Estruturas de Dados e Projetos de Algoritmos
Tutores (URI 2120)

Fabrício S. Paranhos[1] **e Leandro A. Vianna**[1]

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

1. Descrição do Problema (Tutores - URI 2120)

O problema Tutores[1] consiste em dado uma ordem de inserções de chaves em uma Árvore Binária de Busca (ABB), deve ser respondido para várias chaves qual a chave do nó pai do nó em que essa chave está.

Formalmente, dado uma ordem de inserção de chaves $O = \{X_1, X_2, \ldots, X_n\}$ e uma ABB T em que foram inseridas as chaves O. Considere $p(X)$ como a chave do nó pai do nó em que a chave X está em T. Para uma sequência $I = \{i_1, i_2, \ldots, i_m\}$, a solução deve apresentar uma sequência de resposta $Y = \{p(X_{i_1}), p(X_{i_2}), \ldots, p(X_{i_m})\}$.

Por exemplo, suponhamos a ordem $\hat{O} = \{2, 4, 5, 6, 1\}$ e a sequência $\hat{I} = \{2, 3, 4\}$. A sequência de resposta é $Y = \{p(X_2 = 4) = 2, p(X_3 = 5) = 4, p(X_4 = 6) = 5\}$. Os valores de p podem ser visualizados na árvore binária de busca T desse exemplo que está na figura 1.

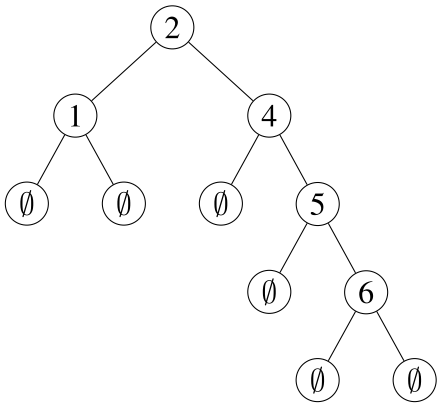

Figura 1. Árvore Binária de Busca T gerada a partir da ordem de inserção de chaves $\hat{O}$

As restrições do problema são:

- Dado ordem de chaves $O = \{X_1, X_2, \ldots, X_n\}$, $1 \leq X_i \leq 10^9$ e $2 \leq n \leq 100000$.
- Seja Q o número de perguntas, $1 \leq Q \leq 99999$.

O tempo limite de execução do problema é de 1 segundo. Logo, dado que o tamanho do problema (número de chaves) n pode chegar a 10^5 e considerando que a cada segundo são executadas aproximadamente 10^8 operações, a solução deve ser da ordem de $O(n \ln n)$ para executar no tempo exigido pelo juiz *online* URI.

[1]`https://www.urionlinejudge.com.br/judge/en/problems/view/2120`

2. Solução Proposta

2.1. Algoritmo

Pela descrição do problema, para definir os tutores de cada aluno o sistema utiliza uma Árvore Binária de Busca (ABB), em que o pai do aluno na árvore é seu tutor. Entretanto, o custo da inserção seria proibitivo se utilizássemos a mesma estrutura para encontrar os tutores, que no pior caso, seria uma árvore binária degenerada com complexidade de inserção e busca $O(n)$.

Observa-se que a ABB define uma sequência de matrículas relacionada a um determinado percurso e a estrutura da árvore. Nossa solução (algoritmo 1 e 2) propõe modelar a mesma sequência, entretanto utilizando uma árvore binária balanceada auxiliar para calcular os pais e altura dos alunos em relação a ABB original.

Como a árvore binária balanceada auxiliar foi escolhido utilizar a *Splay Tree* [Sleator and Tarjan 1985], que não tem garantia de uma árvore com altura $O(\lg n)$, mas tem a complexidade amortizada de $O(m \lg n)$ para uma sequência de m operações. Para guardar os pais e alturas dos alunos na ABB foi escolhido a *Hash Table* [Cormen et al. 2009] com tratamento de colisão por encadeamento.

Algoritmo 1 Calcula o pai (tutor) de cada matrícula, modelando a sequência *inorder* de uma Árvore Binária de Busca (ABB) utilizando uma árvore binária balanceada.

Require: $mat[N]$ vetor de matrículas de tamanho N; $parent$ tabela *hash*.

1: **procedure** PARENTS($mat[N], parent$)
2:   $order \leftarrow$ SPLAYTREE( )
3:   $level \leftarrow$ HASHTABLE(N)

4:   **for** $i \leftarrow 0, N-1$ **do**
5:    **if** $order \neq \emptyset$ **then**
6:     PARENTSINNER(mat, $parent$, $order$, $level$)
7:    **end if** $\triangleright$ $order \neq \emptyset$

8:    INSERT($order, mat[i]$)
9:   **end for**
10: **end procedure**

O algoritmo recebe uma lista de matrículas (mat) e uma tabela ($parent$) associando alunos e tutores vazia. Inicialmente definimos uma *Splay Tree* ($order$), aonde serão feitas as inserções das matrículas, e outra tabela ($level$) associando a matrícula à altura na árvore original, linhas 2 e 3. Após a inicialização, percorremos todos os alunos (linha 4), simulando o percurso original (procedimento PARENTSINNER) e finalmente inserimos na árvore balanceada (linha 29).

Dentro do procedimento PARENTSINNER queremos encontrar qual posição da sequência a matrícula i ($mat[i]$) encaixará, para isso procuramos (linha 2) pela menor matrícula ($upper$) tal que seja maior ou igual à matrícula i ($mat[i]$) na árvore $order$. Caso a matrícula ($upper$) não exista, podemos concluir que nossa sequência terá o seguinte formato após inserção: $(\ldots, w, mat[i])$, ou seja $mat[i]$ é maior que todas as matrículas na ABB e filho de w. Assim obtemos o maior elemento da sequência (w linha 4) e a altura

Algoritmo 2 Procedimento interno de PARENTS.

Require: $mat[N]$ vetor de matrículas de tamanho N; $parent$ tabela *hash*.

1: **procedure** PARENTSINNER(mat, $parent$, $order$, $level$)
2: $\quad upper \leftarrow$ LOWERBOUND($order, mat[i]$)

3: $\quad$ **if** $upper$ not found **then**
4: $\quad\quad w \leftarrow$ MAX($order$) ▷ Maior elemento de $order$.
5: $\quad\quad l \leftarrow$ LOOKUP($level, w + 1$) ▷ $l = level[w + 1]$
6: $\quad\quad$ INSERT($parent, mat[i], w$) ▷ $parent[mat[i]] = w$
7: $\quad\quad$ INSERT($level, mat[i], l$)

8: $\quad$ **else if** $upper =$ MIN($order$) **then** ▷ Menor elemento de $order$
9: $\quad\quad l \leftarrow$ LOOKUP($level, upper + 1$)
10: $\quad\quad$ INSERT($parent, mat[i], upper$)
11: $\quad\quad$ INSERT($level, mat[i], l$)

12: $\quad$ **else**
13: $\quad\quad lower \leftarrow$ PREVIOUS($order, upper$) ▷ Elemento anterior em $order$

14: $\quad\quad$ **if** LOOKUP($level, upper$) $>$ LOOKUP($level, lower$) **then**
15: $\quad\quad\quad l \leftarrow$ LOOKUP($level, upper$)
16: $\quad\quad\quad$ INSERT($parent, mat[i], upper$)
17: $\quad\quad\quad$ INSERT($level, mat[i], l + 1$)

18: $\quad\quad$ **else**
19: $\quad\quad\quad l \leftarrow$ LOOKUP($level, lower$)
20: $\quad\quad\quad$ INSERT($parent, mat[i], lower$)
21: $\quad\quad\quad$ INSERT($level, mat[i], l + 1$)
22: $\quad\quad$ **end if**
23: $\quad$ **end if** ▷ $upper$ not found
24: **end procedure**

(l) na qual $mat[i]$ seria inserido na árvore (linha 5), finalmente atualizamos as tabelas de pais e alturas com os novos dados de $mat[i]$ (linhas 6–7). Caso a matrícula exista ($upper$) ela pode ser o primeiro elemento da sequência (linhas 8–11), isto é a menor matrícula, ou estar em qualquer posição no meio da sequência (linhas 12–23). Se $upper$ for o menor elemento da árvore $order$, a sequência terá formato: $(mat[i], w, \ldots)$, ou seja, $mat[i]$ é a menor matrícula na árvore original com pai $upper$. Analogamente, obtemos a altura (l) na qual $mat[i]$ seria inserido e atualizamos a tabela de pais e alturas (linhas 10–11). Para o último caso teríamos sequência de formato: $(\ldots, lower, mat[i], upper, \ldots)$, em que $lower$ é a matrícula anterior a $upper$ em $order$ (linha 13).

Nesse passo temos duas configurações possíveis, como pode ser visto na figura 2, portanto precisamos determinar quem está na subárvore de quem (linha 14) para escolher a configuração correta: se $upper$ está na subárvore de $lower$ (caso 1) então $upper$ é pai de $mat[i]$ (linhas 15–17), caso contrário (caso 2) $lower$ é pai de $mat[i]$ (linhas 19–21). Ao final do algoritmo a tabela $parent$ possui os tutores de cada aluno.

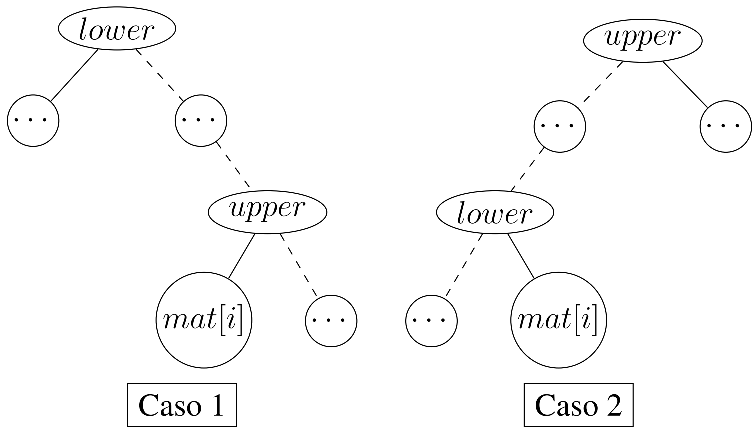

Figura 2. Se $mat[i]$ será inserido entre duas matrículas na sequência, existem dois casos para a estrutura da ABB.

2.1.1. Análise da complexidade

As operações INSERT($parent$) e INSERT($level$), inserção em tabela *hash*, possuem complexidade $O(1)$. Como discutido na seção 2.1 a operação MAX, MIN, INSERT($order$), LOWERBOUND e PREVIOUS, utilizando análise amortizada, possui complexidade $O(m \lg n)$ para m operações, em que n é a quantidade de elementos na árvore. Analogamente, por análise amortizada, a operação LOOKUP, de uma tabela *hash* com tratamento de colisão por encadeamento, possui complexidade $O(1)$. Com isso, os trechos 3–7, 8–11 e 12–21 do algoritmo 2 possuem complexidade $O(\lg n)$, então o bloco entre 3–24 tem complexidade $O(\lg n)$. Concluímos que a complexidade do algoritmo PARENTS é, pela aproximação de Stirling:

$$\sum_{i=1}^{N}(\mathrm{O}(\lg i) + \mathrm{O}(\lg i) + \mathrm{O}(\lg i)) = \sum_{i=1}^{N}(\mathrm{O}(\lg i)) = \mathrm{O}(\lg \prod_{i=1}^{N} i) = \mathrm{O}(\lg N!) \sim N \lg N$$

3. Resultados

3.1. Casos de Teste

Nessa seção demostraremos alguns exemplos de casos de teste do problema Tutores e qual o comportamento da *Splay Tree* ao realizar a inserção das matrículas.

3.1.1. Exemplo de Caso de Teste 1

Listing 1. Instância de teste 1

```
3        // Número de estudantes
5 1 2    // Ordem de inserção dos estudantes
1        // Quantas consultas serão realizadas
2        // Tutor do estudante 2
```

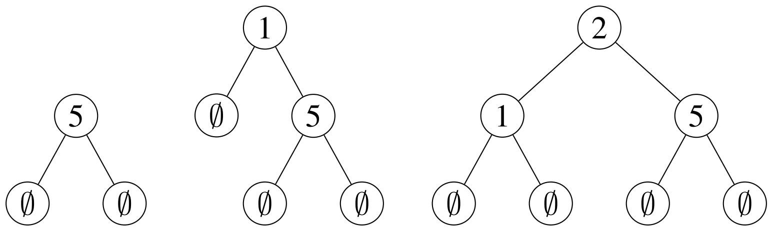

Figura 3. ***Splay Tree*** **após cada inserção de chave do Caso de Teste 1**

3.1.2. Exemplo de Caso de Teste 2

Listing 2. Instância de teste 1

```
5            // Número de estudantes
3 1 4 2 5    // Ordem de inserção dos estudantes
2            // Quantas consultas serão realizadas
2 5          // Tutores dos estudantes 2 e 5
```

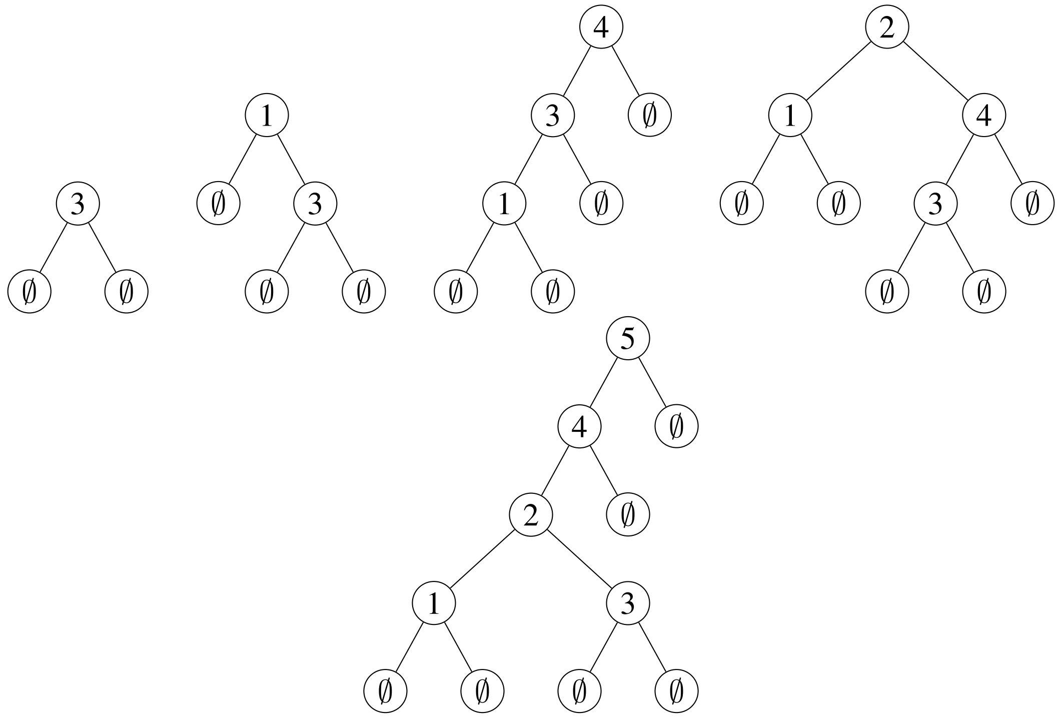

Figura 4. ***Splay Tree*** **após cada inserção de chave do Caso de Teste 2**

3.1.3. *Benchmarks*

Com o objetivo de verificar o tempo de execução, testamos nossa solução com casos de diferentes de tamanhos. São dois tipos de casos em tamanhos de 10 à 10^5 com as chaves gerados uniformemente aleatórias:

- Matrículas em qualquer ordem (não ordenadas).
- Matrículas **ordenadas**.

Os tempos de execução estão descrito no gráfico da figura 5.

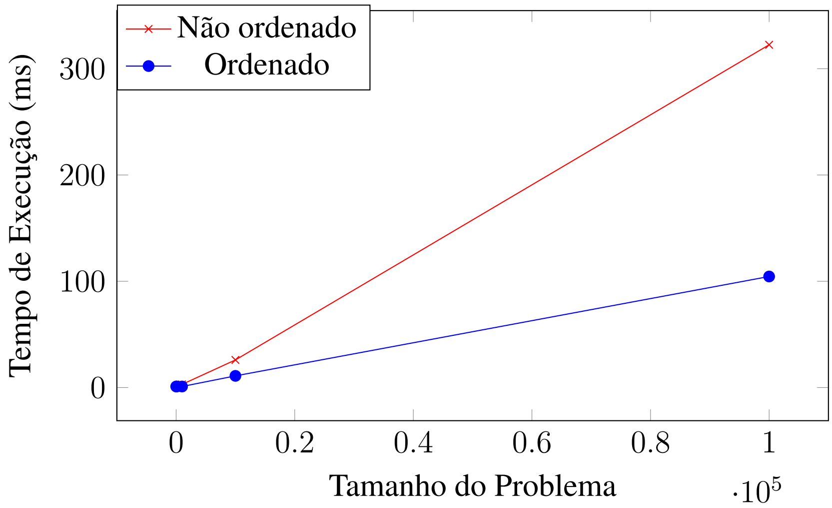

Figura 5. Tempo de execução em milissegundos para os testes gerados.

Se observa um tempo melhor para os casos em que as chaves são adicionadas em **ordem crescente**, o que pode ser explicado pois a cada inserção o último nó inserido vai estar na raiz da *Splay Tree* (por característica da estrutura de dados) e como esse nó vai ser o pai do nó que está sendo inserido nesse momento, isso acelera a busca por ele.

4. Conclusões

Obtivemos uma solução capaz de realizar consultas aos tutores do colégio de complexidade O($n \lg n$), superando o método anterior utilizando uma ABB da ordem de O(n^2). Assim não há mais a necessidade de armazenar a ABB para realizar consultas, contudo devemos armazenar uma *Splay Tree* e duas *Hash Tables*, o que acarreta um uso memória maior mas ainda da ordem de $O(n)$.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Sleator, D. D. and Tarjan, R. E. (1985). Self-adjusting binary search trees. *J. ACM*, 32(3):652–686.