Técnicas de Busca e Ordenação
Roteiro de Laboratório – Árvores Balanceadas

1 Objetivo

O objetivo deste laboratório é estudar, implementar e avaliar alguns dos diferentes métodos propostos para se manter uma **árvore binária de busca** (*binary search tree – BST*) razoavelmente balanceada. Essa propriedade é absolutamente essencial para as BSTs pois o desempenho de todas as suas operações (busca, inserção, etc) é totalmente dependente da altura da árvore.

2 BSTs com entrada aleatória

Uma BST com implementação padrão, isto é, que não cuida de balanceamento, é totalmente suscetível à ordenação da entrada (sequência de inclusão dos elementos na árvore). Como já visto anteriormente, o pior caso possível para a BST padrão ocorre quando a sequência de inserções é uma sequência ordenada (crescente ou decrescente) de chaves. Nesse caso, a árvore fica degenerada, com altura N, onde N é o número de elementos inseridos (assumindo que não há repetição de chaves).

Para combater esse problema de forma simples, basta embaralhar a sequência de entrada, de forma similar ao que foi feito para o algoritmo de *quick sort*, garantindo assim um limite assintótico probabilístico de $N \lg N$. A maior desvantagem desse método, além do trabalho adicional de embaralhar a entrada, é a necessidade de termos todas as chaves que devem ser inseridas já no início. (Muitas aplicações de BSTs alternam as operações de inserção e consulta – também deleção – e nem sempre todas as chaves são conhecidas já no início.)

Uma BST totalmente balanceada possui altura $h = \lfloor \lg N \rfloor$, o limite inferior de todos os métodos de balanceamento de BSTs. Uma BST padrão construída com uma entrada aleatória possui uma altura esperada da ordem de $h \sim 4.311 \ln N$, conforme resultado obtido por Reed em 2003.

Para o experimento realizado nessa seção, não é estritamente necessário ler uma sequência de chaves de entrada e a seguir embaralhá-la. Uma forma ainda mais simples, conforme visto no Laboratório de Árvores e Recursão (2), é simplesmente gerar N números aleatórios e os inserir na BST na ordem que eles forem gerados. Isso garante as mesmas propriedades da BST descritas no parágrafo anterior.

Exercício 0. Implemente (ou recupere do Laboratório de Árvores e Recursão) uma BST padrão conforme descrição abaixo:

- O nó da BST é somente uma chave (um número inteiro positivo), um valor (um número inteiro positivo) e um ponteiro para os filhos da esquerda e direita.
- Implemente funções que criam e destroem uma BST.
- Implemente uma função que calcula e retorna a altura de uma BST.
- Implemente uma função que insere uma chave dada em uma BST.

- Implemente um programa cliente que gera $N = 10^6$ chaves aleatórias e as insere na BST. A seguir, o programa calcula a altura da árvore e exibe na tela. Repita esses passos 10 vezes, calculando a altura média da BST.
- Avalie a altura média obtida pelo seu cliente segundo os limites inferior e superior descritos acima.

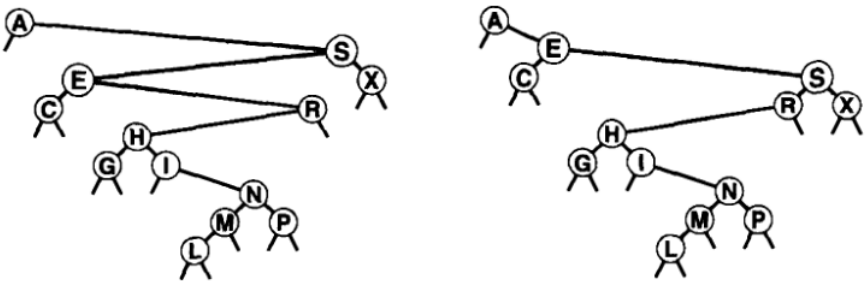
Obs.: Toda as funções que operam sobre a BST podem ser implementadas de forma recursiva ou iterativa, como você preferir. Entretanto, as versões recursivas tendem a ser mais simples.

3 Inserção na raiz em BSTs

Na implementação da BST padrão, todo nó novo é inserido no final da árvore. No entanto, isso não é essencial mas somente uma consequência do algoritmo recursivo de inserção. Nessa seção, vamos considerar um método de inserção alternativo, onde qualquer nova chave sempre é inserida na raiz da BST.

Para fazer a inserção na raiz funcionar, são necessárias duas operações básicas: *rotação à direita* e *rotação à esquerda*. Essas operações são transformações fundamentais em BSTs e essencialmente permitem trocar o papel da raiz e de um dos filhos da raiz, *preservando a ordenação da BST*.

Uma *rotação à direita* envolve a raiz e o seu filho da *esquerda*. A figura abaixo mostra um exemplo de uma BST antes e depois da rotação à direita sobre o nó S:

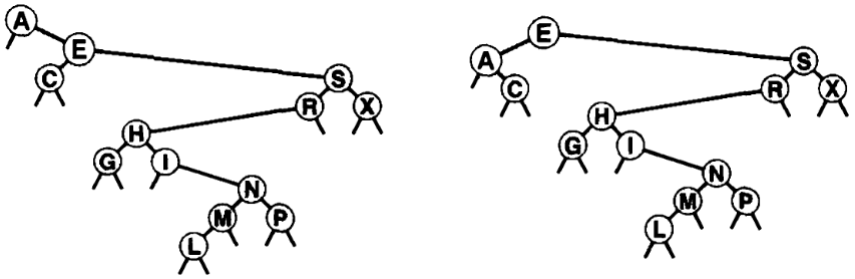


Essa operação pode ser implementada de forma bastante simples, como mostra a função abaixo:

```
BST* rotate_right(BST *n) {
    BST *t = n->l;
    n->l = t->r;
    t->r = n;
    return t;
}
```

Como pode ser visto no código acima, uma rotação é uma mudança local na árvore, envolvendo somente dois nós e três ponteiros.

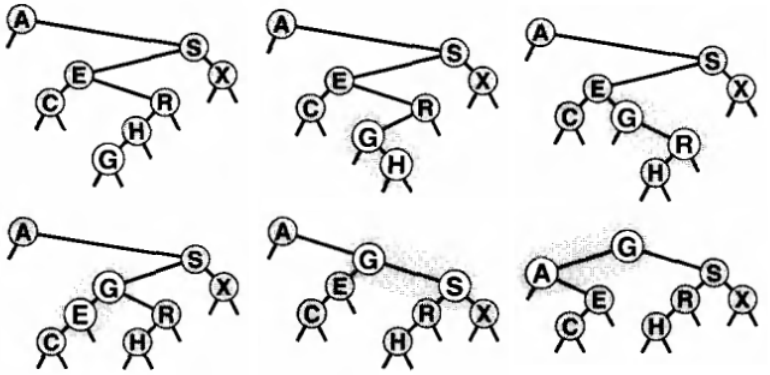
Uma *rotação à esquerda* envolve a raiz e o o seu filho da *direita*. A figura abaixo mostra um exemplo de uma BST antes e depois da rotação à esquerda sobre o nó A:



As operações de rotação nos levam a uma implementação recursiva para inserção na raiz: insira a nova chave na subárvore apropriada para manter a ordenação da BST e a seguir realize uma rotação apropriada para trazer a nova chave para a raiz. O algoritmo recursivo de inserção pode ser descrito como abaixo.

```
Entrada: nó n da BST, chave k a ser inserida.
Caso base: n é nulo. Retorne um novo nó com a chave k.
Passo recursivo:
    Se k é menor que a chave do nó:
        Insira recursivamente a chave na subárvore da esquerda.
        Realize uma rotação à direita, atualizando o ponteiro n.
    Se k é maior que a chave do nó:
        Insira recursivamente a chave na subárvore da direita.
        Realize uma rotação à esquerda, atualizando o ponteiro n.
    Retorne n.
```

A figura abaixo mostra o resultado de inserir G na BST superior esquerda, com rotações recursivas após a inserção para trazer o novo nó G para a raiz. Esse processo é equivalente a inserir G nas folhas e a seguir realizar uma sequência de rotações para trazer o nó para a raiz.



Exercício 1. Modifique a sua implementação do Exercício 0 conforme descrição abaixo:

- Implemente e teste as operações de rotação à direita e à esquerda.
- Implemente a inserção na raiz segundo o algoritmo descrito acima.
- Utilize o mesmo programa cliente do Exercício 0 para inserir (na raiz) as $N = 10^6$ chaves aleatórias e calcular a altura média da BST após 10 repetições.
- Compare a altura média obtida nesse exercício com a altura média obtida no Exercício 0.

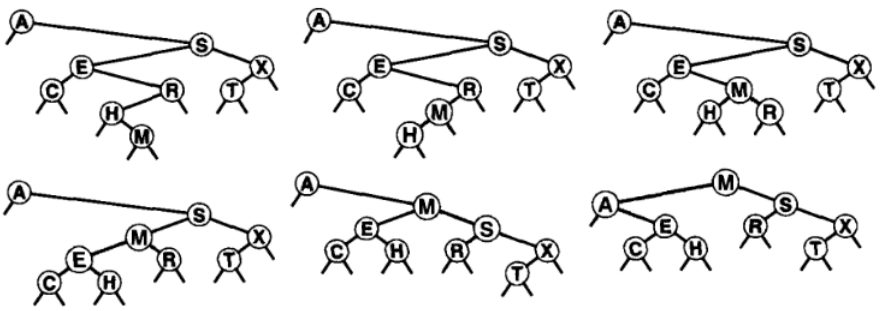
4 Particionamento e balanceamento de BSTs

4.1 Particionamento

Uma operação de *particionamento* da BST com parâmetro k rearruma a árvore de forma a colocar o k-ésimo menor elemento na raiz. Essa operação pode ser implementada de forma muito similar à definição recursiva de inserção na raiz vista na seção anterior: se colocarmos (recursivamente) o nó desejado na raiz de uma das subárvores, podemos então tornar esse nó a raiz geral com uma única rotação. O algoritmo abaixo ilustra esse método.

```
Entrada: nó n da BST, ordinal k da chave que deve ir para a raiz.
Seja t o tamanho da subárvore da esquerda de n.
Se t > k:
    Particione recursivamente a subárvore da esquerda de n com parâmetro k.
    Realize uma rotação à direita, atualizando o ponteiro n.
Se t < k:
    Particione recursivamente a subárvore da direita de n com parâmetro k-t-1.
    Realize uma rotação à esquerda, atualizando o ponteiro n.
Retorne n.
```

A figura abaixo ilustra o particionamento da BST superior esquerda pela chave *mediana* M, usando rotações (recursivas) da mesma forma que na inserção na raiz.



Um ponto fundamental do algoritmo de particionamento apresentado acima é a necessidade de se saber o *tamanho* de uma subárvore começando em um determinado nó. Isso pode ser feito incluindo um novo campo `size` em cada nó da BST e atualizando esse campo a cada operação que modifique uma subárvore.

4.2 Balanceamento

Em um cenário ideal, nós gostaríamos que todas as BSTs fossem perfeitamente balanceadas, de forma a garantir um desempenho da ordem de $\lg N$ para qualquer busca na árvore. Até agora, nós nos baseamos na *aleatoriedade da entrada* para termos uma garantia probabilística de que as BSTs não são muito desequilibradas.

Um outro método para se produzir BSTs melhores é rebalanceá-las periodicamente de forma explícita. É possível, por exemplo, balancear uma BST em tempo linear, usando um algoritmo recursivo como abaixo.

```
Entrada: nó n da BST.
Se tamanho da árvore em n é < 2: retorne n.
```

```
Execute o particionamento no nó n, com k = (tamanho de n) / 2.
Balanceie recursivamente a subárvore da esquerda de n.
Balanceie recursivamente a subárvore da direita de n.
Retorne n.
```

Embora um balanceamento como acima possa ser muito custoso em alguns casos, em outros ele vale a pena, por tornar a árvore totalmente balanceada. Um exemplo de tal uso é quando sabemos que vai ser necessário realizar uma grande quantidade de buscas na BST. Assim, o custo do balanceamento fica *amortizado* entre as buscas.

Exercício 2. Modifique a sua implementação do Exercício 1 conforme descrição abaixo:

- Modifique a implementação do nó da BST para incluir um campo de tamanho.
- Modifique a implementação das rotações e da função de inserção para ajustar corretamente o tamanho das subárvores a cada operação. *Obs.: Cuidado a tentar acessar o campo de tamanho de nós nulos. A forma mais segura é implementar uma função de tamanho que cuida desse caso.*
- Implemente uma função de particionamento conforme algoritmo descrito na seção 4.1.
- Implemente uma função de balanceamento conforme algoritmo descrito na seção 4.2.
- Utilize o mesmo programa cliente do Exercício 1 para inserir (na raiz) as $N = 10^6$ chaves aleatórias. A seguir balanceie a BST. Meça a altura antes e depois do balanceamento.
- Analise a altura da árvore balanceada para se certificar que a sua implementação está correta.

5 BSTs aleatórias

Até agora, para manter as BSTs razoavelmente balanceadas nós assumimos que as chaves são inseridas em uma ordem aleatória. A principal consequência dessa suposição é que cada nó da árvore tem a mesma probabilidade de estar na raiz, e essa propriedade também vale (recursivamente) para as subárvores. Vamos agora introduzir aleatoriedade diretamente no algoritmo de inserção, *sem fazer nenhuma suposição sobre a ordem em que os itens são inseridos.*

A ideia é simples: para inserir um novo nó em uma árvore com N nós, o novo nó deve aparecer na raiz com probabilidade $1/(N+1)$. Assim, tomamos uma decisão randomizada de usar inserção na raiz com essa probabilidade. Caso contrário, usamos a inserção randomizada recursivamente na subárvore apropriada. Essas ideias estão resumidas no algoritmo abaixo:

```
Entrada: nó n da BST, chave k a ser inserida.
Caso base: n é nulo. Retorne um novo nó com a chave k.
Passo recursivo:
    Com probabilidade = 1 / ((tamanho de n) + 1),
        retorne o resultado da inserção na raiz em n.
    Se k é menor que a chave do nó:
        Execute recursivamente inserção randomizada na subárvore da esquerda.
    Se k é maior que a chave do nó:
        Execute recursivamente inserção randomizada na subárvore da direita.
    Ajuste o tamanho de n.
    Retorne n.
```

Exercício 3. Modifique a sua implementação do Exercício 2 conforme descrição abaixo:

- Implemente o algoritmo de inserção aleatória descrito acima.
- Utilize o mesmo programa cliente do Exercício 2 para inserir (na raiz) as $N = 10^6$ chaves aleatórias e calcule a altura média das BSTs após 10 execuções.
- Modifique o cliente para inserir a sequência ordenada de chaves $0, \ldots, N-1$. Calcule novamente a altura média das BSTs após 10 execuções.
- Compare as alturas encontradas em cada um dos clientes.