UNIVERSIDADE FEDERAL DE ITAJUBÁ

CIÊNCIA DA COMPUTAÇÃO

CONVERSÃO DE UMA ÁRVORE 2-3-4 PARA UMA ÁRVORE RUBRO-NEGRA E ANÁLISE DOS COMPORTAMENTOS DE UMA ÁRVORE B DE ORDEM 4

2024007172 - Lucas Alexandre dos Santos Baesso
2024009426 - Rafael Fernando Aurélio Ribeiro

CTCO02 - ALGORITMOS E ESTRUTURA DE DADOS II

Profa. Vanessa Souza

2025

INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
UNIFEI - Itajubá

UNIVERSIDADE FEDERAL DE ITAJUBÁ
INSTITUTO DE MATEMÁTICA E COMPUTAÇÃO
CIÊNCIA DA COMPUTAÇÃO
CTCO02 - ALGORITMOS E ESTRUTURA DE DADOS II

Conversão de uma árvore 2-3-4 para uma árvore rubro-negra e análise dos comportamentos de uma árvore B de ordem 4

---

1 Introdução

As estruturas de dados de árvores balanceadas são fundamentais para diversas aplicações em Ciência da Computação, pois garantem busca, inserção e remoção de elementos em tempo logarítmico mesmo no pior caso. Dentre essas estruturas, as árvores 2-3-4 (*B-trees* de grau mínimo 2) e as árvores Rubro-Negras (*Red-Black Trees*) se destacam por sua eficiência e pelas propriedades de balanceamento que mantêm profundidade controlada. Neste trabalho, implementamos ambas as estruturas em C e exploramos a equivalência teórica e prática entre elas, realizando a conversão direta de uma *2-3-4 Tree* em uma *Red-Black Tree*.

A organização deste relatório segue a sequência lógica do desenvolvimento: primeiro apresentamos o cenário de estudo e conceitos associados às árvores 2-3-4 e Rubro-Negras (Seção 1.1), em seguida explicitamos os objetivos do trabalho (Seção 1.2). Na Seção 2, descrevemos o referencial teórico. A Seção 3 detalha a implementação e os testes de desempenho, e a Seção 4 analisa os resultados obtidos. Por fim, a Seção 5 conclui, apontando contribuições e possíveis extensões.

1.1 Cenário de estudo

As árvores 2-3-4 são uma forma de *B-tree* de grau mínimo 2, em que cada nó armazena de 1 a 3 chaves e possui de 2 a 4 filhos. Essa estrutura é amplamente utilizada em sistemas de arquivos e bancos de dados, pois permite bloqueios de disco eficientes graças ao alto fator de ramificação. Já as árvores Rubro-Negras são árvores binárias de busca auto-balanceadas que utilizam marcações de cor (vermelho ou preto) para garantir que o caminho da raiz a qualquer folha contenha a mesma quantidade de nós pretos, mantendo operações em $O(logn)$.

A equivalência entre essas duas estruturas surge do mapeamento estrutural: cada nó-3 ou nó-4 da 2-3-4 se traduz em "*links* vermelhos" na Rubro-Negra, de forma que a árvore resultante mantenha as propriedades de balanceamento sem necessidade de rotações imediatas. Dessa maneira, uma *2-3-4 Tree* pode ser convertida em uma Rubro-Negra válida em tempo linear, preservando a ordenação e a profundidade balanceada.

No contexto de pesquisa e ensino de estruturas de dados, entender essa conversão facilita a transição entre *B-trees* (mais adequadas para acesso em bloco) e árvores Rubro-Negras (mais usadas em memória). Em sistemas reais, esse algoritmo é útil para converter árvores armazenadas em disco numa forma binária para processamento em memória, sem perder garantias de desempenho.

Este trabalho insere-se no cenário de algoritmos avançados, contribuindo para a compreensão prática dessa equivalência e oferecendo uma biblioteca em C que implementa a conversão direta e as operações básicas de ambas as árvores.

1.2 Objetivo do trabalho

O principal objetivo deste trabalho é implementar em C um algoritmo de conversão direta de uma árvore 2-3-4 em uma árvore Rubro-Negra, de modo que a estrutura resultante seja válida sem necessidade de rotações ou recolorações adicionais. Além da conversão, propusemos implementar todas as operações de inserção, remoção e impressão em ambas as árvores, fornecendo uma interface interativa via menu.

Visando avaliar o comportamento da árvore 2-3-4, elaboramos um conjunto de testes de desempenho que medem métricas como número de *splits*, altura da árvore e quantidade de nós (blocos) em cenários de inserção ordenada, decrescente e aleatória, para entradas de 100, 1.000, 10.000 e 100.000 elementos. Em seguida, aplicamos remoções parciais (10%, 20%, 35% e 50%) registrando *merges*, rotações, altura e quantidade de nós (blocos).

Com essas métricas, poderemos comparar empiricamente a eficiência e o balanceamento da árvore 2-3-4 nos diferentes cenários de carga e remoção, fornecendo *insights* sobre suas vantagens e limitações práticas. Esses resultados também servem de base para compreender como tais operações influenciam a forma final da árvore Rubro-Negra convertida.

Por fim, a próxima seção apresenta o referencial teórico: definimos formalmente as propriedades de *B-trees* e *Red-Black Trees* e detalhamos o algoritmo de conversão, fundamentado em artigos e referências da área.

2 Referencial Teórico

As árvores 2-3-4 são uma forma de *B-tree* de ordem 4, em que cada nó pode armazenar de 1 a 3 chaves e ter de 2 a 4 filhos. A implementação seguida neste trabalho foi baseada no *tutorial* da Programiz, que detalha as operações de inserção, divisão de nó (*split*) e remoção em *B-trees* genéricas, adaptadas ao caso específico de grau mínimo t = 2 e grau máximo = 4. Neste *site*, o funcionamento básico é apresentado como um conjunto de regras recursivas: quando um nó atinge capacidade máxima (*4-node*), divide-se elevando a chave do meio ao pai; na remoção, subfluxos são corrigidos via rotações ou *merge* (*4-node*) de vizinhos para evitar *underflow*. [Programiz].

O algoritmo de inserção inicia descendo recursivamente até uma folha "não cheia" (*preemtive split*), garantindo que nunca se tente inserir em um nó já cheio. Cada divisão (*split*) produz dois nós-2-chave e promove uma chave ao pai, mantendo invariantes de *B-tree*. Já a remoção percorre três casos: se a chave está num nó interno, troca-se pela predecessora ou sucessora em folha; se o filho alvo tem menos de t chaves, faz-se rotação (*borrow*) de um irmão ou *merge* com o pai. [Programiz].

A conversão de uma árvore 2-3-4 para uma árvore Rubro-Negra explora a equivalência estrutural bem conhecida nestes materiais de referência. No estudo do Mead, cada nó-2 (1 chave) converte-se em um nó preto, cada nó-3 (2 chaves) torna-se um nó preto com um filho vermelho, e cada nó-4 (3 chaves) em um nó preto com dois filhos vermelhos. Esses "*links* vermelhos" correspondem internamente aos nós adicionais da 2-3-4, de modo que as propriedades de altura negra constante e ausência de dois vermelhos consecutivos são asseguradas por construção. [Mead].

Os *slides* do Chien and Huang reforçam essa correspondência: eles mostram que toda *2-3-4 Tree* pode ser vista como uma *Red-Black Tree* "expandida" onde os nós vermelhos indicam enlaces internos de nós-3 ou nós-4. A conversão recursiva preserva a *in-order traversal* (mesma ordenação) e mantém a altura de preto idêntica em todos os caminhos, pois cada nível da *B-tree* gera um nó preto na *RB* e apenas "horizontaliza" os nós vermelhos no mesmo nível. [Chien and Huang].

Na prática, implementamos a conversão em C seguindo o raciocínio do Vaity [2016]: a cada nó da 2-3-4, alocamos primeiro o nó pai preto (chave do meio), depois os nós filhos vermelhos (primeira e terceira chaves, quando presentes), e ligamos recursivamente suas subárvores filhos segundo mapeamentos fixos. Cada *link* vermelho nasce sempre de um nó preto, evitando a violação de dois vermelhos consecutivos; e como as folhas da *B-tree* estão todas no mesmo nível, a contagem de nós pretos em qualquer caminho da raiz a uma folha *RB* é igual, satisfazendo a propriedade da altura de preto. [Vaity, 2016].

Em suma, o referencial teórico combina a descrição formal das *B-trees* [Programiz] com o mapeamento bivariado entre 2-3-4 e Rubro-Negra [Mead, Chien and Huang, Vaity, 2016]. Dessa fusão extraiu-se um algoritmo de conversão direto, de complexidade $O(n)$, que produz uma árvore Rubro-Negra válida, pronta para operações posteriores sem necessidade de rotações ou recolorações extras.

3 Desenvolvimento

A conversão de uma árvore 2-3-4 para uma árvore Rubro-Negra fundamenta-se na equivalência estrutural clássica entre os dois modelos. O algoritmo percorre recursivamente cada nó da *B-tree* e aplica um mapeamento local de acordo com o número de chaves presentes: nós com 1, 2 ou 3 chaves geram, respectivamente, um nó preto simples, uma estrutura preto-vermelha ou uma combinação de preto com dois vermelhos. Para estruturar melhor a implementação em C, a conversão foi dividida em duas funções:

- Uma função *static* auxiliar que atua como *handler* recursivo, responsável por converter cada subárvore da 2-3-4 para seu correspondente em *RB*;
- Uma função principal pública, responsável por alocar a estrutura base da árvore Rubro-Negra e chamar o *handler* iniciando a conversão a partir da raiz da *B-tree*.

As duas funções são representadas nos pseudocódigos a seguir, que detalham o comportamento local e global da conversão.

3.1 Pseudocódigo

Algorithm 1: `234paraRB_Handler`(b, $nova$)

Data: b: Nó 2-3-4, $nova$: Árvore rubro-negra
Result: ponteiro para nó RB convertido

if $b = \emptyset$ **then**
  return sentinelaFolha($nova$)
end
$numChaves \leftarrow$ `getBNumChaves`(b);
if $numChaves = 1$ **then**
  corRaizRB ← 'P';
  Converter filhos esquerdo e direito da 2-3-4;
  return raizRB;
end
else if $numChaves = 2$ **then**
  $(k_0, k_1) \leftarrow$ `getBChave`(b, $[0, 1]$);
  corRaizRB ← 'P';
  Criar nó 'V' com k_0;
  Nó 'V' ← filho esq da raizRB;
  Converter filho dir do nó 'P' ← filhoB[2];
  Converter filhos do nó 'V' ← filhosB[0, 1];
  return raizRB;
end
else
  $(k_0, k_1, k_2) \leftarrow$ `getBChave`(b, $[0, 1, 2]$);
  corRaizRB ← 'P';
  Criar nó 'V' com k_0 e nó dir com k_2;
  Configurar pais e ligações;
  Converter filhos do nó 'V' da esq ← filhosB[0, 1];
  Converter filhos do nó 'V' da dir ← filhosB[2, 3];
  return raizRB;
end

A função acima é responsável por percorrer cada nó da árvore 2-3-4 e construir, de forma recursiva, os respectivos nós da *RB-Tree*, respeitando as transformações descritas para os casos de 1, 2 ou 3 chaves. Ela retorna um ponteiro para o nó Rubro-Negro criado e é usada apenas internamente.

Algorithm 2: `converte234paraRB`($raizB$)

Data: $raizB$: raiz de uma árvore 2-3-4
Result: $novaRB$: árvore Rubro-Negra após conversão

Aloca $novaRB \leftarrow$ `alocaArvoreRubroNegra()`
if $raizB = \emptyset$ **then**
  return $novaRB$
end
`converte234ParaRB_Handler`($raizB$, $novaRB$);
`// Garante que o nó raiz da RB seja preto`
$raizCor \leftarrow$ `'P'`;
return $novaRB$;

Este procedimento em $O(n)$ percorre todos os n elementos da árvore B exatamente uma vez, criando ao final uma *RB-tree* válida, pronta para operações subsequentes sem rotações ou recolorações adicionais.

Em linhas gerais, o algoritmo percorre cada nó da 2-3-4 em pré-ordem e, conforme o número de chaves:

1. **Nó-2 (1 chave):** Cria um único nó preto com essa chave e recursivamente converte seus dois filhos.
2. **Nó-3 (2 chaves):** Aloca primeiro o nó pai preto com a segunda chave, depois um nó filho vermelho com a primeira chave; o filho vermelho recebe as duas subárvores correspondentes, e o pai preto recebe a terceira subárvore.
3. **Nó-4 (3 chaves):** Cria um nó pai preto com a chave do meio, e dois nós filhos vermelhos com a primeira e a terceira chaves, ligando as quatro subárvores originais de forma análoga. Cada nó vermelho liga-se sempre a um nó preto, garantindo que nunca haja dois vermelhos consecutivos, e como as folhas da *B-tree* estão todas ao mesmo nível, todos os caminhos de raiz a folhas na *RB (Red-Black)* terão o mesmo número de nós pretos, preservando a propriedade da altura de preto.

Em cada etapa, por construção:

- **Não ocorrem dois nós vermelhos consecutivos**, pois cada nó vermelho nasce diretamente de um pai preto.
- **A altura de preto é preservada**, uma vez que cada nível da *B-tree* gera exatamente um nó preto na *RB*, e os nós vermelhos apenas "horizontalizam" a estrutura sem afetar a contagem de pretos em qualquer caminho.

No módulo de testes, geramos 48 conjuntos de inserção: para cada tamanho $N = 100, 1000, 10000, 100000$ criamos um arquivo em:

- Ordem crescente
- Ordem decrescente
- 10 permutações aleatórias de 1...N (para calcular média das métricas)

Cada arquivo de texto armazena os N valores separados por espaço. Ao rodar os testes de inserção, lemos cada arquivo, inserimos todos os valores na 2-3-4 e registramos, em *metricas_insercao.csv*, as métricas de *splits*, altura e número de nós (blocos).

De modo semelhante, para remoção criamos 48 cenários: utilizamos os mesmos 48 arquivos de entrada, mas a cada caso reconstruímos a árvore completa e, após embaralhar os valores, removemos 10%, 20%, 35% e 50% (separadamente) armazenando em *metricas_remocao.csv* os contadores de rotações, *merge*, altura e blocos ocupados.

Dessa forma, obtemos duas bases de dados completas que permitem calcular médias e desvios das métricas nos testes aleatórios e comparar o desempenho da árvore 2-3-4 em diferentes padrões de carga.

3.2 Área de experimentos

Os testes foram realizados, utilizando um processador Intel® Core™ i9-13900HX com frequência base de 2,20GHz e 16GB de memória RAM DDR5 a 5600MT/s. O sistema operacional utilizado foi o Windows 11 Pro (versão 24H2), e todas as implementações foram desenvolvidas na linguagem C, compiladas com o GCC na versão 6.3.0. Para fins de portabilidade e simplicidade, as únicas bibliotecas utilizadas foram *stdio.h* e *stdlib.h*.

Na próxima seção, apresentaremos as tabelas com as médias obtidas e a análise do comportamento observado em cada cenário de inserção e remoção.

4 Resultados

Nesta seção apresentamos os resultados obtidos nos testes de desempenho da árvore 2-3-4, divididos em inserção e remoção. Para cada métrica, exibimos tabelas contendo médias sobre 10 execuções em ordem aleatória e cenários específicos em ordem crescente e decrescente. Em seguida, analisamos individualmente os comportamentos observados e, ao final, fazemos uma síntese comparativa.

4.1 Inserção

- **Pior caso:** Ordenação (crescente/decrescente)
- **Melhor caso:** Aleatoriedade

4.1.1 Métrica: *Splits*

A ordem de inserção é crítica. Sequências ordenadas forçam *splits* em quase todos os nós, enquanto inserções randômicas promovem melhor distribuição, reduzindo operações de divisão em 43%.

Tabela 1: Inserção: Quantidade de *splits*

Tamanho	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
100	90	90	51,0
1.000	983	983	564,3
10.000	9.979	9.979	5.698,4
100.000	99.974	99.974	56.958,9

Inserções ordenadas (crescente/decrescente) geram aproximadamente 99% de *splits* (ex: 99.974 *splits* para 100.000 elementos)

Inserções aleatórias reduzem *splits* para aproximadamente 57% (56.958,9 *splits* para 100.000)

4.1.2 Métrica: Altura

Inserções aleatórias produzem árvores mais balanceadas. A altura cresce logaritmicamente, mas sequências ordenadas geram estruturas desequilibradas, aumentando caminhos de acesso.

Tabela 2: Inserção: Altura da árvore

Tamanho	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
100	6	6	5
1.000	9	9	7,9
10.000	13	13	10,2
100.000	16	16	13

Ordem ordenada: altura 16 para 100.000 elementos. Ordem aleatória: altura 13 (redução de 19%)

4.1.3 Métrica: Blocos ocupados

Cada *split* consome 1 bloco adicional. Inserções aleatórias economizam 43% de memória versus ordenadas.

Tabela 3: Inserção: Quantidade de blocos ocupados

Tamanho	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
100	96	96	56
1.000	992	992	572,2
10.000	9992	9992	5708,6
100.000	99990	99990	56971,9

4.2 Remoção

- **Impacto da ordem:** Operações em árvores ordenadas exigem até 2,8 vezes mais *merges* e 15% mais rotações;
- **Estratégia ótima:** Remoções em árvores aleatórias reduzem operações de manutenção (*merges*/rotações) em menos que 60% e melhoram a compactação.

4.2.1 Métrica: Rotações

Remoções em árvores sequenciais desbalanceiam a árvore, exigindo correções frequentes via rotações.

Tabela 4: Remoção: Quantidade de rotações

%Remoção	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
10%	1397	1473	1269,3
20%	2698	2611	2449,7
35%	4377	4348	3978,3
50%	6224	6100	5431,7

Remoções em árvores ordenadas exigem até 6.224 rotações (50% elementos). Já em árvores aleatórias requerem 12-15% menos (ex: 5.431,7 para 50%)

4.2.2 Métrica: *Merges*

Merges são custosos. Remoções em árvores randômicas preservam melhor o balanceamento, minimizando fusões de nós.

Tabela 5: Remoção: Quantidade de *Merges*

%Remoção	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
10%	2664	2679	483,2
20%	3966	4003	802,4
35%	5346	5362	1452,9
50%	6469	6524	2281,7

Árvore ordenada: aproximadamente 6.500 *merges* (50% remoções). Árvore aleatória: aproximadamente 2.300 *merges* (redução de 65%)

4.2.3 Métrica: Altura

A 50% de remoções, altura cai para 8 (árvores ordenadas) mas mantém-se 8 (árvores aleatórias), indicando resiliência do balanceamento.

Tabela 6: Remoção: Altura da árvore

%Remoção	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
10%	9	9	9
20%	9	9	8,8
35%	9	9	8,2
50%	8	8	8

Altura mantém-se 9 até 20% de remoções, mesmo em cenários ordenados.

4.2.4 Métrica: Blocos ocupados

Remoções em árvores aleatórias permitem compactação mais eficiente da estrutura.

Tabela 7: Remoção: Quantidade de blocos ocupados

%Remoção	Ordem Crescente	Ordem Decrescente	Ordem Aleatória
10%	7324	7309	5198,5
20%	6022	5985	4880,1
35%	4642	4626	4229
50%	3518	3463	3400

Remoção de 50% em árvores ordenadas: 3.518 blocos (redução de 65% vs. inicial). Remoção de 50% em árvores aleatórias: 3.400 blocos (economia adicional de 3%)

4.3 Análise

A aleatorização nas operações é determinante para eficiência. Cenários ordenados representam o pior caso teórico, enquanto operações randômicas aproximam-se do comportamento ótimo $O(log n)$ em árvores balanceadas.

Portanto, pode-se deduzir algumas recomendações para quando vamos trabalhar com aplicações práticas dessa estrutura. Evitar inserção de dados ordenados. Em casos inevitáveis, aplicar *shuffling* prévio. Em questão de desempenho, monitorar *splits/merges* como indicadores de degradação. Valores maiores que 90% sugerem necessidade de rebalanceamento.

5 Conclusões

Neste trabalho, avaliou-se o problema da equivalência estrutural entre árvore 2-3-4 e *Red-Black Trees*, propondo e implementando um algoritmo de conversão direta em C que dispensa operações adicionais de recoloração ou rotação. A solução adotada segue uma abordagem recursiva, na qual cada nó da árvore 2-3-4 é tratado por um *handler* auxiliar responsável por gerar, em tempo linear, uma subárvore rubro-negra válida.

Os resultados experimentais confirmaram as hipóteses iniciais:

1. **Validade da conversão em $O(n)$:** todas as árvore 2-3-4 testadas foram convertidas com sucesso em *Red-Black Trees* que satisfazem as propriedades de altura de preto constante e ausência de nós vermelhos consecutivos.
2. **Manutenção de propriedades de balanceamento:** verificou-se, por meio de percursos em pré-ordem e checagens estruturais, que a árvore resultante preserva tanto o número de nós pretos em qualquer caminho da raiz às folhas quanto a condição de não haver dois nós vermelhos adjacentes.

Além da fundamentação teórica, desenvolveu-se uma biblioteca em C que oferece operações de inserção, remoção e conversão de uma árvore 2-3-4 em rubro-negra via um menu interativo. O estudo de desempenho incluiu *benchmarks* de *splits*, *merges* e rotações em cenários da árvore em ordem e aleatória, mostrando redução média de 43% no número de *splits* e blocos ocupados em árvores aleatórias, bem como até 15% menos rotações e 65% menos *merges* em remoções em árvores randômicas. Esses resultados apontam para a eficácia prática da estratégia recursiva em situações reais de uso.

Em suma, comprovou-se que a conversão direta proposta é viável e eficiente, abrindo caminho para extensões e aplicações práticas em diferentes contextos de armazenamento e recuperação de dados.

Link do GitHub: https://github.com/01baesso/Conversao_Arvore_2-3-4_RB

Referências

Pao-Chu Chien and Jui-Lin Huang. 2-3tree, 2-3-4 tree & red-black tree. Disponível em: https://smile.ee.ncku.edu.tw/old/Links/MTable/Course/DataStructure/2-3,2-3-4&red-blackTree_952.pdf. Acessado: 30 Jun 2025.

M. Mead. Mapping 2-3-4 trees into red-black trees. Disponível em: https://pontus.digipen.edu/~mmead/www/Courses/CS280/Trees-Mapping2-3-4IntoRB.html. Acessado: 30 Jun 2025.

Programiz. Deletion from a b-tree. Disponível em: https://www.programiz.com/dsa/deletion-from-a-b-tree.

Yogesh Umesh Vaity. Converting a 2-3-4 tree into a red black tree. Disponível em: https://stackoverflow.com/questions/35955246/converting-a-2-3-4-tree-into-a-red-black-tree, 2016.