Relatório Projeto AS - Otavio Bettega

Estruturas de Dados e Algoritmos - Prof. Rafael de Pinho
Julho, 2024

Introdução:

As árvores rubro-negras são estruturas de dados balanceadas que garantem operações eficientes de inserção, remoção e busca, mantendo propriedades que asseguram um desempenho consistente. Para isso, essa estrutura de dados está sempre se corrigindo e reorganizando através de rotações e recolorizações. Estas árvores são versões especiais das árvores binárias de busca, onde cada nó possui uma cor que pode ser vermelha ou preta. Ela tem 5 propriedades fundamentais que devem ser respeitadas para que se tenham os benefícios de eficiência.

1. Um node é sempre preto ou vermelho;
2. A raiz original da árvore é preta;
3. Todos os ponteiros nullptr de seus nodes (folhas) são pretos;
4. Nodes vermelhos somente tem filhos pretos;
5. Todos os caminhos que levam da raiz original às folhas tem o mesmo número de nodes pretos.

Implementação:

Para criar uma árvore rubro-negra, criamos a estrutura node que representará nossos elementos. Nosso node tem valor inteiro, dois ponteiros para seus nodes filhos, cor, e um ponteiro para seu parent (pai).

1. Função de criação: createNode(int iData)

- Propósito: Cria um novo node com um dado valor inteiro, e o retorna.
- Detalhes: Node é criado com memória alocada dinamicamente. Seus pointers iniciais são todos nullptr pois especificamos sua posição na árvore fora desta função. Note que todas os nodes são criados com cor vermelha, e então a árvore deve ser ajustada para acomodá-lo.

2. Funções de Rotação: rotateLeft(Node* root, Node* nodeA) e rotateRight(Node* root, Node* nodeA)

- Propósito: Realizam "rotações" para corrigir violações das propriedades da árvore durante operações de inserção e remoção.
- Detalhes: As rotações são transformações fundamentais para manter o balanceamento da árvore. A rotação à esquerda (rotateLeft) e à direita (rotateRight) reorganizam os nós conforme necessário, garantindo que as propriedades da árvore rubro-negra sejam mantidas. Essa transformação é melhor entendida vendo os comentários diretamente no código.

3. Funções de Inserção: insert(Node* root, int iData) e insertNode(Node* root, Node* newInsert)

- Propósito: Utilizam createNode para inserir um novo node na árvore rubro-negra. Utilizando as funções de rotação e recolorizando nodes, mantém garante que a árvore mantenha suas propriedades.
- Detalhes: A função insert gerencia a inserção de um novo nó na árvore, chamando insertNode para percorrer a estrutura e posicionar o node. Note que o novo node é inserido corretamente em relação a uma árvore ordinária, somente comparando valor dos nodes que passa para se posicionar. Após a inserção, é realizada uma verificação com fixInsertion para garantir que as propriedades da árvore sejam preservadas.

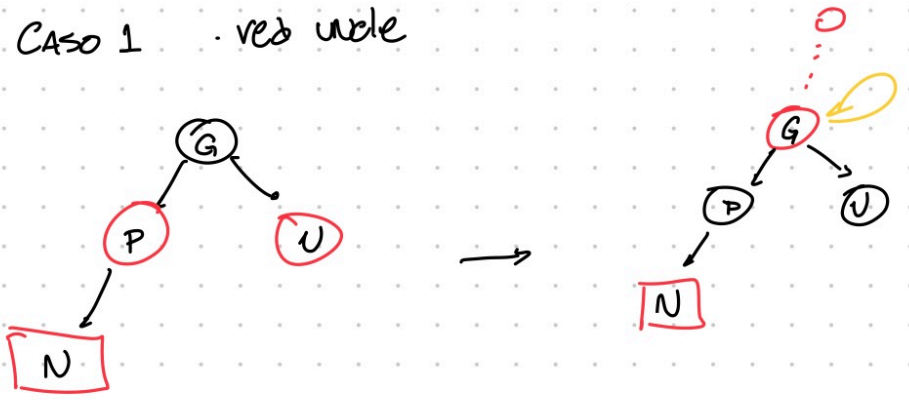
4. Função de inserção: fixInsertion(Node* root, Node* currentNode)

- Propósito: Corrige violações das propriedades da árvore após a inserção de um novo nó.
- Detalhes: Esta função é essencial para manter o balanceamento da árvore após uma inserção. Ela ajusta a estrutura da árvore através de rotações e recolorações conforme necessário, garantindo que todas as propriedades da árvore rubro-negra sejam cumpridas. Ela começa no nosso node inédito e sobe a árvore recursivamente até chegarmos a raiz original.

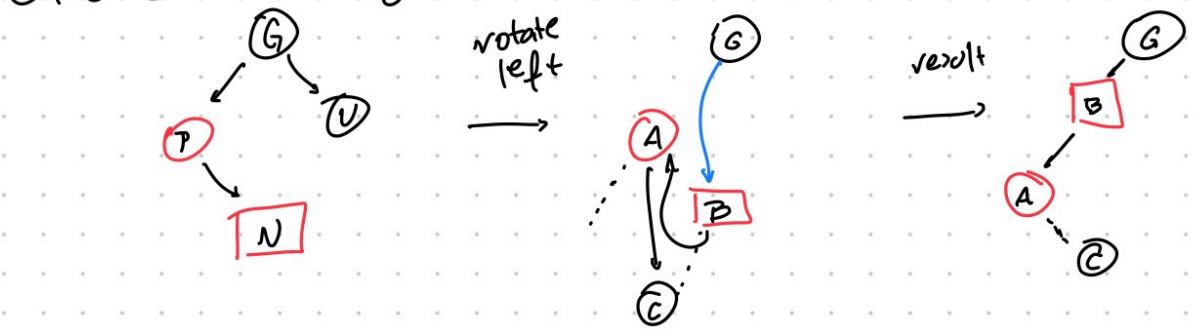
  Os comentários no código demarcam os casos que a função lida, e o passo a passo de como ela funciona. Referenciando a esses comentários, podemos complementá los com alguns diagramas que mostram o que está acontecendo localmente em uma árvore. (note que o diagrama somente demonstra as transformações nos casos especificados por "Esquerda" no código, mas os de "Direita" representam as mesmas transformações, somente espelhadas.

ESQUERDA:

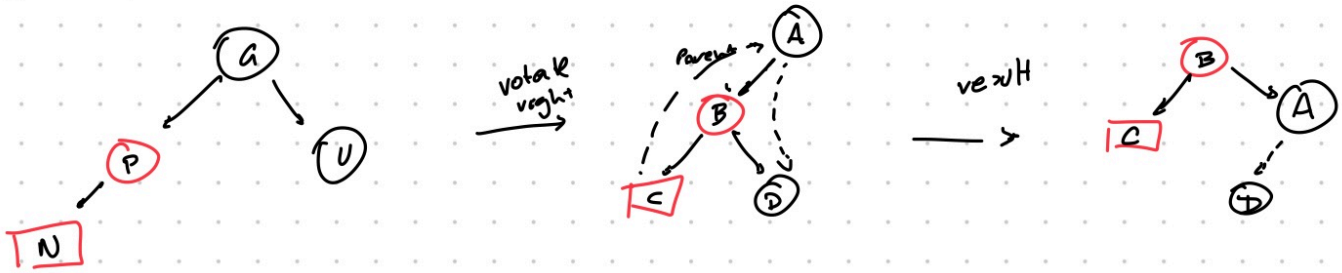
CASO 1 · red uncle



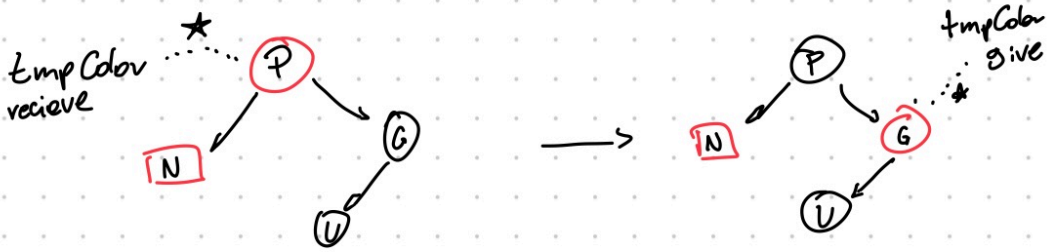
CASO 2 · triangula



CASO 3 · linear

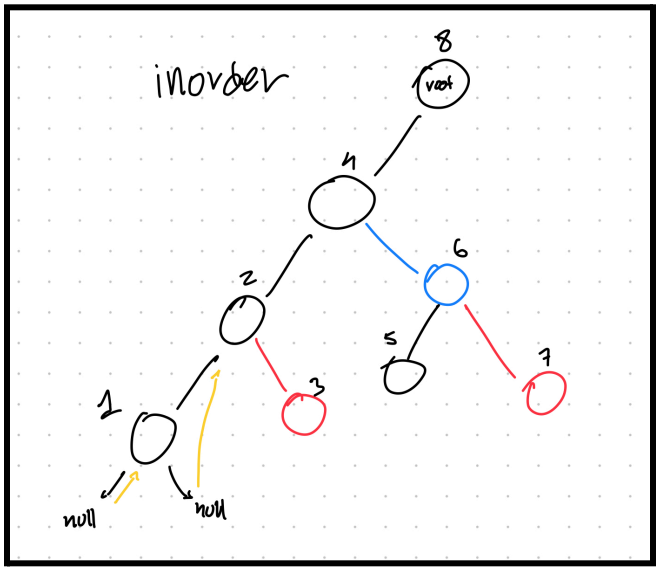


Recolor:



5. Função de percorrer: inorder(Node* root)

- Propósito: Realiza um percurso em ordem simétrica (inorder) na árvore para imprimir seus elementos em ordem crescente.
- Detalhes: Esta função percorre recursivamente a árvore, visitando os nós na ordem correta e imprimindo os valores dos nós, o que é útil para debugar e verificar a estrutura correta da árvore. Ela percorre pela esquerda da árvore até uma folha, e com as chamadas na stack, gradualmente vai completando as partes à direita. Esse percurso pode ser visto seguindo a ordem mostrada no diagrama:



6. Função de busca: searchValue(Node* root, int iValue)

- Propósito: Busca por um valor específico na árvore rubro-negra, retornando o node com esse valor
- Detalhes: Realiza uma busca binária na árvore para encontrar o nó que contém o valor especificado. Isso é útil para recuperar dados específicos da árvore, ou para a remoção de dados. Vale notar que um dos grandes benefícios da árvore rubro-negra é a garantia que uma tal busca sempre terá complexidade O(log n), pois a árvore é balanceada.

7. Funções mínimo e máximo: minValue(Node* root) e maxValue(Node* root)

- Propósito: Encontram o nó com o valor mínimo e máximo na árvore, respectivamente.
- Detalhes: Essas funções percorrem a árvore para encontrar o nó mais à esquerda (mínimo) ou mais à direita (máximo), fornecendo acesso rápido aos extremos da árvore.

8. Função altura: height(Node* root)

- Propósito: Calcula a altura máxima da árvore rubro-negra.

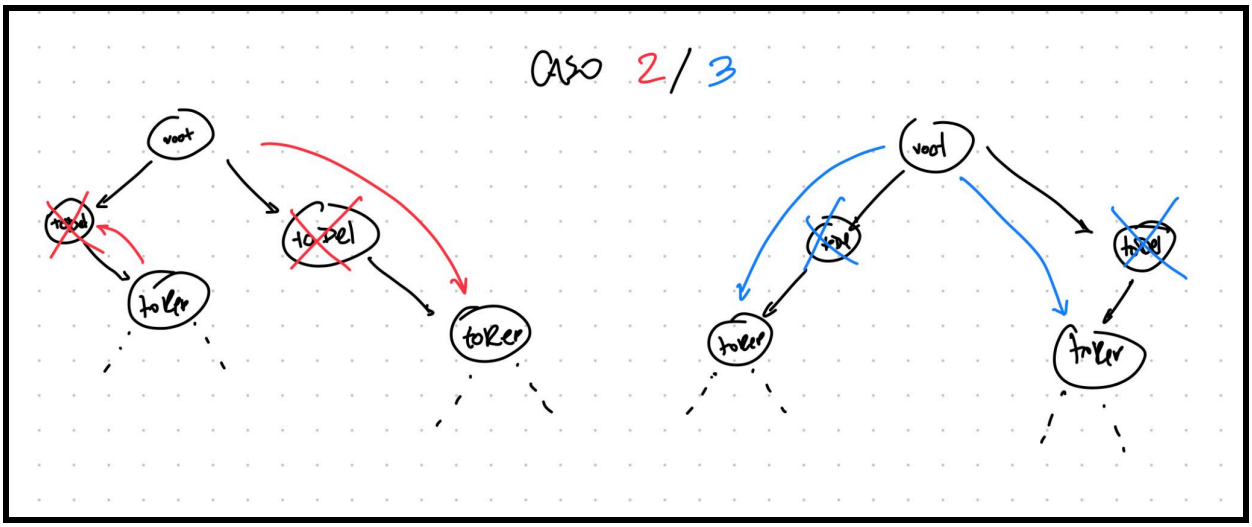
- Detalhes: Utiliza recursão para determinar a altura da árvore, contando o número de níveis a partir da raiz. Note que somente o caminho da raiz até a folha mais longo é retornado.

9. Funções de Validação: validTreeTesting(Node* currentNode, int* blackTally, int blackCount) e validTreeTest(Node* root)

- Propósito: Verificam se a árvore satisfaz as propriedades de uma árvore rubro-negra.
- Detalhes: As funções de validação verificam se todas as propriedades da árvore rubro-negra estão sendo obedecidas. Elas verificam recursivamente que nodes vermelhos não tem filhos vermelhos. Além disso, com um contador é possível fazer a contagem de nós pretos nos caminhos da raiz às folhas, garantindo o balanceamento.

10. Funções de Remoção: fixDeletion(Node* root, Node* currentNode) e deleteNode(Node* root, int iData)

- Propósito: Removem um nó específico da árvore rubro-negra enquanto mantêm suas propriedades.
- Detalhes: A função deleteNode procura o node com o nó especificado da árvore, e o deleta ajustando de caso a caso. O caso 2 e 3 são apresentados na imagem. Após a deleção, é chamado fixDeletion para restaurar o balanceamento da árvore e garantir colorização e balanceamento correto. O funcionamento das funções é descrito no código.



Testes:

Para efetivamente realizar os testes, foram criadas duas árvores em ordens distintas. Root 1 é criado de forma aleatória, e são efetuados testes de sua validade constantemente. Vemos que a função inorder consegue nos apresentar a ordem correta dos nodes, significando

que a árvore foi criada com sucesso. Vemos também que a função de busca encontra o valor 10 corretamente, e que a função de remoção consegue o remover, sem prejudicar a validade da árvore. Em seguida a função de busca não encontra o - agora removido - elemento 10.

Para a segunda árvore, Root2, a inserção dos seus elementos foi feita de forma quase linear no começo, algo que força a árvore a fazer muitas rotações, visto que sua raiz original está sempre mudando para acomodar os novos nodes e manter-se balanceada. Também foram inseridos números muito próximos uns dos outros de forma alternada, forçando a árvore a fazer vários ajustes. Novamente, vemos que a árvore é montada com sucesso, visto que inorder consegue percorrê-la sem erros, e nossos testes de validação são positivos. Em Root2, foi feita a remoção de 2 elementos, que eram cercados com possíveis filhos, e vemos que não só inorder mostra que a árvore corretamente consegue se ajustar aos restantes valores, como também nossos testes mostram que ainda é rubro-negra.

Resultados (da linha de comando):

```
====================================
Teste com Root1

Percuso Inorder root1:
1 2 3 5 7 8 10 11 18 22 26 118 23456

Teste de validade da árvore: Válida


Busca por valor 10:
Encontrado valor 10, cor: BLACK

Apagando valor 10:
1 2 3 5 7 8 11 18 22 26 118 23456

Teste de validade da árvore: Válida


Busca por valor 10:
Não encontrado valor 10.

Valor Maximo: 23456
Valor Minimo: 1

Altura: 5

Teste de validade da árvore: Válida

====================================
Teste com Root2

Percuso Inorder root1:
1 2 3 4 5 6 399 400 401 402 499 500 501
```

Teste de validade da árvore: Válida

Busca por valor 6:
Encontrado valor 6, cor: BLACK

Árvore apagando valor 6:
1 2 3 4 5 399 400 401 402 499 500 501

Teste de validade da árvore: Válida

Busca por valor 400:
Encontrado valor 400, cor: BLACK

Árvore apagando valor 6:
1 2 3 4 5 399 401 402 499 500 501
Valor Maximo: 501
Valor Minimo: 1

Altura: 5

Teste de validade da árvore: Válida

Conclusão:

Dentre as estruturas de dados que estudamos durante o semestre, esta era certamente a menos trivial. Dito isso, é importante reforçar a utilidade e praticidade desta árvore comparada a estruturas mais ordinárias. Vemos que elementos podem ser inseridos e removidos sem preocupação com ordem, diferente de stacks, e que a inserção e busca de novos elementos é muito eficiente relativo ao possível tamanho da árvore, comparado a listas. Até comparada a árvores binárias ordinárias, a rubro-negra é superior devido ao fato que seu tamanho é consistente em seus branches, dessa forma independente da ordem de sua criação, ela mantém-se balanceada e com complexidade de busca constantemente em O(log n). É necessário perdoar as complexidades trazidas com suas regras para desfrutarmos do seu valor.

Link do Repositório: https://github.com/OtavioBettega/AS-EDD-2024.1