Estrutura de Dados

Avaliação Substitutiva

Alunos:

Eduardo Alves & Saulo Pires

Professor:

Dr. Rafael de Pinho

Fundação Getúlio Vargas

Rio de Janeiro, Brasil

Julho 2024

Introducão

Uma *Árvore Rubro-Negra* é uma árvore binária com um bit extra de armazenamento: sua cor, a qual pode ser preta ou vermelha.
Foi proposta em 1972 por Rudolf Bayer e, ao longo dos anos, seus algoritmos de inserção, deleção, entre outros, foram aprimorados.

A estrutura de uma Árvore Rubro-Negra pode ser definida a partir de suas propriedades, de maneira recursiva:

- Todo nó é preto ou vermelho;
- A raiz é preta;
- Todo nó nulo é preto;
- Se um nó é vermelho, ambos seus nós filhos são pretos;
- Para cada nó, todos os caminhos a partir desse nó para suas folhas contem a mesma quantidade de nós pretos.

A *Struct* usada nesse trabalho para definir um nó dessa árvore foi o seguinte:

```
enum Color {
    RED,
    BLACK
};

template<typename T>
struct Node {
    T payload;
    Node* ptrParent;
    Node* ptrLeft;
    Node* ptrRight;
    Color color;
};
```

Usando um *enum* para definir a cor do nó.[1]
No contexto do trabalho desenvolvido sobre árvores Red-Black em C++, foi aplicado o conceito de instanciação explícita para operar com diferentes tipos de dados, como inteiros, caracteres e floats. Essa abordagem permitiu adaptar as operações de inserção, remoção, busca e outras, para cada tipo específico de dado, enquanto compartilhava a mesma lógica subjacente implementada nas funções genéricas.

[1] Uma prática comum é utilizar um booleano para tal.

Lema

Uma árvore rubro-negra com n nós internos tem altura no máximo $2\log_2(n+1)$.

Prova:
Começamos mostrando que a subárvore enraizada em qualquer nó x contém pelo menos $2^{bh(x)-1}$ nós internos, onde $bh(x)$ é o número de nós pretos da raiz até uma folha qualquer, ou seja a altura preta da arvore. Vamos escrever $n(x)$ simbolizando o número de nós enraizados em x. Demonstramos essa afirmação por indução.

- **Base da indução:** Se a altura de x é 0, então x deve ser uma folha , e a subárvore enraizada em x contém pelo menos $2^{bh(x)-1} = 2^0 = 1$ nó interno, o que é verdadeiro.
- **Passo da indução:** Considere um nó x com pelo menos um filho não nulo, sejam x_1, x_2 seus filhos, se x_i é preto, então $bh(x) = bh(x_i) + 1$, se x_i é vermelho, então $bh(x) = bh(x_i)$. Sabemos que vale:

$$\begin{aligned} n(x) &= n(x_1) + n(x_2) - 1 \\ &\geq 2^{bh(x_1)} + 2^{bh(x_2)} - 1 \\ &\geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \\ &= 2^{bh(x)} - 1 \end{aligned}$$

  Aqui usamos a nossa hipótese. Logo temos:

$$n(x) \geq 2^{bh(x)} - 1 \Leftrightarrow n(x) + 1 \geq 2^{bh(x)}$$

Para completar a prova do lema, seja h a altura da árvore. De acordo com a propriedade 4, pelo menos metade dos nós em qualquer caminho simples da raiz a uma folha, excluindo a raiz, devem ser pretos. Consequentemente, a altura-preta da raiz deve ser pelo menos $h/2$, e assim,

$$n(x) + 1 \geq 2^{h/2} - 1 \Leftrightarrow 2\log_2(n(x)+1) \geq h(x)$$

Como consequência imediata deste lema, cada uma das operações dinâmicas de conjunto (`SEARCH, MINIMUM, MAXIMUM`) rodam em tempo $O(\log n)$ em uma árvore rubro-negra.

Implementações:

Inserção de um nó:

A função `insertNode` é responsável pela inserção de um novo nó em uma árvore rubro-negra. A função recebe como parâmetros um ponteiro duplo para a raiz da árvore e o valor a ser inserido.
Primeiramente, a função cria um novo nó com o valor a ser inserido e percorre a árvore a partir da raiz para encontrar a posição adequada. Durante a travessia, compara-se o valor

do novo nó com os valores dos nós existentes para determinar se ele deve ser inserido à esquerda ou à direita.
Após encontrar a posição, o novo nó é inserido na árvore e seus ponteiros pai e filhos são ajustados adequadamente. Se a árvore estava vazia, o novo nó se torna a raiz. Caso contrário, ele é inserido como filho do nó encontrado durante a travessia.
Depois de inserir o novo nó, a função `fixInsertRedBlack` é chamada para corrigir quaisquer violações das propriedades da árvore rubro-negra. Esta função é essencial para manter o balanceamento da árvore e garantir que todas as propriedades mencionadas anteriormente sejam mantidas.
A função `fixInsertRedBlack` se apropria das funções `rotationLeft` e `rotationRight`. Tais são utilizadas para fazer as rotações e corrigir as cores dos nós após ser colocado um novo nó.
Embora a inserção inicial do nó siga o mesmo padrão de uma árvore binária de busca, a correção das violações da árvore rubro-negra é o que diferencia este tipo de árvore, garantindo uma altura balanceada e, consequentemente, operações eficientes.

Remoção de um nó:

A função `removeNode` é responsável pela remoção de um nó em uma árvore rubro-negra. A função recebe como parâmetros um ponteiro duplo para a raiz da árvore e o valor do nó a ser removido.
Primeiramente, a função encontra o nó a ser removido utilizando a função `searchNode`. Se o nó não for encontrado, a função simplesmente retorna. Caso contrário, inicia-se o processo de remoção.
Se o nó a ser removido não possui um filho à esquerda, ele é substituído pelo seu filho à direita. Caso o nó a ser removido não possua um filho à direita, ele é substituído pelo seu filho à esquerda. Quando o nó a ser removido possui ambos os filhos, o sucessor é o nó de menor valor na subárvore à direita do nó a ser removido. O sucessor é removido da sua posição original e substitui o nó a ser removido, herdando seus filhos e sua cor.
A substituição de um nó pelo seu filho ou sucessor é realizada pela função `transplant`, esta função ajusta os ponteiros pai e filhos para manter a integridade da árvore.
Após o nó ser removido, ele é liberado da memória. Se a cor original do nó removido era preta, a função `fixRemoveRedBlack` é chamada para corrigir quaisquer violações das propriedades da árvore rubro-negra.

Verificação das Propriedades:

A função `verifyRedBlack` e suas funções auxiliares são responsáveis por verificar três propriedades fundamentais da árvore rubro-negra. Primeiramente, verifica-se se a raiz da árvore é preta. Se a raiz não for preta, isso constitui uma violação das propriedades da árvore rubro-negra.
A segunda propriedade verificada se todos os nós vermelhos possuem filhos pretos. A função `verifyRedProperty` é utilizada para percorrer a árvore de forma recursiva e verificar se nenhum nó vermelho possui filhos vermelhos.
Além disso, a função `verifyBlackProperty` conta o número de nós pretos em todos os caminhos da raiz até as folhas. Esta verificação é realizada para garantir que todos os caminhos da raiz até qualquer folha contenham o mesmo número de nós pretos.

Caso alguma das propriedades seja violada durante a verificação, a função `verifyRedBlack` retornará `false` e exibirá uma mensagem de erro indicando qual propriedade foi comprometida.

Cálculo da Altura da Árvore:

As funções `treeHeight` e `treeHeightOptimized` são responsáveis por calcular a altura de uma árvore binária, considerando diferentes abordagens para otimização, especialmente relevante em árvores rubro-negras.
A função `treeHeight` calcula a altura da árvore de maneira convencional, utilizando recursão para percorrer todos os nós da árvore. Ela inicia verificando se o nó atual é nulo (ou seja, uma folha), retornando -1 para indicar uma altura nula. Caso contrário, calcula as alturas das subárvores esquerda e direita recursivamente e retorna o máximo entre essas alturas, adicionando 1 para incluir o próprio nó.
A função `treeHeightOptimized` é otimizada para árvores rubro-negras, considerando que caminhos com mais nós vermelhos são mais longos, devido à propriedade de que todo caminho da raiz à folha tem o mesmo número de nós pretos. A função verifica se o nó tem um filho vermelho; se tiver, calcula a altura apenas para esse lado vermelho. Caso ambos os filhos sejam pretos ou vermelhos, a função chama recursivamente para ambos os lados, garantindo assim que os caminhos mais longos, com nós vermelhos, sejam considerados na altura calculada.

Operações Padrão em Árvores Rubro-Negras:

As operações básicas em uma árvore rubro-negra, como busca por um nó específico, percurso em ordem simétrica (inorder), encontrar o nó máximo e encontrar o nó mínimo, são similares às operações em uma árvore binária de busca padrão. Em uma árvore rubro-negra, não há otimizações específicas nessas operações como existe para o cálculo da altura..
O tempo de execução dessas operações em uma árvore rubro-negra é $O(\log n)$, onde n representa o número de nós na árvore. Isso se deve ao fato de que a altura da árvore rubro-negra é limitada por $2\log_2(n+1)$, garantindo que as operações de busca e acesso aos extremos (mínimo e máximo) sejam realizadas de forma eficiente, similar a uma árvore binária de busca balanceada.
Portanto, embora as operações básicas sejam análogas às de uma árvore binária de busca, a estrutura adicional das cores dos nós em uma árvore rubro-negra contribui para manter o balanceamento e garantir que as propriedades sejam preservadas ao longo das operações dinâmicas na árvore.

Testes

No nosso arquivo main, `redBlackTree`, temos testes para verificar a eficácia de nosso programa.
Abaixo, temos a criação de uma árvore onde cobrimos todos os casos de inserção. Note que depois de cada inserção, chamamos a função `verifyRedBlack` para confirmamos que ainda temos uma árvore rubro-negra.

```
#=#=#=#=#=#=# Insertion Tests #=#=#=#=#=#=#

After each step, we use the function 'verifyRedBlack', which returns '0'
if the tree is not a red-black tree and '1' if it is

Adding nodes 100, 80 and 110:

 +---100
    +---80
    +---110

Verifying red-blach property: 1


Case A: The node's parent is the left child of its parent:

Adding node 70. We have Case A1: Uncle is RED.
+---100
    +---80
    |   +---70
    +---110

Verifying red-black property: 1


Adding node 75. We have case A2: Uncle is BLACK and current node is a
right child
+---100
    +---75
    |   +---70
    |   +---80
    +---110

Verifying red-black property: 1


Adding node 65.
+---100
```

```
    +---75
    |   +---70
    |   |   +---65
    |   +---80
    +---110

Verifying red-black property: 1


Adding node 60. We have Case A3: Uncle is BLACK and current node is a
left child
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    |   +---80
    +---110

Verifying red-black property: 1


Case B: The node's parent is the right child of its parent:

Adding node 115. We have case B1: Uncle is RED
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    |   +---80
    +---110
        +---115

Verifying red-black property: 1


Adding node 120. We have Case B2: Uncle is BLACK and current node is a
right child
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    |   +---80
    +---115
        +---110
        +---120
```

```
Verifying red-black property: 1


Adding node 105
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    |   +---80
    +---115
        +---110
        |   +---105
        +---120

Verifying red-black property: 1


Adding node 117. We have Case B3: Uncle is BLACK and current node is a
left child
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    |   +---80
    +---115
        +---110
        |   +---105
        +---120
            +---117

Verifying red-black property: 1
```

Agora, o teste de máximos e mínimos com a árvore criada no teste passado.

```
#=#=#=#=#=#=# Max and Min Tests #=#=#=#=#=#=#

Maximum payload in the tree: 120


Minimum payload in the tree: 60
```

Dois testes para a função `searchNode`. Um teste onde procuramos um nó existente na árovre, outro teste onde não há o nó procurado.

```
#=#=#=#=#=#=# Finding Test #=#=#=#=#=#=#

Using the function 'searchNode' to find the node 117. The function
returns the node itself:
Payload of ptrSearchNode: 117

The function works.

Shall we find one nonexistent node in the tree we've made. For example
the node 666.
Error: Node with value 666 not found.
```

Na função `traverseInOrder` criada, que printa a árvore em ordem, também é colocado as cores dos nós.

```
#=#=#=#=#=#=# In Order Test #=#=#=#=#=#=#

Using the 'traverseInOrder' function we have:
60-R 65-B 70-R 75-R 80-B 100-B 115-R 110-B 115-R 117-R 120-B
```

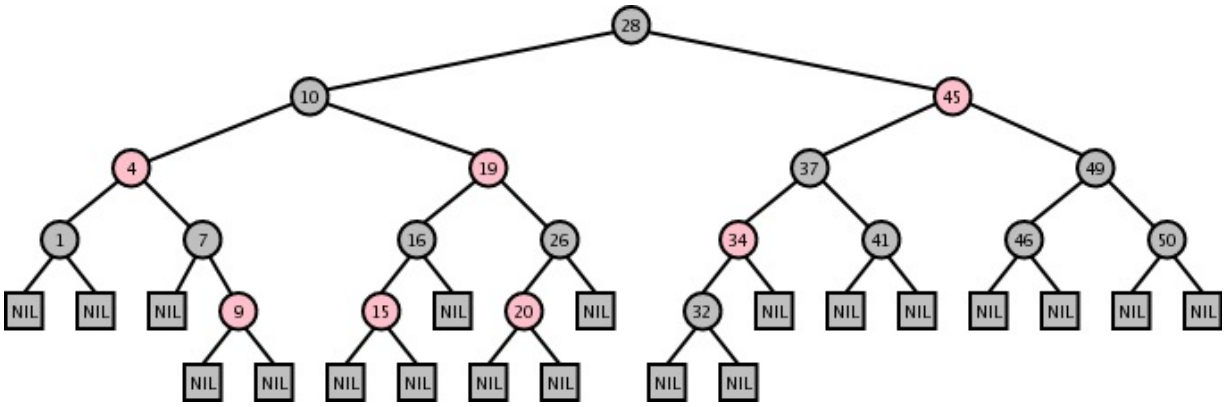
Abaixo, testes onde cobrimos todos os casos possíveis da ação de tirar um nó da árvore rubro-negra.

```
#=#=#=#=#=#=# Removing Tests #=#=#=#=#=#=#


We are going to test the 'removeNode' function using the tree we've
created.
After each step, we use the function 'verifyRedBlack', which returns '0'
if the tree is not a red-black tree and '1' if it is

Removing node 8. Case 1: The node doesn't have children.
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    +---115
        +---110
        |   +---105
        +---120
            +---117

Verifying red-black property: 1
```

```
Removing node 120. Case 2: The node has the left children.
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    +---115
        +---110
        |   +---105
        +---117

Verifying red-black property: 1

Removing node 115. Case 3: The node has both childrens.
+---100
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    +---117
        +---110
        |   +---105

Verifying red-black property: 1

Removing node 100. Case 4: Removing the root.
+---105
    +---75
    |   +---65
    |   |   +---60
    |   |   +---70
    +---117
        +---110

Verifying red-black property: 1
```

Conjectura: Existem árvores que obedecem as propriedades rubro-negras, mas não são atingíveis pelos algorítimos clássicos de construção.

Segue abaixo um exemplo de árvore que obedece todas as propriedades das árvores rubro-negras.



Entretanto, observe os nós irmãos 10 e 45. Ambos tem filhos, mas são irmãos com cores diferentes.
Conjecturamos que tais árvores não podem ser construídas com os algorítmos clássicos.
Corrobora com nossa conjectura:

- Os casos inicias, quando temos apenas 3 nós. É impossível fazer os dois primeiros irmão com cores diferentes. Isso pode ser provado por exaustão,. cobrindo todos os 6 casos de inserção.
- É impossivel fazer os dois nós irmão sem filhos terem cores diferentes. De fato, teríamos pelo menos um caminho com uma cor preta a mais que outro.

Otimizações aplicadas:

A função `treeHeightOptimized` calcula a altura de uma árvore red-black de forma otimizada, considerando caminhos que alternam entre nós vermelhos seguidos por nós pretos.
Nesta função, além do caso base onde `ptrRoot` é `nullptr`, são considerados dois tipos específicos de caminhos: caminhos que começam com um nó vermelho seguido por um nó preto à esquerda, e caminhos que começam com um nó vermelho seguido por um nó preto à direita. Em ambos os casos, a função retorna a altura aumentada em 1 mais a altura da subárvore correspondente, refletindo a interpretação estrita das propriedades de uma árvore red-black. Se nenhum desses caminhos específicos é encontrado, a função calcula recursivamente as alturas das subárvores esquerda e direita e retorna 1 mais o máximo dessas alturas, representando a altura real da árvore.
A conjectura que formulamos é crucial para essa função, porque se tivéssemos a árvore mostrada na imagem com o nó 34 sendo preto e deletássemos o nó 32, nossa função estaria incorreta. Isso ocorreria porque ela chamaria a recursão apenas para o lado direito, enquanto observamos que a altura da sub-árvore no lado esquerdo é maior.
Ainda não estamos completamente certos da nossa otimização, mas decidimos apresentar os resultados que obtivemos. Nos testes que realizamos, em todas obtivemos resultados iguais nas duas funções de altura.

```
#=#=#=#=#=#=# Height test #=#=#=#=#=#=#

We are going to test the funtions 'treeHeight' and 'treeHeightOptimized'.
The following results is the mean of 10 iterations, using trees with
1000000 nodes each, in both functions:

Average execution time of treeHeightOptimized: 10446490 nanoseconds
Average execution time of treeHeight:          81542042 nanoseconds
```

Conclusão

Neste projeto, exploramos detalhadamente a implementação e análise de árvores Red-Black, uma estrutura de dados fundamental na ciência da computação devido à sua capacidade de manter o balanceamento dinâmico durante operações de inserção, remoção e busca.
Além disso, identificamos desafios e questões teóricas, como a conjectura sobre a construção de árvores que obedecem às propriedades Red-Black, mas que não são alcançáveis pelos algoritmos clássicos de construção. Essas questões abrem caminho para futuras investigações e refinamentos na teoria e prática das estruturas de dados.
Em resumo, este trabalho não apenas aprofundou nossa compreensão das árvores Red-Black, mas também destacou a importância de métodos rigorosos de implementação e análise para garantir a robustez e eficiência de estruturas de dados fundamentais na computação moderna.

Referências

[1] https://www.programiz.com/dsa/red-black-tree

[2] https://www.geeksforgeeks.org/introduction-to-red-black-tree/#1-insertion

[3] "Introduction to Algorithms"by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

[4] Link do código para printar a árvore binária https://stackoverflow.com/questions/36802354/print-binary-tree-in-a-pretty-way-using-c