The Falling Leaves Problem

Ayrton Denner da Silva Amaral[1], **Fabrício Alves de Freitas**[1], **José Adenaldo Santos Bittencourt Júnior**[1], **Pedro Vitor Quinta de Castro**[1]

[1] Instituto de Informática – Universidade Federal de Goiás (UFG)
Goiânia – GO – Brazil

ayrtondenner_2013@hotmail.com,

{fabricioalvesdefreitas, joseadenaldo, pvcastro}@gmail.com

Abstract. *This extended scientific summary is a partial requirement of the Project 2018/1 of the Estruturas de Dados e Projeto de Algoritmos (CCO0348) taught by Prof.ª. Dr.ª. Marcia Cappelle. The 1525 Falling Leaves problem was selected in the UVa Online Judge (UVa) repository which consists of performing a sequence of operations on a binary search tree and printing the paths in depth and width. In the implementation of the solution in C++ we used a Stack to keep the input data and Stack and Queue for the depth and breadth traversals of the Binary Search Tree, respectively. The general analysis of the complexity of the algorithms was supported by Big-O notation. The proposed solution of the Binary Search Tree used four sets of input data containing 31, 63, 127 and 255 elements.*

Resumo. *Este resumo científico estendido é um requisito parcial do Projeto 2018/1 da disciplina Estruturas de Dados e Projeto de Algoritmos (CCO0348) ministrado pela Prof.ª. Dr.ª. Marcia Cappelle. Foi selecionado o problema 1525 Falling Leaves no repositório UVa Online Judge (UVa) que consiste em realizar uma sequência de operações em uma árvore de busca binária e impressão dos percursos em profundidade e largura. Na implementação da solução em C++ foram utilizadas uma Pilha para manter os dados de entrada e Pilha e Fila para os percursos de profundidade e largura, respectivamente. A análise geral da complexidade do algorítmo foi suportada pela notação Big-O. A solução proposta da Árvore de Busca Binária utilizou quatro conjuntos de dados de entrada contendo 31, 63, 127 e 255 elementos.*

Introdução

A estrutura de dados **Árvore de Busca Binária** é organizada, como o nome sugere, de forma que cada nó correspondente a um elemento da árvore seja um objeto. Esse objeto possui um pai e no máximo dois filhos: um à esquerda e um à direita. Quando um nó não possui pai significa que ele é a raiz da árvore, ou seja, o primeiro nó; o que precede todos [Cormen T. H. 2012].

Toda árvore de busca binária é armazenada de maneira a satisfazer a seguinte propriedade: *"Considere n um nó da árvore de busca binária T. Se x é um nó na sub-árvore à esquerda de n, então x.chave <= n.chave. Se x é um nó na sub-árvore à direita de n, então x.chave >= n.chave"* [Sedgewick R. 2011].

É uma propriedade da árvore de busca binária, o fato de que sempre que percorremos seus nós, partindo dos elementos mais profundos e à esquerda possíveis, até os mais profundos à direita, teremos os valores sempre ordenados. Nós de uma árvore que não possuem filhos são considerados folhas, ou seja, os nós mais nas extremidades da árvore. Já o nó que precede todos, é chamado de nó raiz. Esse nó nunca possui pai [Szwarcfiter J.L. 2010].

Árvores de busca binárias tem uma natureza recursiva, considerando que cada nó pode possuir uma sub-árvore à sua esquerda ou à direita [Sedgewick R. 2011].

O tipo abstrato de dados Árvore de busca binária permite várias operações de forma dinâmica como: inserção, remoção, busca, percurso. Existem várias variações de árvore, como por exemplo Árvore rubro-negra, Árvore B, entre outras [Ziviani 2010]. Este resumo científico estendido apresenta a árvore de busca binária e algumas de suas operações.

1. Métodos e Ferramentas

Para implementação da solução do problema (ver seção 3) foi utilizada a linguagem C++ em IDE Dev-C++, versão 20120426 (compilador GCC 4.6.1), em um computador com sistema operacional Microsoft Windows 7 Professional na versão 64 bits, processador Intel core i7-3630QM com 4 núcleos de 2,4 GHz, 8GB de memória física (RAM) e 1 TB de disco rígido.

Na solução[1] foram utilizadas estruturas de dados como Pilha para alocação das listas de folhas da ABB, e Pilha e Fila para passagem dos nós em percursos de pré-ordem e largura, respectivamente.

O custo computacional das operações foi analisado utilizando a tabela[2] Big-O Cheat Sheet, de Brandon Amos.

2. Descrição do Problema

O problema a ser solucionado é uma adaptação do problema ***Falling Leaves*** (p1525) sobre Árvore de Busca Binária encontrado na seção Estrutura de Dados do site de competição de programação UVa Online Judge (disponível em `https://uva.onlinejudge.org`). O problema solicita a impressão do percurso em pré-ordem de uma árvore que deve ser construída a partir de um sequência de listas de folhas, correspondentes à poda de todas as folhas da árvore, até que ela seja vazia. Segue abaixo a descrição detalhada do problema:

Contexto: Considere a seguinte sequência de operações em uma árvore de busca binária: a) remova as folhas e liste os dados removidos e b) repita este procedimento até que a árvore esteja vazia. Dada uma árvore qualquer, fazer a remoção de suas folhas e imprimir as folhas removidas, conforme o passo a, e depois dar sequencia ao passo b.

[1] O código, dados e instruções de execução estão disponíveis em `https://github.com/pvcastro/edpa`

[2] A tabela Big-O Cheat Sheet está disponível em `https://bit.ly/1m2zRMh`

O problema é começar com uma sequencia de linhas de folhas de uma árvore de busca binária de números e imprimir a passagem dos percursos por profundidade em pré-ordem da árvore resultante.

Entrada: o arquivo de entrada contém um ou mais conjuntos de dados. Cada conjunto de dados é uma sequência de uma ou mais linhas. As linhas contêm as folhas que foram consecutivamente removidas de uma árvore de busca binária. Conjuntos de dados são separados por uma linha contendo apenas o carácter especial asterisco. O último conjunto de dados é seguido por uma linha contendo apenas o carácter especial dólar. Os números em cada linha são separados por espaços em branco, e não há linhas vazias na entrada.

Saída: para cada conjunto de dados de entrada, existe uma árvore de busca binária única que produziria a sequência de folhas. A saída são linhas contendo os percursos de largura e de profundidade em pré-ordem da árvore.

3. Descrição da Solução e das Estruturas de Dados

A solução é composta por um conjunto de arquivos que contém a implementação da estrutura de dados Árvore de Busca Binária (`binary_tree.cpp`), das estrutura de dados Fila (`simple_queue.cpp`) e Pilha (`simple_stack.cpp`).

No código foram utilizadas as bibliotecas do C++ `fstream`, `vector`, `string` e `ctime` e os arquivos próprios `simple_stack.cpp` e `simple_queue.cpp`. Os métodos desenvolvidos são: o `size_t split()`, que trata a entrada dos dados no vetor; a classe `Node` que constrói a estrutura do Nó da Árvore, o método `add_node()` para inserção do nós, o método `preorder_traversal()` para o percurso de profundidade em pré-ordem e `breadth_traversal()` para o percurso em largura. O método `build_tree()` é o responsável pela construção da árvore a partir do conjunto de listas de folhas, assim como por os métodos de percurso para impressão da árvore resultante.

A seguir apresentamos a implementação da estrutura Árvore de Busca Binária com a classe `Node` e construção da Árvore com o método `add_node`:

O código abaixo da classe `Node` define a estrutura de cada nó da árvore. A estrutura possui o valor do nó, que será utilizado como nosso `value`, assim como dois ponteiros `left` e `right`, representando os filhos à esquerda e à direita do nó.

```
1 class Node {
2 public:
3         Node(int v) {...}
4
5         int value;
6         Node *left;
7         Node *right;
8 };
```

O construtor apenas inicializa o valor do nó a partir do valor `v` especificado, assim como define as referências dos filhos esquerdo e direito para nulo.

```
void add_node(Node *new_node) {}
```

O método `add_node` é responsável pela inserção dos nós da árvore. Este método garante as propriedades da ABB, percorrendo cada nó, desde a raiz, e verificando onde deve ser adicionado o nó. Primeiro, verifica se o valor da raiz é nulo e, em caso afirmativo, a raiz assume o valor do novo nó. Na sequência, percorre todos os nós a partir da raiz, verificando se o valor é menor ou maior do que o do nó percorrido, para determinar se o nó será adicionado em sua sub-árvore esquerda ou direita. Se o elemento do novo nó for igual ao de algum nó da árvore, o método `add_node` informa que já existe o elemento na árvore e finaliza o método.

A operação de inserção tem custo computacional de tempo execução `O(h)`, onde h é a altura da árvore. Com isso, podemos concluir que o tempo de inserção de um elemento na árvore é proporcional à profundidade da mesma [Cormen T. H. 2012].

As estruturas Pilha (`simple_stack.cpp`) e Fila (`simple_queue.cpp`), e dos métodos de percurso em largura e pré-ordem (`breadth_traversal` e `preorder_traversal`) foram utilizadas no projeto de forma didática, para apresentar as diversas alternativas de implementação das estruturas de dados que estudamos.

```
void preorder_traversal(Node *root) {}
```

O método iterativo `preorder_traversal` imprime os nós do percurso de profundidade em pré-ordem. O método começa inicializando uma pilha, adicionando a raiz da árvore na mesma. Em seguida, de forma iterativa, enquanto a pilha não estiver vazia, visita cada nó no topo da pilha, imprimindo seu valor e desempilhando o mesmo. Em seguida, empilha seus filhos direito e esquerdo, nesta ordem. Como o nó esquerdo é sempre empilhado depois do direito, isso garante que os nós da sub-árvore esquerda sejam visitados antes dos nós da sub-árvore direita, conforme determina o percurso em pré-ordem.

```
void breadth_traversal(Node *root) {}
```

O método `breadth_traversal` tem um algoritmo semelhante ao do método `preorder_traversal`, só que ao invés de utilizar uma pilha para armazenar os nós visitados, utiliza uma fila. Desta forma, ao enfileirar cada nó esquerdo e direito do nó visitado, é garantida a impressão dos elementos da árvore no sentido de sua largura, pois a fila vai sendo esvaziada conforme cada nível da mesma é percorrido. Todos os nós na profundidade k são visitados antes de serem visitados os nós de profundidade k + 1.

```
int get_tree_height(Node* node) {}
```

O método `get_tree_height` tem por objetivo calcular a altura da árvore. Ele foi implementado de forma recursiva, incrementando a contagem de cada caminho percorrido no sentido da profundidade da árvore, mantendo a profundidade máxima calculada como correspondente à altura.

```
int main(int argc, char* argv[]) {}
```

O método main é responsável por fazer uma leitura do arquivo de texto que contém os dados de entrada, e percorrer cada linha do arquivo, que é correspondente a uma lista de folhas de uma árvore. Cada lista de folhas é uma string que é adicionada em uma pilha. Ao ler uma linha do arquivo que indique o fim de uma árvore (carácter asterisco ou dólar), é chamado o método `build_tree`, passando a pilha de listas de folhas para construção da árvore.

```
void build_tree(simple_stack<string> &leavesStack) {}
```

O método base da solução proposta é o `build_tree`, que depende de todo o restante da implementação. Neste método, a pilha de listas de folhas é percorrida, e cada string desempilhada é segmentada em uma matriz unidimensional de números inteiros, a partir do método `split`. A matriz de números é percorrida, e cada número é adicionado como um nó da árvore, usando o método `add_node`. Ao percorrer todos os elementos da pilha, a construção da árvore é finalizada, e, ao final, os métodos `pre-order_traversal` e `breadth_traversal` são chamados para imprimir os nós da árvore, de acordo com cada tipo de percurso.

Segue um exemplo de saída do projeto, para uma árvore de 15 elementos:

```
...lendo dados em /home/pedro/repositorios/edpa/exemplo.txt

linha: 1: 930 753 393 350 971 13 175 402
linha: 2: 150 885 793 287
linha: 3: 214 801
linha: 4: 958

Construindo a partir das folhas:

958
214 801
150 885 793 287
930 753 393 350 971 13 175 402

Altura da árvore: 8

Imprimindo os nós da árvore em largura:
958 214 971 150 801 13 175 793 885 287 930 753 393 350 402
Imprimindo os nós da árvore em pré-ordem:
958 214 150 13 175 801 793 287 753 393 350 402 885 930 971

================================================
Tempo de execução dos algoritmos:
================================================
  Duração da construção da árvore: 0.019 ms.
  Duração do cálculo da altura: 0.002 ms.
  Duração do percurso em largura: 0.004 ms.
  Duração do percurso em profundidade: 0.004 ms.
================================================
```

4. Casos de Teste

Na etapa de análise e desenho da solução foram desenvolvidos casos de teste para estudo das saídas esperadas, bem como para análise das respostas necessárias para

verificação da execução dos algoritmos. Segue abaixo um conjunto de entradas de teste e um conjunto de saída esperado:

Entrada esperada:

```
930 753 393 350 971 13 175 402
150 885 793 287
214 801
958
$
```

Saída esperada:

```
Imprimindo o percurso em pré-ordem:
958 214 150 13 175 801 793 287 753 393 350 402 885 930 971

Imprimindo o percurso em largura:
958 214 971 150 801 13 175 793 885 287 930 753 393 350 402
```

Foram criados casos de testes para tratar de diferentes quantidades de elementos a serem adicionados como nós de árvores binárias de busca. Para que pudessem ser avaliadas quantidades exponenciais de elementos, optamos por usar 4 árvores diferentes, com 31, 63, 127 e 255 elementos.

Para geração dos números correspondentes aos elementos de cada árvore, foi usado um script de geração aleatória de números inteiros, em linguagem python[3]. No código abaixo foi utilizado o método `sample` da biblioteca `random` com a especificação da quantidade de números inteiros a serem gerados, e o intervalo dos mesmos. Para cada tamanho desejado de árvore, geramos uma quantidade de elementos correspondente, com valores entre 1 e 1000. Com isso, foram geradas 4 amostras diferentes de árvores distintas, contendo, respectivamente, 5, 6, 7 e 8 linhas de folhas. As alturas calculadas pela solução, para cada uma das árvores, foram de 10, 11, 14 e 15. A tabela 1 contém as informações acerca dos tempos de cada operação de cada árvore.

```
for n_elementos in [31, 63, 127, 255]:
    print(random.sample(range(1, 1000), n_elementos))
```

Exemplo do conjunto de entrada para a árvore com 31 elementos, contendo 5 linhas de folhas[4]:

```
957 404 349 441 486 587 340 255 770 732 466 794 522
...continua:  784 365 986
152 55 529 236 344 242 978 211
655 68 351 945
485 573
659
*
```

[3] `https://www.python.org`

[4] Todos os 4 conjuntos de teste podem ser vistos em `https://github.com/pvcastro/edpa/blob/master/input.txt`

Tabela 1. Tempos de execução para cada árvore avaliada. A tabela apresenta os tempos de execução para cada árvore avaliada. As colunas de Construção, Cálculo da Altura, Largura e Profundidade possuem o tempo de execução de cada uma destas operações, em milissegundos. As colunas Largura e Profundidade indicam as operações de percurso destes sentidos.

Qtd. Nós	Altura	Construção	Cálculo da Altura	Largura	Profundidade
31	10	0.022	0.002	0.004	0.005
63	11	0.029	0.003	0.01	0.009
127	14	0.045	0.004	0.016	0.016
255	15	0.07	0.005	0.028	0.026

5. Complexidade de Tempo e de Espaço

A análise geral da complexidade pode ser feita através dos resultados do Tempo de Execução ou do Uso de Espaço de Memória de um algoritmo[Cormen T. H. 2012]. Seguindo a literatura, analisou-se a complexidade em tempo e em espaço de memória no Pior, Médio e no Melhor caso para as operações do Projeto [Szwarcfiter J.L. 2010]. Para a análise da solução proposta foi utilizada a tabela a Big-O Cheat Sheet, de Brandon Amos[5].

Assintoticamente, a complexidade de espaço da Árvore Binária de Busca é da ordem de O(n). A operação de inserção implementada na solução tem como pior caso O(n) e O(log n) no caso médio. As implementações de Pilha e Fila possuem complexidade O(1) para as operações de inserção e remoção utilizadas nos percursos, assim como para manutenção dos dados de entrada. Como os percursos são operações que passam por todos os nós da árvore, o a complexidade do tempo de execução dos mesmos é de O(n).

Usamos 4 conjuntos de entrada contendo 31, 63, 127 e 255 elementos, contendo respectivamente cada uma 5, 6, 7 e 8 linhas de folhas. Caso estas árvores fossem Árvores Binárias Perfeitas, teriam a altura, respectivamente, 5, 6, 7 e 8. Como usamos números gerados aleatoriamente para a construção das árvores, a disposição aleatória dos mesmos resultou em árvores de alturas 10, 11, 14 e 15. Assim, pudemos ver que a variação do tempo das operações avaliadas não resultou em crescimento exponencial. O tempo de construção das árvores de 31 e 255 nós variou 3.18 vezes, enquanto o crescimento do número de nós foi de 8.22 vezes. O tempo de cálculo da altura cresceu somente 2.5 vezes, enquanto o tempo de percurso e de profundidade foram os que mais aumentaram, em 7 e 5.2 vezes respectivamente. É razoável que estas últimas duas operações tenham aumentado mais do que as primeiras, pois dependem linearmente da quantidade de elementos da árvore.

6. Conclusão

Neste projeto implementamos uma árvore binária simples, não balanceada e as operações básicas de percursos a fim de mostrar as propriedades de Árvore de Busca Binária e discutir sua análise assintótica geral.

[5]tabela disponível em `https://bit.ly/1m2zRMh`

Vimos então o tipo abstrato de dados Árvore de Busca Binária. Essa estrutura possui, para a maioria de suas operações, complexidade de tempo O(log n) para o caso médio e O(n) para o pior caso. A complexidade de espaço encontrada é O(n).

Em comparação com outras estruturas, caso tivéssemos implementado balanceamento AVL ou rubro-negro para garantir uma complexidade O(log n) para as operações, teríamos percebido que o aumento de tempo das operações teria sido menor ainda.

Referências

Cormen T. H., Leiserson C. E., R. R. L. S. C. (2012). *Algoritmos Teoria e Pratica*, volume 3a Edicao.

Sedgewick R., W. K. (2011). *Algorithms*, volume 4a Edicao.

Szwarcfiter J.L., M. L. (2010). *Estruturas de Dados e seus Algoritmos*, volume 2a Edicao.

Ziviani, N. (2010). *Projeto de Algoritmos com Implementacoes em Pascal e C*.