo para equilibrar a árvore.

Para equilibrar a árvore resolvendo o problema da contagem de negros no caminho de um nó até um nó externo descendente, bastaria fazer com que q (o filho de y ou nó externo que ficou no lugar do y) tivesse "um negro a mais". Ou seja, nós "empurramos" a negritude de y para q. Se q for a raiz ou se q for rubro, basta então tornar q negro. Mas se q não for raiz e já for negro, esse "negro a mais" o tornou "duplamente negro". No entanto isso viola a definição de árvores rubro-negras, na qual cada nó é rubro ou negro (ou seja, não existe "duplamente negro"). Para resolver esse problema, é preciso "subir" esse negro extra (em um laço) até que:

a) q aponte para um nó vermelho (e então ele "absorve" esse negro extra tornando-se preto)

ou

b) x aponte para a raiz, e então esse negro extra pode ser simplesmente "removido" (ou seja, nada preciso ser feito, apenas se assegurar que a raiz seja negra como já convencionamos)

ou

c) que seja possível fazer rotações e mudanças de cor que resolvam o problema. Para isso, há quatro casos a considerar. Em todos eles consideramos v = pai de q, e t = irmão de q. Sabemos que o nó t não é externo (caso contrário, o número de negros no caminho de v de q até t seria menor que o do caminho de v até q). Nas figuras a seguir, os triângulos negros com rótulo u (ou z) representam subárvores rubro-negras (equilibradas) tendo o nó u (ou z) como raiz, e a cor descrita neste triângulo representa a cor de u (ou z).

**Caso 1**: t = irmão(q) é rubro. Portanto, v = pai(t) = pai(q) é negro (Figura 7).

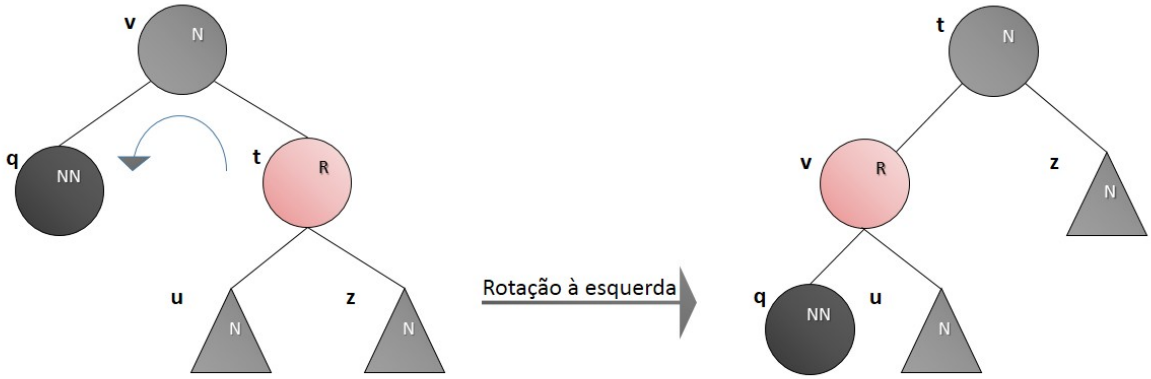


Fig 7: Equilíbrio quando t = irmão de q é rubro.

Neste caso, t deve se tornar negro, v deve se tornar rubro. Além disso, é preciso fazer uma rotação à esquerda em v, como mostra a figura 7. Depois da rotação, devemos atualizar t (irmão de q), que agora deve apontar para o nó apontado por u (pois deve ser iniciada uma nova iteração do processo para equilibrar a árvore... afinal, q ainda é duplamente negro). Ou seja, como u é negro, convertemos o caso 1 para um dos casos 2, 3 ou 4.

**Caso 2:** t = irmão(q) é negro e os dois filhos de t são negros (Figura 8).

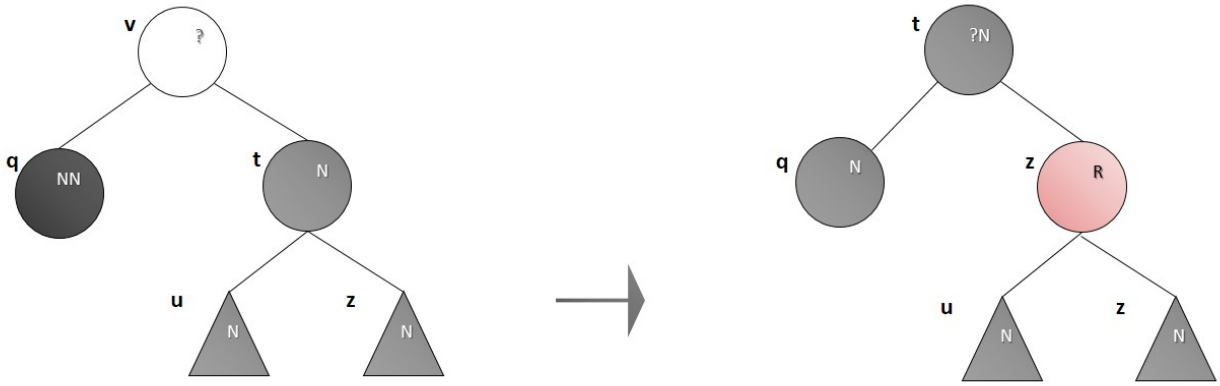


Fig 8: Equilíbrio quando t = irmão de q é negro e os dois filhos de t também são.

Neste caso, "podemos retirar um negro de q e um negro de t e adicionar um negro ao pai de q e t" (ou seja, t deve ser tornar rubro, e q agora é considerado apenas negro). Como não sabemos a cor do pai de q, esse negro a mais vai torná-lo negro ou "duplamente negro". Por isso o laço do algoritmo deve ser repetido considerando agora o pai de q como sendo o novo q.

**Caso 3**: t = irmão(q) é negro, u = filho esquerdo de t é rubro e z = filho direito de t é negro (Figura 9).

Neste caso, troca-se as cores de t e u, faz uma rotação à direita em t e chama de t o novo irmão de q (t $\leftarrow$ u). Assim, o caso 3 foi transformado no caso 4.

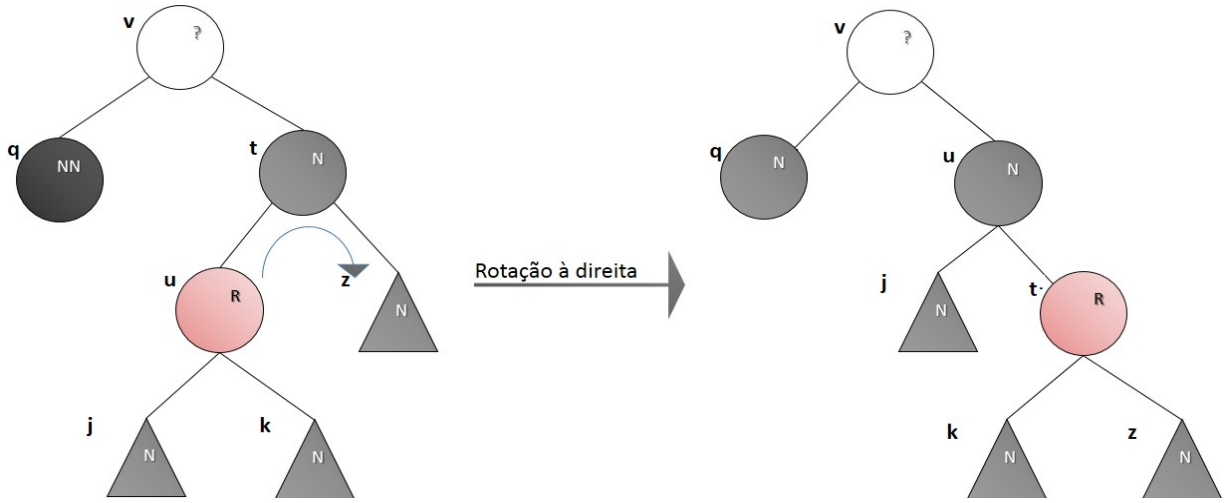


Fig 9: Equilíbrio quando t = irmão(q) é negro, u = filho esquerdo de t é rubro e z = filho direito de t é negro.

**Caso 4**: t = irmão(q) é negro e z = filho direito de t é rubro (Figura 10).

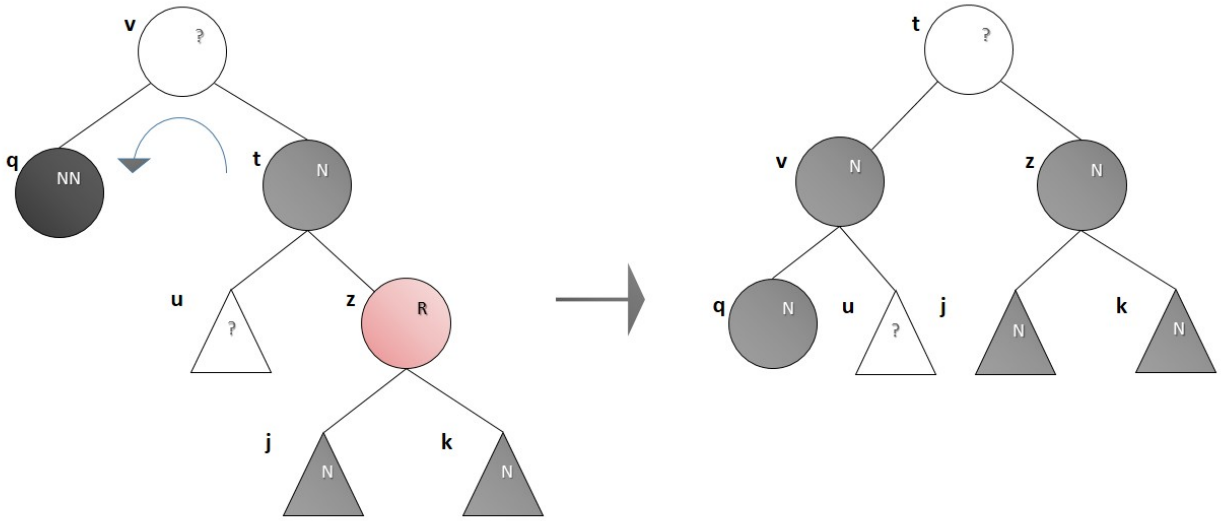


Fig 10: Equilíbrio quando t = irmão(q) é negro, z = filho esquerdo de t é rubro.

Neste caso, t deve receber a cor de v, v e z devem se tornar negros e deve ser feita uma rotação à esquerda em v. Com isso, “o negro extra” de q pode ser “removido”. Para sair do loop de equilibrio, faça q apontar para a raiz (critério de parada b).

A segunda parte do algoritmo de equilibrar, ainda dentro do laço, é verificar os casos 1 a 4 para quando q é o filho direito. Nestes casos é preciso inverter tudo o que é esquerdo e direito.

Por fim (já fora do loop), q deve se tornar negro.

## 6 Implementação da Remoção em Árvores Rubro-Negras

A função de remoção `remover_RN(PNO* raiz, TIPOCHAVE x)` remove a chave `x` da árvore apontada por *`raiz`, devolvendo true se removeu com sucesso ou false caso contrário (por exemplo, se a chave não estava na árvore). Os passos da remoção são descritos no início da seção 5. Para achar o nó com chave `x`, deve utilizar a função `buscar_no(PNO raiz, TIPOCHAVE x)`, que faz uma busca pelo nó com chave = `x`, retornando o ponteiro para esse nó. No caso do nó com chave = `x` possuir dois filhos não-externos, o nó substituto deve encontrado utilizando a função `menor_descendente_direito(PNO no)`, que retorna o ponteiro para o nó que é o menor descendente direito de `no`. Seja y o nó que foi REALMENTE removido. Se y era negro, a função de remoção deve chamar a função `equilibrar_RN_apos_remocao(PNO* raiz, PNO q)`, e retornar true. Dica: use uma variável booleana para indicar se y era negro ou não.

A função `equilibrar_RN_apos_remocao(PNO* raiz, PNO q)` equilibra a árvore apontada por *`raiz` após uma remoção, considerando que o nó `q` é o que pode estar desequilibrando a árvore (sem retornar nada). Essa função deve ser implementada iterativamente (com um laço, como descrito na seção 5). Conforme o caso, essa função deve chamar as funções `rotacionar_a_direita(PNO* raiz, PNO no)` ou `rotacionar_a_direita(PNO* raiz, PNO no)`, que fazem uma rotação à direita ou à esquerda, respectivamente, no nó `no`.

*** Não esqueçam de atualizar o campo pai de cada nó! ***

Observação: a cor “duplamente negro” é fictícia, apenas para melhor entender o algoritmo. Ou seja, em nenhum momento da codificação um nó recebe cor “duplamente negro”, e nem há nehuma variável com o “negro extra”.

## 7 Complexidade do Algoritmo RemoveRubroNegra

A remoção propriamente possui complexidade $O(\log_2 n)$, pois precisa buscar o nó a ser removido. O método para equilibrar a árvore, nos casos 1, 3 e 4 faz um número constante de trocas de cores e no máximo 3 rotações simples. O caso 2 é o único que promove novas iterações do laço, cada vez subindo o ponteiro q para um nível acima até no máximo a raiz. Logo, este caso pode ser repetido no máximo $O(\log_2 n)$. Assim, a complexidade total da remoção também é de $O(\log_2 n)$, como na inserção.

## 8 Arquivos fornecidos

Para este exercício são fornecidos três arquivos:

- arvore_rubro_negra.h: contendo o cabeçalho das funções e a estrutura de dados que será utilizada para o gerenciamento de árvores rubro-negras. NÃO ALTERE AS DEFINIÇÕES FORNECIDAS! ;

- arvore_rubro_negra.c: arquivo contendo a implementação das funções especificadas em arvore_rubro_negra.h e, potencialmente, funções auxiliares. Fiquem à vontade se quiserem escrever funções AUXILIARES ADICIONAIS.

- testa_rubro_negra.c: arquivo principal (a partir do qual será gerado o programa executável) que utiliza as funções implementadas em arvore_rubro_negra.c para a inserção, remoção e impressão em/de árvores rubro-negras. Segundo a rotina principal, o programa executável testaRB.exe lê da entrada padrão uma sequência de operações, uma em cada linha. Cada operação pode ser "i" (inserção), "r" (remoção), "p" (impressão) e "f" (fim). Nas linhas contendo as operações "i" e "r" deve ser fornecida também a chave a ser inserida/removida da árvore. "p" imprime a árvore atual e "f" finaliza a execução.

Além desses arquivos serão fornecidos o arquivo Makefile (para compilação do programa utilizando a ferramenta make) e arquivos de entrada e de saída para que os alunos possam testar suas implementações.

Exemplo de arquivo de entrada:

```
p
i 10
p
i 20
p
r 15
r 10
p
f
```

## 9 Entrega

Cada aluno deverá entregar sua implementação do arquivo **arvore_rubro_negra.c** (apenas este arquivo). Este arquivo deve ser submetido via TIDIA dentro de um arquivo NUM_USP.zip ou NUM_USP.rar (onde NUM_USP deve ser o valor do número USP do aluno [por exemplo, 1234567.zip]) até às 23:00h do dia 22/12/2013 (com 59 minutos de tolerância).

## 10 Observações

Para facilitar o entendimento durante a implementação, a cor foi definida aqui como um enum de rubro e negro. No entanto, para economizar espaço, bastaria um bit para armazenar a cor.

Como árvores rubro-negras possuem altura $H(t) \leq 2 \log_2 (n+1)$, os algoritmos de busca, inserção, remoção, máximo, mínimo, sucessor, predecessor possuem complexidade $O(\log_2 n)$, e portanto são mais eficientes do que aqueles implementados em árvores binárias de busca convencionais.

Árvores AVL possuem altura máxima e média menor do que árvores rubro-negras ( $H(t_{AVL}) \sim 1{,}44 * \log_2 n$). Por isso, buscas em árvores AVL são mais rápidas (apesar da complexidade ser igual). No entanto, inserções e remoções em árvores rubro-negras são mais rápidas do que em árvores AVL, ou seja, são mais indicadas para árvores que sofrem muitas alterações. Além disso, árvores rubro-negras ocupam um espaço ligeiramente menor (cada nó possui a mais um bit para a cor, enquanto árvores AVL precisam de dois bits a mais para o fator de balanceamento.

Neste EP implementamos árvores rubro-negras nas quais cada nó possui um ponteiro para o nó pai. Isso simplifica e acelera a remoção em árvores rubro-negras. No entanto, se o espaço em memória for um fator crítico, é possível implementar essas árvores sem o ponteiro para o pai. Neste EP a inserção pôde ser feita sem depender do pai (embora esses ponteiros tiveram que ser ajustados para serem utilizados na remoção). Para que a remoção também não dependesse do pai teríamos que, ao atualizar q para o seu pai no caso 2, descobrir quem era seu pai. Para isso teríamos que gerenciar uma PILHA para armazenar todos os nós do caminho da raiz até o pai do nó a ser removido (a raiz ficando na base da pilha). Os nós seriam empilhados durante a localização do nó a ser removido. E no tratamento do caso 2, o pai de q seria desempilhado.