Fundação CECIERJ - Vice Presidência de Educação Superior a Distância
Curso de Tecnologia em Sistemas de Computação
Disciplina: Programação com Interfaces Gráficas
AD1 1^o semestre de 2019.
Professores: Mario Benevides e Paulo Roma

AD1: Árvores Balanceadas-Ponderadas Amortizadas

1 Objetivos

Neste projeto, você:

- ficará mais familiarizado com árvores binárias de busca [1] e
- praticará a implementação de estruturas de dados encadeadas.

2 Introdução

A complexidade de tempo das operações básicas em uma árvore de busca binária — *contains()*, *add()*, e *remove()* — são proporcionais a altura da árvore. No caso ideal, a altura de uma árvore com n elementos é no máximo $log_2\ n$. Se nenhuma precaução for tomada, no entanto, a altura pode ser $n - 1$.

Uma maneira de garantir que a altura de uma árvore seja $O(log_2\ n)$ é mantê-la *com balanceamento ponderado* [2] [3]. Informalmente, isso significa que para cada nó x na árvore, garantimos que os tamanhos (números de elementos) das subárvores esquerda e direita de x não diferem muito. Formalmente, seja T uma árvore de busca binária e seja α uma constante, tal que $\frac{1}{2} < \alpha < 1$. Seja x qualquer nó em T e seja **tamanho**, o número de

[1] `http://knuth.luther.edu/~leekent/CS2Plus/chap6/chap6.html`
[2] `http://cglab.ca/~morin/teaching/5408/refs/gr93.pdf`
[3] `https://en.wikipedia.org/wiki/Scapegoat_tree`

elementos na subárvore com raiz em x. Dizemos que x é α-***balanceado*** se

$$\text{(número de elementos na subárvore esquerda de x)} \leq \alpha \cdot \text{tamanho}, \quad (1)$$

e

$$\text{(número de elementos na subárvore direita de x)} \leq \alpha \cdot \text{tamanho}. \quad (2)$$

Dizemos que a árvore T é α-***balanceada*** se todo nó for α-balanceado. Uma árvore α-balanceada está mostrada na Figura 1.

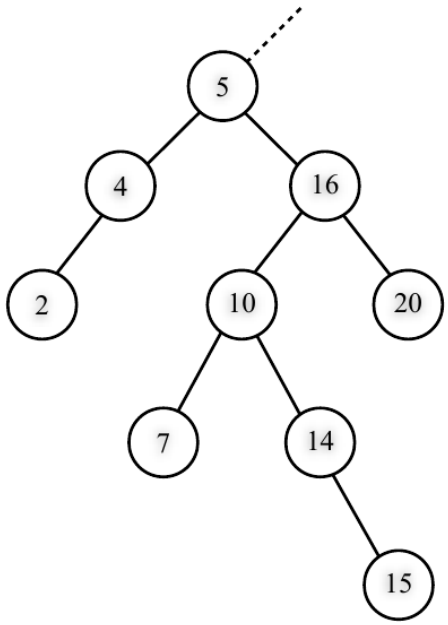

Figura 1: Uma árvore α-balanceada (que é uma subárvore de uma árvore maior) com $\alpha = 2/3$. Por exemplo, considere o nó contendo 5. O número total de nós na subárvore é 9. A subárvore esquerda possui 2 nós, então temos $2/9 \leq 2/3$. A subárvore da direita possui 6 nós, então temos $6/9 \leq 2/3$.

Note que a altura de uma árvore α-balanceada com n nós é no máximo $log_{1/\alpha}\ n$. Como α é uma constante, isto significa que *contains()*, *add()*, e *remove()*, gastam tempo logarítmico. No entanto, adicionar ou remover

elementos pode levar a árvores que não mais satisfazem as condições de balanceamento (1) e (2). Neste projeto, você implementará um certo tipo de árvore de busca binária balanceada por peso, que mantém o α-balanceamento através de uma estratégia de rebalanceamento descrita na próxima seção.

Nota. Uma abordagem alternativa para obter altura logarítmica é usar árvores com altura balanceada. Nestas árvores, as alturas das subárvores esquerda e direita de qualquer nó não diferem muito umas das outras. Por exemplo, em uma árvore AVL [4], as alturas das árvores esquerda e direita em qualquer nó diferem no máximo de uma unidade. Uma árvore *red-black* [5] é outro tipo de árvore balanceada em altura bastante popular, com um condição de balanceamento mais complexa. Árvores rubro-negras são usadas na implementação das classes TreeSet [6] e TreeMap [7] do Java. Árvores AVL e *red-black* estão descritas na Wikipedia, onde é possível encontrar links para conseguir informação adicional. Árvores balanceadas em altura não serão mais discutidas aqui.

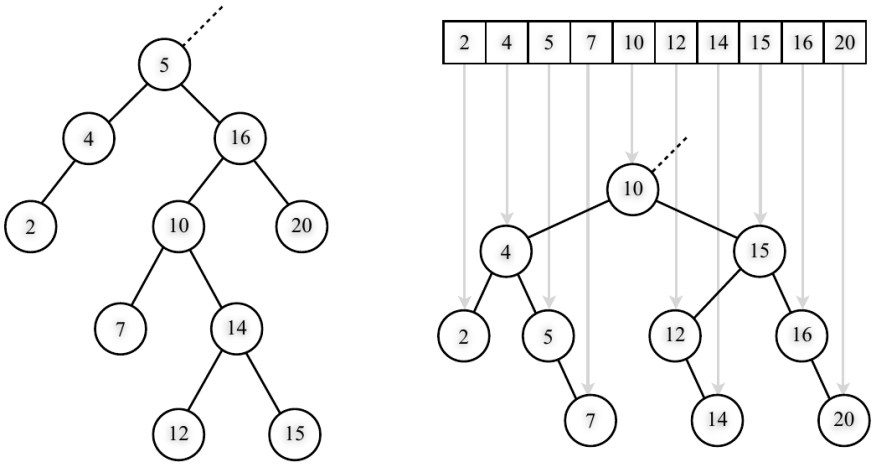

Figura 2: Decomposição Recursiva.

[4] `https://en.wikipedia.org/wiki/AVL_tree`
[5] `https://en.wikipedia.org/wiki/Red-black_tree`
[6] `https://www.geeksforgeeks.org/set-in-java/`
[7] `https://www.callicoder.com/java-treemap/`

3 Árvores α-Balanceadas Amortizadas

As árvores com balanceamento ponderado a serem implementadas mantêm o balanceamento por meio de reestruturações periódicas de subárvores inteiras, reconstruindo-as para que elas se tornem $\frac{1}{2}$-balanceadas. O trabalho necessário para rebalancear é $O(n)$ no pior dos casos, mas pode ser mostrado que o tempo amortizado para uma adição ou remoção é $O(log\ n)$. Embora uma prova formal disso esteja além do escopo deste curso, é intuitivo que o rebalanceamento é relativamente raro.

Em seguida, será explicado o método de rebalanceamento do qual dependem, as árvores α-balanceadas amortizadas. A seguir, vamos explicar como o rebalanceamento é feito após uma atualização.

3.1 A Operação de Rebalanceamento

Suponha que x seja algum nó de uma BST [8], e que a subárvore enraizada em x tenha k nós. A operação de *rebalanceamento* reorganiza a estrutura de uma subárvore com raiz em x, de modo que tenha as mesmas chaves, mas sua altura seja no máximo $log_2\ k$. O rebalanceamento pode ser feito usando um percurso em-ordem da subárvore com raiz em x. À medida que percorremos a árvore, colocamos os nós, na ordem, em uma lista. O ponto médio da lista será a raiz da nova subárvore, onde, como de costume, o ponto médio é (primeiro + último)//2. Todos os elementos à esquerda do ponto médio irão para o filho esquerdo, e todos os elementos à direita do ponto médio vão para o filho direito. Um exemplo é mostrado na Figura 2. Talvez a maneira mais natural de construir uma árvore seja usando recursão, como mostrado na Figura 3.

Notas

- O rebalanceamento de uma subárvore é uma operação puramente estrutural, que rearruma as conexões dos nós. Não devem ser criados quaisquer novos nós, ou executadas comparações de chaves, durante o rebalanceamento.
- Rebalancear uma subárvore de tamanho k deve levar tempo $O(k)$.

[8] `https://en.wikipedia.org/wiki/Binary_search_tree`

- A operação deve ser executada em uma subárvore, portanto, não esqueça de atualizar seus pais se necessário.

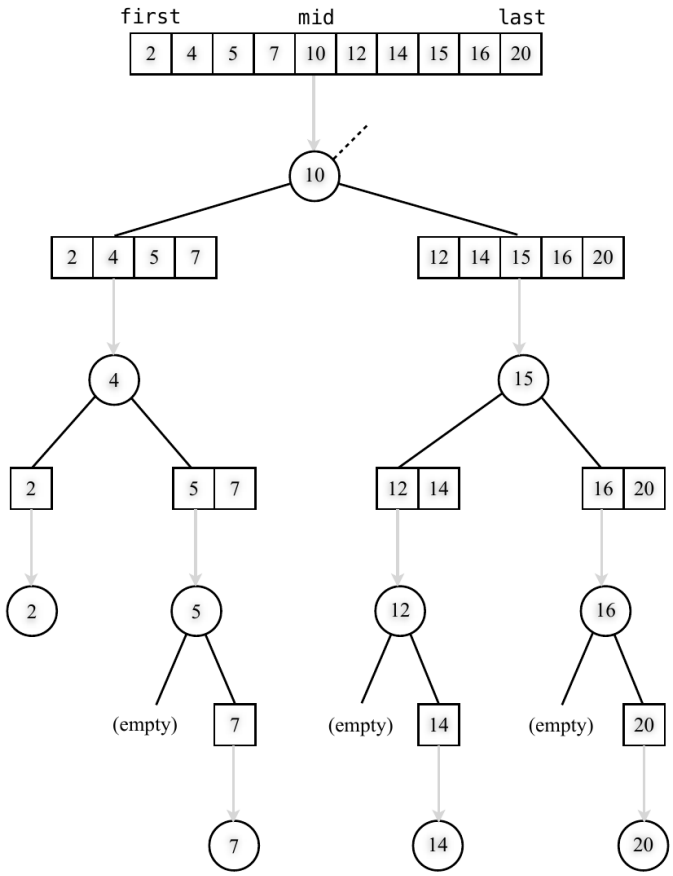

Figura 3: Rebalanceando uma subárvore.

3.2 Restaurando o balanceamento após Atualizações

Depois de atualizarmos uma árvore, devemos verificar se ela permanece α-balanceada. Se sim, nada mais precisa ser feito. Caso contrário, devemos rebalanceá-la. Para ser capaz de detectar rapidamente quando as condições de balanceamento - desigualdades (1) e (2) - são violadas, mantemos para

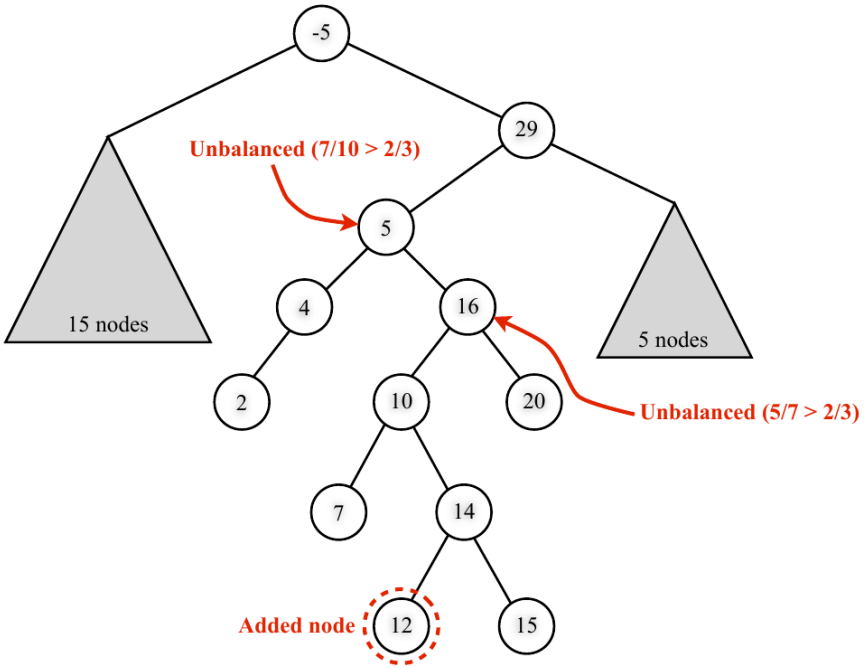

Figura 4: Adicionando a chave 12 a uma árvore balanceada, usando $\alpha = 2/3$. O nó contendo 5 é o nó desbalanceado mais alto.

cada nó uma contagem do número de elementos na subárvore desse nó. Sempre que um nó for adicionado ou removido, precisamos iterar na árvore ao longo do caminho até a raiz, começando com o pai do nó, atualizando as contagens de cada nó. Também precisamos verificar se algum nó ao longo do caminho se desbalanceou e identificar o nó desbalanceado mais alto (se houver) ao longo desse caminho. A operação de rebalanceamento deve ser executada no nó mais próximo da raiz.

Figura 4 ilustra uma árvore com 31 elementos antes da adição da chave 12. Usando um valor de $\alpha = 2/3$, a árvore está balanceada inicialmente. Após a adição de 12, dois nós ao longo do caminho até a raiz ficam desbalanceados: os nós contendo 5 e 16, respectivamente. Balanceamos no nó contendo 5, já que é aquele que está mais próximo da raiz.

4 Tarefas

Seu trabalho é implementar a classe BalancedBSTSet, usando árvores α-balanceadas amortizadas. O ponto de partida para a sua implementação deve ser o código exemplo [9], fornecido juntamente com este projeto. Especificamente, existem métodos públicos para fornecer uma visão, somente para leitura, da estrutura da árvore, e há um método público *rebalance()*, que deve implementar a operação de rebalanceamento descrita na seção 3.1. Para facilitar os testes e a correção, utilize o visualizador de árvores, escrito em PyOpenGL, fornecido [10].

Para evitar problemas com aritmética de ponto flutuante que podem ocorrer ao usar as desigualdades (1) e (2), representaremos α por duas variáveis de instância inteiras, top e bottom, que serão o numerador e denominador; i.e., $\alpha = top/bottom$. Então, as desigualdades (1) e (2) serão expressas como

$$\text{(número de elementos na subárvore esquerda de x)} \cdot \text{bottom} \leq \text{tamanho} \cdot \text{top}, \tag{3}$$

e

$$\text{(número de elementos na subárvore direita de x)} \cdot \text{bottom} \leq \text{tamanho} \cdot \text{top}. \tag{4}$$

O valor default deve ser top = 2 e bottom = 3 (i.e., $\alpha = 2/3$).

BalancedBSTSet possui uma classe interna *Node* que implementa um nó e você pode fazer a modificação que quiser nela.

A classe *Node*, fornecida com este projeto, define as seguintes variáveis para um nó de uma árvore de busca binária:

```
1 def class Node(object):
2     def __init__(self, key, parent):
3         self.data = key
4         self.parent = parent
5         self.counter = 0
6         self.left = None
7         self.right = None
```

As variáveis *left*, *right*, *parent*, e *data* são auto explicativas. A variável *counter* armazena o número total de elementos na subárvore enraizada neste

[9] http://orion.lcg.ufrj.br/python/ADs/BSTSet.py

[10] http://orion.lcg.ufrj.br/python/ADs/treeGL.py

nó. Esta variável é necessária para determinar quais nós, se houver algum, ficaram desbalanceados como resultado de uma atualização, e é usada para encontrar a raiz da subárvore na qual a operação de rebalanceamento deve ser aplicada (veja a Seção 3.2). A variável mantém o tamanho da subárvore inteira, ou separadamente os tamanhos das subárvores esquerda e direita. De qualquer modo, *counter* deve ser calculada em tempo constante.

A classe BalancedBSTSet possui dois métodos adicionais públicos:

```
1     def root(self):
2         """Retorna a raiz da árvore."""
3
4     def rebalance(self, bstNode):
5         """Executa a operação de rebalanceamento na subárvore,
6            que está enraizada no nó dado."""
```

Existe apenas um construtor.

```
1   def __init__(self, isSelfBalancing=False, top=0, bottom=0):
2      """Se isSelfBalancing for True, constrói uma árvore
3      auto-balanceada, com α = top/bottom.
4      Se bottom for zero, considere top = 2 e bottom = 3.
5      Se isSelfBalancing for False, constrói uma árvore sem
6      auto-balanceamento (top e bottom devem ser ignorados)."""
```

Em resumo, suas tarefas principais são as seguintes.

1. Implementar a operação *rebalance()* em BalancedBSTSet.

2. Modificar a classe *Node* e os métodos *add()*, *remove()*, e *BSTIterator.remove()* para manter contadores em cada nó, que devem ser calculados em $O(1)$.

3. Modificar os métodos *add()*, *remove()*, e *BSTIterator.remove()* para que, se a árvore for construída com o flag isSelfBalancing True, a árvore seja auto-balanceada. Ou seja, se uma operação causar o desbalanceamento de qualquer nó, o rebalanceamento deve ser executado automaticamente no nó desbalanceado mais alto (que estará sempre em algum lugar ao longo do caminho até a raiz).

4. O código a seguir retorna a interseção [11] [12] de duas BSTSet através de *iterators* [13]. Implemente as operações de união e diferença [14].

[11] `https://www.programiz.com/python-programming/methods/set/`
[12] `https://www.probabilitycourse.com/chapter1/1_2_2_set_operations.php`
[13] `https://opensource.com/article/18/3/loop-better-deeper-look-iteration-python`
[14] `http://orion.lcg.ufrj.br/python/ADs/peekable.py`

```
try:
   from peekable import peekable
except ImportError:
   ## return an iterator for a BSTSet.
   def peekable(it):
       return it.iterator()

## set intersection given two mutable ordered sequences.
 # @see https://docs.python.org/3.0/library/stdtypes.html \
                                        #typesseq-mutable
def set_intersection(itr1, itr2):
    p1 = peekable(itr1)
    p2 = peekable(itr2)
    result = type(itr1)()
    while p1.hasNext() and p2.hasNext():
        i1 = p1.peek()
        i2 = p2.peek()
        if i1 < i2:
            next(p1)
        elif i2 < i1:
            next(p2)
        else:
            result.append(i1)
            next(p1)
            next(p2)
    return result
```

Observe que os itens 1 e 2 podem ser feitos independentemente.

Notas

- A árvore deve manter contadores corretos nos nós, quer seja auto-balanceada ou não.
- Qualquer subárvore pode ser explicitamente rebalanceada usando o método *rebalance()*, quer seja auto-balanceada ou não.
- Perceba que sua implementação poderia ser utilizada para implementar conjuntos ordenados em Python. Embora existam coleções ordenadas (*OrderedDict*) [15] em Python, elas mantêm a ordem de inserção apenas [16] [17].

[15] https://docs.python.org/2/library/collections.html#collections.OrderedDict

[16] https://pythontips.com/2016/04/24/python-sorted-collections/

[17] https://pypi.org/project/ordered-set/

- O código de interseção é basicamente uma adaptação de $C^{++}/Java$. Não existe *peek()* ou *hasNext()* nos iteradores de python. A forma pythonica de resolver o problema seria usando exceções e avançando o iterador para poder ler um valor. Tente fazer dessa maneira. É um bom exercício...

5 Submissão

Entregue um arquivo zip chamado Nome_Sobrenome_AD1_PIG.zip, contendo todo o seu código-fonte para a classe BalancedBSTSet. O nome do diretório para o projeto deve ser AD1. Inclua a diretiva Doxygen [18] @author em cada classe do arquivo fonte. **Não entregue** arquivos .pyc.

Você é fortemente encorajado a escrever testes do unittest [19] à medida que desenvolver sua solução. No entanto, não é necessário entregar nenhum código de teste na AD1. Portanto, você pode compartilhar seus testes na plataforma.

A submissão deve ser feita através da plataforma. Por favor, siga as diretivas postadas na "Sala de Tutoria".

6 Notas

Uma parte significativa da sua nota será baseada nos testes do unittest [20] que serão aplicados na sua classe. O código será inspecionado para verificar que os métodos foram implementados conforme especificado. Documentação e estilo contarão cerca de 15% do total de pontos.

Penalidades por Atraso

Os projetos devem ser submetidos até às 24 horas do dia definido para a entrega.

[18] `https://www.stack.nl/~dimitri/doxygen/manual/starting.html`
[19] `http://pythontesting.net/framework/unittest/unittest-introduction/`
[20] `https://pymotw.com/2/unittest/`