Lista 2 - PMR3201

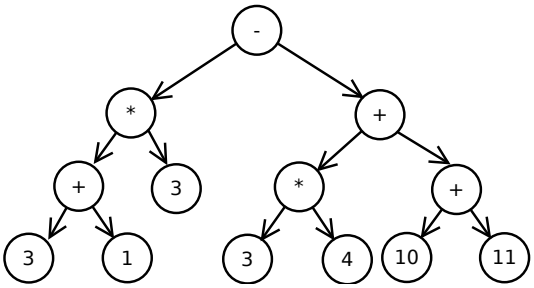
Fabio G. Cozman, Thiago Martins

10 de junho de 2018

Exercícios

Nota: Exercícios marcados com * têm solução no final da lista.

1. (*)Considere a árvore de expressão abaixo. Nessa árvore, qual é o nó raiz? Quais são as folhas? Qual é a altura da árvore? Qual é a profundidade do nó com dado 10? Qual é a expressão representada pela árvore, em notação infixa (justifique)?



2. (*)Considere a inserção dos seguintes números em uma árvore binária de busca: 30, 40, 24, 58, 48, 26, 11, 13. Desenhe a árvore após cada inserção.

3. Apresente um exemplo que contradiz a seguinte afirmação: a ordem de inserções em uma árvore binária de busca não importa; qualquer que seja a ordem de inserção dos elementos em uma árvore binária de busca, a árvore é a mesma.

4. (*) Uma lista *duplamente ligada* é uma lista na qual cada nó tem uma ligação para o próximo nó e também para o nó anterior. Uma lista duplamente ligada é frequentemente referida como *deque*. Suponha as seguintes estruturas de classes:

```
""
class Deque:
    class Node:
        def __init__(self, v = None):
            self.e = v
            self.prev = None
            self.next = None

        def __str__(self):
            return "Node:␣" + str(self.e)
```

```
def __init__(self):
    self._first = Deque.Node()  # sentinela vazio
    self._last = Deque.Node()   # sentinela vazio
    self._first.next = self._last
    self._last.prev = self._first
```

Nesta classe, os nós apontados por `self._first` e `self._last` são ditos *sentinelas*, e sinalizam o início e término da estrutura. Nenhum dado é armazenado neles.

Apresente o código para achar o nó do meio de um deque.

5. (*) Crie um método para concatenar duas lista ligadas e formar uma nova lista ligada.

6. (*) Apresente um método para concatenar dois deques com as estruturas esquematizadas no Exercício 4.

7. (*) Codifique um método que troca a posição de dois nós `x` e `y` em uma lista *simplesmente* ligada. O método deve receber referências aos dois nós, com a seguinte chamada:

   ```
   def troque(x, y):
   ```

   onde `x` e `y` são referências aos nós cujas posições devem ser permutadas.

8. (*) (P2 PMR2300 2015) Um vetor *esparso* é um vetor no sentido algébrico do termo, no qual a maioria dos coeficientes é zero. Um vetor esparso pode ser eficientemente representado na memória de um computador por meio de uma lista ligada que armazena somente os coeficientes não-nulos. Seja um coeficiente de vetor esparso representado por um elemento da classe `CoeficienteVetor`. O campo `indice` desta classe armazena o índice do coeficiente e o campo `valor` armazena o valor do ceoficiente. Um vetor esparso nesta implementação seria uma lista ligada com nós da classe `NoVetorEsparso` em que cada um, no seu campo `coef`, contém um coeficiente e no seu campo `n` o próximo nó, ou, no caso dos nós finais, `None`. Os coeficientes são armazenados em *ordem crescente* de índice do coeficiente. Por exemplo, nesta representação, o vetor de cinco dimensões $A = [0, 1.5, 0, 2.0, 0]$ é representado pela lista ligada:

   Note que $A_1 = 0, A_2 = 1.5, A_3 = 0, A_4 = 2.0, A_5 = 0$. Os coeficientes nulos do vetor (índices 1, 3 e 5) são representados *implicitamente* pela sua ausência.

   Escreva um código que calcula o *produto interno* de dois vetores esparsos A e B definido como $\sum_i A_i B_i$.

   Use a seguinte assinatura:

   ```
   def produto_interno(a, b):
   ```

   Onde `a` e `b` são os primeiros nós das listas ligadas que representam os vetores A e B. Seja N o número de coeficientes *não-nulos* do vetor A e M o equivalente para B. Tente obter um algoritmo com complexidade de ordem $\mathcal{O}(N + M)$.

Classe `NoVetorEsparso`:

```
class NoVetorEsparso:
    def __init__(self, valor, n):
        # Coeficiente
        self.coef = valor
        # Próximo Nó
        self.n = n
```

Classe `CoeficienteVetor`:

```
public class CoeficienteVetor {
    def __init__(self, i, v):
        self._indice = i
        self._valor = v
```

9. (*) Codifique um algoritmo para garantir que sequências de caracteres estejam balanceadas em relação a parênteses, colchetes e chaves. O método deve receber uma String contendo uma sequência de caracteres, e verificar o balanceamento dos parênteses, colchetes e chaves.

10. Codifique um método que inverte uma fila. O método recebe um objeto do tipo `FilaAr` e deve retornar uma fila invertida.

11. (*) (P2 PMR2300 2015) Uma fila é uma estrutura de dados na qual o elemento retirado é o *mais antigo* a ser inserido. A classe `FilaCircular` implementa uma fila sobre um arranjo simples. O método `enqueue(x)` enfileira um novo objeto, o método `dequeue()` retira um objeto da fila e `len(f)` retorna a quantidade de objetos enfileirados na fila `f`.

```
class FilaCircular():
    def __init__(self):
        self._dados = [None]*10
        self._indiceVaiSair = 0
        self._indiceEntrou=len(self._dados)-1
        self._tamanho = 0

    def __len__(self):
        return self._tamanho

    def _incrementa(self, indice):
        indice += 1
        if indice == len(self._dados):
            indice = 0
        return indice

    def _duplique_seq(self):
        novo = [None]*2*len(self._dados)
        for i in range (self._tamanho):
            novo[i] = self._dados[self._indiceVaiSair]
            self._indiceVaiSair = self._incrementa(self._indiceVaiSair)

        self._dados = novo
        self._indiceVaiSair = 0
        self._indiceEntrou = self._tamanho - 1;


    def enqueue(self, x):
       if self._tamanho == len(self._dados):
          self._duplique_seq()
       self._indiceEntrou = self._incrementa(self._indiceEntrou)
       self._dados[self._indiceEntrou] = x
       self._tamanho += 1

    def dequeue(self):
       if self._tamanho == 0:
          raise IndexError("Fila vazia")
```

```
        self._tamanho -= 1
        x = self._dados[self._indiceVaiSair]
        self._dados[self._indiceVaiSair] = None
        self._indiceVaiSair = self._incrementa(self._indiceVaiSair)
        return x
```

a) Considere o código abaixo:

```
1    f = FilaCircular()
2    f.enqueue(1)
3    f.enqueue(2)
4    f.enqueue(3)
5    f.enqueue(f.dequeue())
6    f.enqueue(f.dequeue())
7    a = f.dequeue()
```

Mostre esquemas do estado da fila destacando o *último* elemento enfileirado e o *primeiro* elemento a ser retirado após a execução das linhas 4, 5, e 6. Qual o valor da variável `a` *após* a execução da linha 7?

b) Uma pilha é uma estrutura de dados na qual o elemento a ser retirado é o *último* a ser inserido. Mostre que é possível implementar uma pilha usando uma fila. Para tanto, complete a implementação da classe `PilhaPorFila` a seguir, com o corpo dos métodos:

- `push(self, x)`: Adiciona o objeto `x` à pilha.
- `pop(self)`: Remove e retorna o objeto mais recente a ser inserido, ou lança a exceção `ErroPilhaVazia` se não há mais objetos empilhados.
- `__len__(self)`: Retorna a quantidade de objetos empilhados.

```
class PilhaPorFila:
    def __init__(self):
        _f = FilaCircular()

    def push(self, x):
        ⋮

    pop(self):
        ⋮

    __len__(self):
        ⋮
```

12. (*) Considere a seguinte expressão em notação pós-fixa (notação polonesa reversa):

$$ABC*+C*D+$$

Se $A = 2$, $B = 3$, $C = 4$, $D = 5$, qual o valor da expressão? Suponha que a expressão é avaliada através de uma pilha; desenhe o conteúdo da pilha ao encontrar cada operando.

13. Dada a árvore abaixo, imprima os nós em ordem interior, ordem anterior, e ordem posterior.

14. Escreva em Python funções *não* recursivas para imprimir o conteúdo de uma árvore binária em ordem anterior, interior e posterior. Considere que os nós da arvore são implementados pela Classe `NoBinario`:

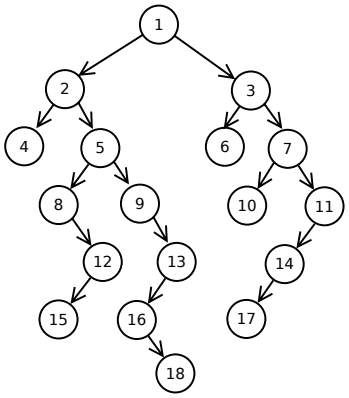


Figura 1: Árvore da questão 13

```
class NoBinario:
    def __init__(self, x, e = None, d = None):
       self.dado = x
       self.esquerda = e
       self.direita = d
```

Sugestão: Considere como o computador controla o fluxo de execução das chamadas recursivas.

15. (*) Demonstre que o número máximo de elementos em uma árvore binária de altura h é $2^h - 1$.

16. (*) Em uma árvore binária, um nó com 2 filhos é dito *completo*. Demonstre que o número de nós completos mais um é igual ao número de folhas.

17. (*)(P2 PMR3201 2015) Uma árvore binária de busca é uma árvore binária na qual para qualquer sub-árvore todos os elementos à esquerda da raiz são estritamente *menores* do que a raiz, e todos os elementos à direita da raiz são estritamente *maiores* que a raiz. Todas as inserções em uma árvore binária são feitas em novas folhas.

    a) Mostre o resultado, na forma de diagramas de árvores, das seguintes operações, partindo de uma árvore vazia:

        1 Inserção do valor 3.
        2 Inserção do valor 4.
        3 Inserção do valor 5.
        4 Inserção do valor 2.
        5 Inserção do valor 1.
        6 Remoção do valor 3.
        7 Remoção do valor 5.

    b) Uma árvore AVL é uma árvore binária de busca auto-balanceada, na qual a maior diferença de altura entre uma sub-árvore direita e esquerda é 1. Esta

propriedade é mantida em operações de inserção e remoção através de *rotações*, nas quais o nível hierárquico de um nó e um de seus descendentes é permutado. Repita o item anterior considerando agora uma árvore AVL.

18. (*) Considere uma árvore binária de busca (ou seja, todo nó tem uma chave maior que a chave do filho esquerdo e menor que a chave do filho direito). Sabemos que em tal estrutura, uma ordem interior produz uma sequência de nós com chaves em ordem crescente. Suponha que uma árvore binária de busca seja implementada de forma que todo nó mantenha um valor inteiro `sizeOfSubtree`, que armazena o número de nós na sub-árvore com raiz no próprio nó. Por exemplo, se um nó tem dois filhos e esses filhos não tem filhos, então `sizeOfSubtree` é igual a 3.

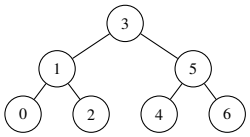
    Codifique um método `kth` na classe `SpecialNode` que encontra o k-ésimo elemento na sub-árvore com raiz no nó de chamada – o k-ésimo elemento alcançado em ordem interior. Ou seja, se `n` é um `SpecialNode` e chamarmos `n.kth(3)`, então o método deve retornar o conteúdo de `key` no terceiro nó (em ordem crescente) na sub-árvore cuja raiz é `n`. Assuma que o usuário sempre chama `n.kth(j)` com um valor de j que é maior ou igual a 1 e menor ou igual ao número de nós na sub-árvore com raiz em n.

19. (*) (P2 PMR2300 2015) Uma árvore de busca binária é uma árvore binária na qual cada nó é *estritamente maior* que *todos* os elementos da sua sub-árvore esquerda e *estritamente menor* que *todos* os elementos da sua sub-árvore direita (vide exemplos a seguir). Esta propriedade permite a rápida localização de elementos nela contidos. A classe `NoBinarioFloat` contém nós de uma árvore binária que armazena números em ponto flutuante. O campo `val` contém o número armazenado no nó, o campo `esquerda` contém uma referência para a raiz da sub-árvore esquerda e o campo `direita` contém uma referência para a raiz da sub-árvore direita. Escreva uma função em java que verifica se uma determinada árvore binária formada por `NoBinarioFloat` é efetivamente uma árvore de busca binária. Use a seguinte assinatura:

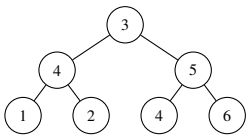
    ```
    def checa_arvore(n):
    ```

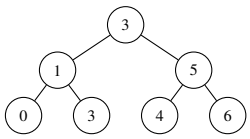
    Onde `n` é o nó raiz da árvore a ser verificada. A função deve retornar verdadeiro caso a árvore seja uma árvore de busca, falso caso contrário. Qual a ordem de complexidade do algoritmo implementado em função do número de elementos da árvore N?

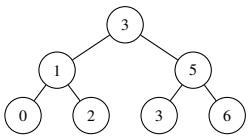
    Exemplo de árvore binária de busca:

    Exemplos de árvores binárias que *não são* de busca:

    Classe `NoBinario`

```
class NoBinario:
    def __init__(self, x, e = None, d = None):
       self.dado = x
       self.esquerda = e
       self.direita = d
```

20. Considere uma linguagem onde os caracteres tem a seguinte distribuição (indicamos espaço por "[ ]"):

D	E	M	O	R	A	[ ]
10	18	9	20	12	23	8

    Para essa linguagem, monte o código de Huffman. Codifique a palavra DEMORA no código obtido. Usando o código obtido, qual seria o tamanho médio de um texto com 100 caracteres?

21. (*) Suponha que uma árvore binária contém um caracter em cada nó. A ordem interior de visita aos nós produz a sequência ABCEDFJGIH. A ordem anterior de visita aos nós produz a sequência JCBADEFIGH. Desenhe uma árvore binária que produza essas ordens.

22. Uma classe que implementa árvores binárias tem a seguinte definição:

```
class NoArvoreBinaria:
   def __init__(self, d):
     self.dado = d
     self.esquerdo = None
     self.direito = None

     # Inserir, remover, etc.
     ...
```

    Deseja-se incluir nessa classe um método que faça busca por nível em árvore, a partir de um nó que chama o método:

```
def busca_por_nivel(d):
  # Retorna True se encontrou d; retorna False caso contrario.
```

    Descreva um algoritmo que realiza busca por nível e escreva a função `buscaPorNivel`. Qual é a complexidade da busca implementada no pior caso, para árvore contendo N nós, em notação BigOh?

    Sugestão: use uma estrutura de dados auxiliar; recursão direta na árvore não resolve essa questão.

23. (*)Considere o seguinte algoritmo de ordenação de um vetor no qual *todos os itens são distintos*:

    - Adicione os elementos do vetor sequencialmente a uma árvore binária de busca
    - Percorra a árvore na ordem interior e escreva os valores dos nós sequencialmente ao vetor.

    (a) Mostre que este algoritmo termina em tempo finito e efetivamente produz uma ordenação do vetor.

(b) Qual o complexidade deste algoritmo em termos do tamanho do vetor N?

(c) Repita o item b) considerando agora o uso de uma árvore AVL:

24. Considere que um texto foi codificado usando os caracteres na tabela abaixo. A tabela indica a frequência com que cada caracter apareceu no texto:

a	d	i	m	n	o	q	r	x	espaço
27	10	18	3	4	5	2	7	1	23

Obtenha o código de Huffman para esse texto. Codifique a mensagem "adiado o dia do xaxa" no código gerado. Se o texto original continha 10.000 caracteres, qual o tamanho do texto em bits após a compressão usando o código gerado?

25. (*)Considere o heap formado pelo vetor de números $\{10, 7, 9, 5, 3, 2, 8, 4, 3, 2, 2, 0\}$. Mostre o heap resultante da remoção do seu elemento de maior valor.

26. (*) Em um heap, quais são as possíveis posições para o *menor* elemento? Escreva um algoritmo para encontrar o menor elemento em um heap. Qual a complexidade deste algoritmo em função do número total de elementos N?

27. (*)Um resultado fundamental para a análise da construção de um heap "de baixo para cima" (com FixHeapDown) é o de que no heap o número de nós no nível h é no máximo $\lceil N/2^h \rceil$, onde N é o número total de elementos da árvore. Demonstre este resultado.

28. (*)(P2 PMR2300 2015) Um heap binário é uma árvore binária completa (todos os níveis exceto o último cheios) preenchido da esquerda para a direita na qual cada nó é maior ou igual a seus filhos. Heaps binários são armazenados em vetores, resultantes da varredura da árvore binária *em largura*. Dada a rígida estrutura, é possível converter um vetor em árvore e vice-versa.

a) Mostre na forma de árvore binária a estrutura do heap binário representado pelo vetor $[6, 6, 2, 5, 3]$.

b) Novos elementos podem ser adicionados ao heap de forma eficiente. Para tanto, o novo elemento é anexado *ao final* do heap. Em seguida, a sub-árvore afetada é "corrigida", movendo-se o elemento novo para sua posição correta na hierarquia. O processo prossegue heap acima, sub-árvore a sub-árvore, até que não sejam necessárias mais alterações. Mostre o resultado na forma de árvores da inserção sucessiva dos valores 3, 8, e 7.

c) Da mesma maneira, *o maior valor* em um heap (nó raiz) também pode ser eficientemente removido. Para tanto, ele é trocado de posição com o último elemento, e removido do vetor. Em seguida, a sub-árvore raiz é corrigida movendo-se o novo elemento para a posição adequada. O processo prossegue heap abaixo até que não sejam necessárias mais alterações. Mostre o resultado na forma de árvores da remoção sucessiva de 3 dos maiores elementos do heap produzido no item b).

29. (*)Um algoritmo de ordenação é dito *estável* se ele *não altera* a posição relativa de elementos de valor de comparação igual. Por exemplo, considere a ordenação alfabética de caracteres, independentemente da letra ser maiúscula ou minúscula. Um algoritmo *estável* produziria a seguinte ordenação da cadeia `cbHebhBhjE`:

`bbBceEHhhj`. Um algoritmo não-estável poderia, por exemplo, produzir a seguinte ordenação: `BbbcEehhHj`.

O heapsort é um algoritmo de ordenação estável? Justifique.

30. (*)(PSUB 2016) Um heap binário é uma árvore binária completa (todos os níveis exceto o último cheios) preenchido da esquerda para a direita na qual cada nó é maior ou igual a seus filhos. Heaps binários são armazenados em vetores, resultantes da varredura da árvore binária *em largura*. Dada a rígida estrutura, é possível converter um vetor em árvore e vice-versa.

    Há dois algoritmos relevantes para alterações em um heap binário implementados nas seguintes funções:

    `def fix_heap_down(a, start, end)` corrige um heap binário no qual um nó é menor ou igual ao seu pai mas possivelmente menor do que os seus filhos. O nó é empurrado "para baixo" no heap até encontrar uma posição adequada. O parâmetro `a` é o vetor que contém o heap, o parâmetro `start` indica o nó cuja posição está incorreta e o parâmetro `end` indica a última posição do heap no vetor.

    ```
    def fix_heap_down(a, start, end):
        pai = start
        filho = 2*pai+1
        while filho<=end:
            maior = pai;
            if a[pai]<a[filho]:
                maior = filho
            if filho+1<=end and a[maior]<a[filho+1]:
                maior = filho+1
            if maior==pai:
                break # Acabou
            aux = a[pai]
            a[pai] = a[maior]
            a[maior] = aux
            pai = maior
            filho = 2*pai+1
    ```

    `def fix_heap_up(a, fim)` corrige um heap binário no qual o nó final é possivelmente maior do que o seu pai.

    O nó é empurrado "para cima" no heap até encontrar uma posição adequada. O parâmetro `a` é o vetor que contém o heap, e o parâmetro `end` indica a última posição do heap no vetor, onde está o elemento possivelmente errado.

    ```
    def fix_heap_up(a, end):
        pai = (end-1) // 2
        while pai!=end:
            if a[pai]<a[end]:
                aux = a[pai]
                a[pai] = a[end]
                a[end] = aux
                fim = pai
                pai = (fim-1)//2
            else: return
    ```

    Escreva uma função que remove um elemento *arbitrário* de um heap mantendo a sua estrutura (mas naturalmente reduzindo o seu tamanho). Use a seguinte assinatura:

```
def remove_from_heap(a, end, pos)
```

Onde `a` é o vetor que contém o heap binário, `end` é a posição do fim do heap no vetor e `pos` é o índice no vetor, entre 0 e `end`, do elemento a ser removido. A função deve retornar a nova posição do final do heap (ou seja, `end`-1). A função deve operar com uma complexidade $\mathcal{O}(\log N)$ ou melhor.

31. Escreva, usando como molde a classe `FilaCircular` (presente no código escrito em sala distribuido aos alunos) uma class FilaPrioritariaAr.

    Esta classe recebe objetos *comparáveis*, ou seja, objetos que suportam a operação `x < y`, que retorna `True` se o valor em `x` for em algum sentido *menor* do que o em `y` e retorna `False` caso contrário.

    Implemente os métodos `enqueue(x)` e `dequeue()`. A função `dequeue()` deve retornar o *maior* elemento enfileirado em tempo $\log N$, onde N é o número de elementos armazenados. A função `enqueue(x)` deve inserir um novo elemento em tempo *amortizado* $\log N$. O caso em que o vetor base está cheio e deve ser duplicado para permitir inserção de novos elementos deve consumir tempo N.

    Sugestões:

    A estrutura base é um vetor, como na classe `FilaCircular`. Este vetor deve formar um heap. A operação FixHeapUp pode ser usada para adicionar um novo elemento, adicionando-o ao final do vetor e "consertando-o" levando o elemento para cima na hierarquia até o local apropriado. A operação FixHeapDown pode ser usada para retirar o maior elemento, substituindo-o pelo último e "consertando-o" levando o último elemento para baixo na hierarquia até o local apropriado.

32. Considere uma hashtable de encadeamento separado, na qual são inseridas árvores binárias de busca. Cada árvore contém N nós, e cada nó contém um dado. Suponha que N árvores foram inseridas na hashtable, e que todos os dados em uma determinada árvore sejam mapeados no mesmo elemento (*bucket*) da hashtable.

    - Suponha primeiro que houve espalhamento perfeito, ou seja, cada posição da hashtable contém apenas uma árvore.
      - Assuma que toda árvore inserida na hashtable está balanceada. Qual é a complexidade de encontrar um dado nessa estrutura, no pior caso, em notação BigOh?
      - Agora abandone a suposição que as árvores estão balanceadas. Qual é a complexidade de encontrar um dado nessa estrutura, no pior caso, em notação BigOh?
    - Agora suponha que não houve espalhamento perfeito.
      - Assuma que toda árvore inserida na hashtable está balanceada. Qual é a complexidade de encontrar um dado nessa estrutura, no pior caso, em notação BigOh?
      - Agora abandone a suposição que as árvores estão balanceadas. Qual é a complexidade de encontrar um dado nessa estrutura, no pior caso, em notação BigOh?

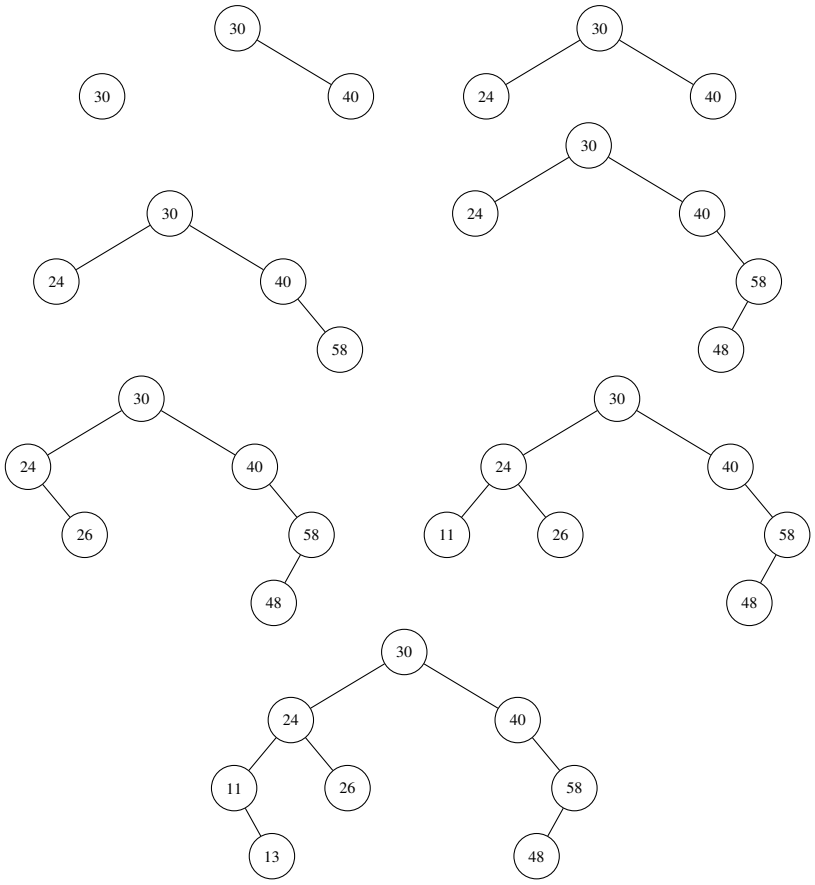
Soluções

Exercício 1:

O nó raiz é o no com o símbolo '-'. Os nós folha, da esquerda para a direita, são '3', '1', '3', '3', '4', '10', '11'. A profundidade do nó '10' é 3. A representação, em notação *infixa*, é:

$$(3 + 1) \times 3 - (3 \times 4 + (10 + 11))$$

Exercício 2:



Exercicio 4: A idéia em um deque é que cada nó aponta para o próximo nó e para o nó anterior na lista. Percorremos um deque fazendo:

```
q = self._first.next
while not (q is self._last):
    q = q.next
```

Por exemplo, suponha que temos um objeto `e` e queremos achar um nó que contém esse objeto em um deque.

Então podemos fazer:

```
        q = self._first.next
        while not (q is self._last):
            if q.v == e:
                break
            q = q.next
```

Ao fim desse processo, `p` é ou um nó que contém `e` ou o nó final `self._last`.

Retornando ao exercício, considere a solução a seguir, que primeiro varre o deque contando o número de nós, e depois vai até a metade (note que estamos usando sentinelas):

```
    def middle_element(self):
        """Encontra elemento do meio com contador"""
        i = 0
        q = self._first.next

        while not (q is self._last):
            q = q.next
            i += 1

        meio = i // 2
        q = self._first
        for i in range(meio):
            q = q.next
        return q.e
```

É fácil resolver o exercício sem usar contadores para deques com número ímpar do nós. A solução é começar a varrer o deque do início e do final, a cada passo incrementando um lado e decrementando outro, até atingir o mesmo nó (que é o nó do meio).

```
    def middle_element_odd(self):
        """Encontra elemento do meio SEM contador
        Para deque com numero impar de elementos!"""
        p = self._first.next
        q = self._last.prev
        while not (q is p):
            p = p.next
            q = q.prev
        return p.e
```

Exercicio 5:

Vamos resolver esse problema assumindo duas listas (listas normais sem "sentinelas"), tomando as estruturas:

```
class ListaLigada:
    class Node:
        def __init__(self, e):
            self.element = e
            self.next = None

    def __init__(self):
        self._first = None
```

Então:

```
    def concatena(self, lista):
        q = None
        # Achamos o pai do ultimo no de self
        p = self._first
```

```
        while p:
            q = p
            p = p.next
        # Agora ligamos o ultimo no com a lista:
        q.next = lista._first
```

O custo desse procedimento é $\mathcal{O}(n)$, onde n é o tamanho de `lista`.

Exercício 6:

Supondo deques *com* sentinelas, basta fazer:

```
def concatena(self, other):
    if self is other:
        raise Exception("Auto-concatenação não suportada")
    other._last.prev.next = self._last
    self._last.prev.next = other._first.next
    other._first.next.prev = self._last.prev
    self._last.prev = other._last.prev
    # Remove os nos da lista other
    other._first.next = other._last
    other._last.prev = other._first
```

Note que esse processo *modifica* os deques iniciais. O Deque em `self` recebe o Deque concatenado e o Deque em `other` é *apagado* Ademais, ele *não* é capaz de concatenar um deque consigo mesmo. Para fazer a mesma coisa sem destruir os deques iniciais, é preciso efetivamente criar novos nós para cada um dos nós em `self` e `other`. Ou seja, é preciso criar um deque nova e varrer os outros dois deques (primeiro `self` e depois `other`), criando um nó para cada elemento nesses deques. Faça isso como um exercício.

Exercício 7:

Se tivéssemos que trocar apenas os conteúdos de `x` e `y`, seria fácil:

```
temp = x.element
x.element = y.element
y.element = temp
```

No entanto temos que trocar efetivamente os nós. Para isso, temos que encontrar o pai e o filho de `x` e `y`. Vamos assumir que `x` e `y` não são `None`. Fazemos então:

```
def troque_nos(self, x, y):
    """Troca dois nos"""
    pai_x = None
    pai_y = None
    p = self._first
    while p:
        if p.next == x:
            pai_x = p
        p = p.next
    q = self._first
    while q:
        if q.next == y:
            pai_y = q
        q = q.next
    filho_x = x.next
    filho_y = y.next
    pai_x.next = y
    y.next = filho_x
    pai_y.next = x
    x.next = filho_y
```

Exercício 8:

É possível explorar a ordenação das listas para produzir um algoritmo de ordem $\mathcal{O}(N + M)$. Para tanto, varre-se sequencialmente as listas procurando-se incrementar sempre o nó da lista com o *menor* indice. Quando os valores dos índices coincide, faz-se o produto dos valores e adiciona-se o resultado ao total. O algoritmo prossegue até uma das listas acabar.

```
def produto_interno(a, b):
    result = 0
    while a and b:
        if a.coef.indice < b.coef.indice:
            a = a.next;
        elif a.coef.indice > b.coef.indice:
            b = b.next;
        else: # iguais
            result += a.coef.valor*b.coef.valor
            a = a.n
            b = b.n
    return result
```

Exercício 9

O problema aqui é garantir que cada parênteses (ou colchete ou chave) que seja aberto seja também fechado, e que não haja problema na ordem de abertura e fechamento. A solução mais simples é através de uma pilha. Suponha que a entrada do algoritmo seja um arranjo contendo caracteres. Assuma que uma classe `Pilha` como vista em aula esteja disponível, porém assuma que esta classe manipule diretamente caracteres (ou seja, dê `push` e `pop` em caracteres). Assuma que `pop` retorna um espaço quando a pilha está vazia.

Então, colocamos cada "abertura" na pilha, e retiramos uma "abertura" quando encontramos um "fechamento". Os erros possíveis são: uma "abertura" nunca é fechada (pilha não está vazia no final), uma "abertura" não casa com um "fechamento", ou um "fechamento" não tem abertura.

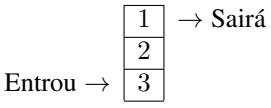
```
def verifica_consistencia(entrada):
    pilha = []
    try:
        for c in entrada:
            if c in {'(', '[', '{'}:
                pilha.append(c)
            elif c==')' and pilha.pop() !='(':
                return
            elif c==']' and pilha.pop() !='[':
                raise ValueError("Falta algum fechamento")
            elif c=='}' and pilha.pop() !='{':
                raise ValueError("Falta algum fechamento")
        if pilha:
            raise ValueError("Falta algum fechamento")
    except IndexError:
        raise ValueError("Falta alguma abertura")
```

Essa função usa uma simples sequência em Python como pilha. Caso a entrada seja consistente, a função não faz nada. Caso contrário, lança uma exceção do tipo `ValueError`.

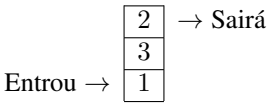
Exercício 11:

a)

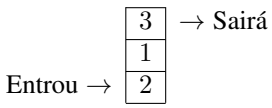
- Após linha 4:



- Após linha 5:



- Após linha 6:



O valor final da variável `a` é 3, o *último* elemento a ser inserido antes das linhas 5 e 6.

b) Como mostra o item a), é possível levar o último elemento de uma fila até a posição de remoção fazendo $n-1$ operações do tipo `f.enqueue(f.dequeue())`, onde n é a quantidade de elementos enfileirados (recuperável pelo método `tamanho()`). O código final é:

```
class PilhaPorFila:
    def __init__(self):
        _f = FilaCircular()

    def push(x):
        self._f.enqueue(x)

    def pop(self):
        if not len(self._f)
            raise ErroFilaVazia()
        for _ in range(len(self._f)):
            self._f.enqueue(self._f.dequeue())
        return self._f.dequeue()

    def __len__():
        return len(self._f)
```

Exercício 12

A sequência abaixo mostra a pilha em vários momentos do algoritmo, indicando as operações que levam a mudanças na pilha:

$$\left|\begin{array}{c} C \\ B \\ A \end{array}\right| * \left|\begin{array}{c} B*C \\ A \end{array}\right| + \left|\begin{array}{c} C \\ A+B*C \end{array}\right| * \left|\begin{array}{c} D \\ C*(A+B*C) \end{array}\right| + \left|\ D+C*(A+B*C)\ \right|$$

Para os valores indicados, temos $5 + 4(2 + 3 * 4) = 61$.

Exercício 13:

Ordem posterior: 4, 15, 12, 8, 18, 16, 13, 9, 5, 2, 6, 10, 17, 14, 11, 7, 3, 1.

Ordem interior: 4, 2, 8, 15, 12, 5, 9, 16, 18, 13, 1, 6, 3, 10, 7, 17, 14, 11.

Ordem anterior: 1, 2, 4, 5, 8, 12, 15, 9, 13, 16, 18, 3, 6, 7, 10, 11, 14, 17.

Exercício 14:

Antes, vejamos as tradicionais implementações recursivas destes algoritmos (nota: o código abaixo supõe que as árvores são sempre *não-nulas*):

```
def anterior_rec(arvore):
    # Fase 1
    print(arvore.dado)
    if arvore.esquerda:
        anterior_rec(arvore.esquerda)
    # Fase 2
    if arvore.direita:
        anterior_rec(arvore.direita)

def interior_rec(arvore):
    # Fase 1
    if arvore.esquerda:
        interior_rec(arvore.esquerda)
    # Fase 2
    print(arvore.dado)
    if arvore.direita:
        interior_rec(arvore.direita)

def posterior_rec(arvore):
    # Fase 1
    if arvore.esquerda:
        posterior_rec(arvore.esquerda)
    # Fase 2
    if arvore.direita:
        posterior_rec(arvore.direita)
    # Fase 3
    print(arvore.dado)
```

Estes algoritmos estão divididos em *fases*, separadas por *retornos* da chamada recursiva. Note que a ordem posterior possui *três* fases.

Claramente, este problema *não* pode ser trivialmente resolvido com uma transformação de *tailcall*. É necessário usar uma pilha externa para substituir a pilha de controle de programa.

Esta pilha deve armazenar, além do parâmetro `arvore`, passado a cada chamada recursiva, o valor da *fase* a ser executada. Na pilha de programa, o papel deste valor é desempenhado pelo controle de endereço de retorno das chamadas de função.

Segue as implementações das versões recursivas:

```
def anterior(arvore):
    pilha = [arvore, 1]
    while len(pilha)>0:
        fase = pilha.pop()
        no = pilha[-1]
        if fase==1:  # Fase:  1
            pilha.append(2) # Próxima fase:  2
            print(no.dado)
            if no.esquerda:
                pilha.append(no.esquerda)
                pilha.append(1)
        else: # Fase:  2
            pilha.pop() # Última fase
            if no.direita:
                pilha.append(no.direita)
                pilha.append(1)

def interior(arvore):
    pilha = [arvore, 1]
    while len(pilha)>0:
```

```
            fase = pilha.pop()
            no = pilha[-1]
            if fase==1:  # Fase:  1
                pilha.append(2) # Próxima fase:  2
                if no.esquerda:
                    pilha.append(no.esquerda)
                    pilha.append(1)
            else: # Fase:  2
                pilha.pop()  # Última fase
                print(no.dado)
                if no.direita:
                    pilha.append(no.direita)
                    pilha.append(1)

    def posterior(arvore):
        pilha = [arvore, 1]
        while len(pilha)>0:
            fase = pilha.pop()
            no = pilha[-1]
            if fase==1:  # Fase:  1
                pilha.append(2) # Próxima fase:  2
                if no.esquerda:
                    pilha.append(no.esquerda)
                    pilha.append(1)
            elif fase==2: # Fase:  2
                pilha.append(3) # Próxima fase:  3
                if no.direita:
                    pilha.append(no.direita)
                    pilha.append(1)
            else:       # Fase:  3
                pilha.pop()  # Última fase
                print(no.dado)
```

Nota: Como visto em aula, a implementação de `anterior` pode ser simplificada, combinando-se a fase 1 e 2 e elimando-se o controle de fase da pilha:

```
def anterior(arvore):
    if arvore == None: return # vazia
    pilha = [arvore]
    while len(pilha)>0:
        no = pilha.pop()
        print(no.dado)
        # Note a inversão de direita/esquerda
        if no.direita: pilha.append(no.direita)
        if no.esquerda: pilha.append(no.esquerda)
```

Este tipo de simplificação *não* se aplica às outras ordens.

Exercício 15

A raiz está na profundidade 0, seus filhos na profundidade 1, e assim por diante. A altura da árvore é igual à maior profundidade de um nó na árvore mais 1.

O número máximo de nós ocorre quando todos os nós tem 2 filhos; nesse caso, o número de nós é:

$$\sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1.$$

Exercício 16:

Denote por C o número de nós completos e por F o número de folhas. A maneira mais simples de demonstrar isso é por indução finita. Considere uma árvore com

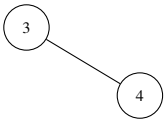
um único nó; nesse caso $C = 0$ e $F = 1$ (portanto a afirmação $C + 1 = F$ é verdadeira). Considere agora uma árvore qualquer com n nós e suponha que $C + 1 = F$ vale para essa árvore. Suponha agora que um nó seja adicionado a essa árvore. O pai desse nó tem que ter zero ou um filho (pois a árvore é binária). Se o pai tem zero filhos, o pai é uma folha; a adição do novo nó não muda nem C nem F e portanto $C + 1 = F$ por hipótese. Se o pai tem um filho, agora o pai ficou completo (portanto C foi incrementado) e o novo nó é uma folha nova (portanto F foi incrementado) e o resultado é que $C + 1 = F$.

Exercício 17:

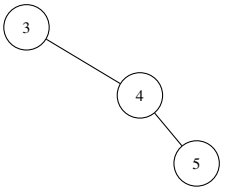
a)

1 Inserção do valor 3.

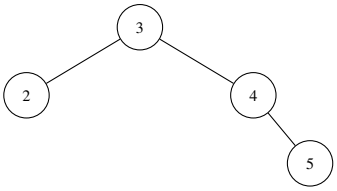
2 Inserção do valor 4.



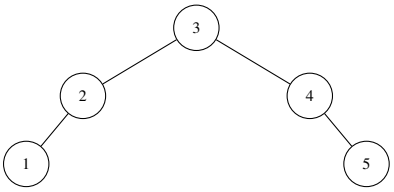
3 Inserção do valor 5.



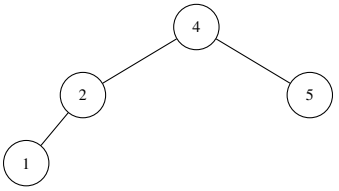
4 Inserção do valor 2.



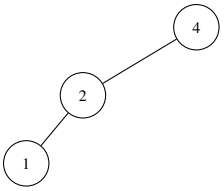
5 Inserção do valor 1.



6 Remoção do valor 5.



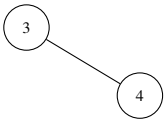
7 Remoção do valor 3.



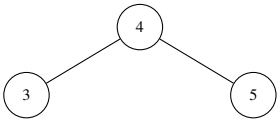
b)

1 Inserção do valor 3.

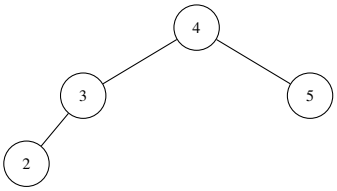
2 Inserção do valor 4.



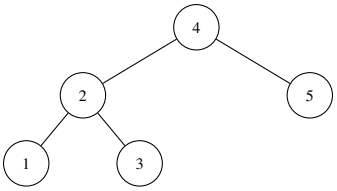
3 Inserção do valor 5.



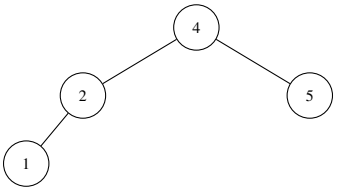
4 Inserção do valor 2.



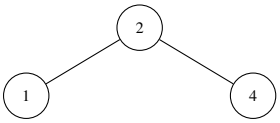
5 Inserção do valor 1.



6 Remoção do valor 3.



7 Remoção do valor 5.



Exercício 18:

Para encontrar a solução, imagine um nó A com dois filhos B e C. Suponha que B tenha `sizeOfSubtree` igual a n e C tenha `size OfSubtree` igual a m. Ou seja, existem n nós *antes* de A na árvore que começa em A, e m nós *depois* de A na árvore que começa em A. Ou seja, A é o $(n+1)$-ésimo nó. Portanto se tivermos `A.kth(j)`, temos três opções:

1. Se $j = n + 1$, então A é o nó procurado.
2. Se $j < n + 1$, então o j-ésimo nó está na sub-árvore com raiz em B.
3. Se $j > n + 1$, então o j-ésimo nó está na sub-árvore com raiz em C.

Temos:

```
class SpecialNode()

    ...


    def kth(self, j):
        """Encontre k-esimo elemento na ordem interior"""
        """presume j>0"
        int n = 0;
        if self._e
            n = self._e._size
        if j == (n+1):
            return self._key
        if j < (n+1)
            return self._e.kth(j)
        if j > (n+1):
            return self._d.kth( j - (n+1) ) )
```

```
    ...
}
```

Note que o problema assume que `kth` nunca é chamado com um valor menor que um ou maior que o número total de nós, portanto não é preciso verificar essas condições. Tente imaginar o que mudaria se essas condições tivessem que ser verificadas; seria preciso verificar o número de nós no filho direito para evitar "estouro".

Exercício 19:

Este é um problema que a um olhar desatento pode parecer erroneamente simples. De fato, *não basta* verificar a ordenação local de um nó e seus filhos imediatos. Como pode-se ver nos exemplos do enunciado, há árvores binárias com ordenação local válida, mas que ainda assim não são árvores binárias de busca.

Há diversas soluções possíveis para o problema. Eis algumas:

- *Validação por busca*: Em uma árvore de busca binária, como menciona o enunciado, os valores de *todos* os nós podem ser rapidamente encontrados via busca binária. Para tanto, implementa-se a sub-função `procura_valor(v, r)` que retorna a referência ao nó no qual está contido o valor `v`, ou `null` se o valor não pode ser encontrado. Esta função parte da árvore `r`. Em seguida, percorre-se *todos* os nós da árvore apresentada e verifica-se para cada nó se o seu valor foi encontrado efetivamente no próprio nó. A ordem de varredura neste caso não importa. Em particular, aqui segue-se a ordem em profundidade anterior, ou seja, o valor do nó atual é verificado, em seguida a busca prossegue na sub-árvore esquerda depois na direita.

```
,
def procura_valor(v, r):
    if r:
        if(r.dado>v) return procura_valor(v, r.direita)
        if(r.dado<v) return procura_valor(v, r.esquerda)
        return r

def checa_arvore(n):
    def checa_sub(sub):
        if sub:
            return sub is procura_valor(sub_dado, n) and \
                   checa_sub(sub.esquerda) and checa_sub(direita)
        else:
            return True # Arvore vazia

    return checa_sub(n)
```

Esta solução faz N buscas na árvore, o que tem complexidade $\mathcal{O}(h)$, h sendo a profundidade do nó sendo buscado. No pior caso a complexidade é $\mathcal{O}(N^2)$ (Pode-se mostrar que para uma árvore razoavelmente balanceada a complexidade *média* é $\mathcal{O}(N \log N)$).

- *Validação por comparação de extremos*: Em uma árvore binária de busca, o nó seguinte ao atual *em ordem crescente de valor* é o nó mais a esquerda da sub-árvore direita. Do mesmo modo, o nó anterior é o nó mais a direita da sub-árvore esquerda.

  E possível implementar as funções `ext_direita` e `ext_esquerda`. Em seguida, verifica-se se o nó atual é estritamente maior do que o extremo direito da sub-árvore esquerda (se ela existe) e estritamente menor

que o extremo esquerdo da sub-árvore direita (se ela existe). Finalmente, o processo é aplicado a todos os nós da árvore. Aqui, novamente, optou-se por uma varredura em profundidade Anterior.

```
,
def ext_direita(n):
    while n.direita:
        n = n.direita
    return n.dado

def ext_esquerda(n):
    while n.esquerda
        n = n.esquerda
    return n.dado

def checa_arvore(n):
    if not n:
        return True # Arvore nula
    if n.esquerda and n.val <= ext_direita(n.esquerda):
        return False
    if n.direita and n.val >= ext_esquerda(n.direita):
        return False;
    return checa_arvore(n.esquerda) &&
            checa_arvore(n.direita);
```

A complexidade deste método depende da altura h do nó sendo visitado. Uma abordagem simplória sugere uma complexidade $\mathcal{O}(N^2)$, e, do ponto de vista estrito da notação *big Oh*, está *correta*. É possível no entanto obter um limite melhor superior. De fato, ao se considerar os níveis *verticais* de uma árvore binária, percebe-se que a adição de um novo nó aumenta o número de passos em apenas *dois* outros nós, no pior caso (os dois nós entre os quais ele é inserido, verticalmente). Como cada nó adicional em uma árvore aumenta a complexidade em um número *constante* de passos, a complexidade deste método é — talvez surpreendentemente — $\mathcal{O}(N)$.

- *Validação por enumeração interior* - 1: A enumeração *interior* dos nós de uma árvore binária de busca retorna uma *ordenação* dos seus valores. Basta então percorrê-los em ordem interior e verificar a ordem dos valores obtida. Há a dificuldade no entanto de comparar valores entre diferentes etapas da enumeração. Uma solução é adicionar os valores, durante a enumeração, a uma estrutura de dados auxiliar e posteriormente verificar se eles estão ordenados. O código-fonte de apoio possui uma classe `FilaCircular` que se presta a esse papel: Esta solução usa um *gerador* para produzir a sequência de elementos da árvore em ordem interior:

```
,
percorre_arvore_interna(n)
    if n:
        for n in percorre_arvore_interna(n.esquerda): yield n
        yield n.dado
        for n in percorre_arvore_interna(n.direita): yield n

def checa_arvore(n):
  saida = FilaCircular()
  for n in percorre_arvore_interna(n):
      saida.enqueue(n)
  if len(saida)==0:
      return True
  prev = saida.dequeue()
```

```
while len(saida):
    this = saida.dequeue()
    if prev >=x:
        return False
    prev = x
return True
```

Esta solução faz N passos de varredura da arvore e N passos de verificação de ordenação. Sua complexidade é $\mathcal{O}(N)$.

Exercício 21:

Para resolver, note que a ordem anterior de uma árvore sempre começa pela raiz, enquanto que a ordem interior tem a sub-árvore "esquerda" antes da raiz, e a sub-árvore "direita" depois da raiz. A figura 2 mostra a solução do exercício:

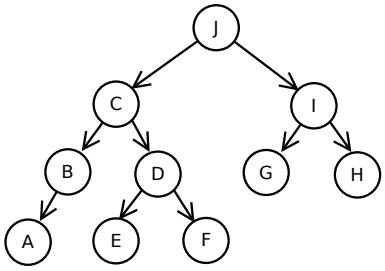


Figura 2: Solução do exercício 21

Exercício 23:

Item a): Esta demonstração usa propriedades conhecidas de árvores binárias de busca. A primeira parte realiza um número finito (N) inserções na árvore binária. Cada inserção pode ser feita em um tempo finito, logo a primeira parte é concluída em tempo finito. A segunda parte percorre a árvore na ordem interior, outra operação que leva tempo finito. Assim, todo o algoritmo é concluído em tempo finito. Finalmente, percorrer uma árvore binária de busca produz uma ordenação de seus elementos. Ora, estes elementos são elementos originais do vetor, assim, o algoritmo efetivamente produz uma ordenação dos elementos do vetor.

Item b): A complexidade é resultado da soma da complexidade das duas operações. A segunda parte é feita em tempo $\mathcal{O}(N)$, pois ela percorre cada nó da árvore, que tem no final a mesma quantidade de nós que o vetor original. A primeira parte depende da estrutura da árvore, e consequentemente da ordem original dos elementos do vetor. Uma operação de inserção em uma árvore leva no pior caso $\mathcal{O}(H)$, onde H é a altura da árvore. A árvore, no pior caso, tem uma altura que é $\mathcal{O}(n)$, onde n é o número de elementos já inseridos (por exemplo, uma árvore na qual todas as subárvores esquerdas estão ausentes). Assim, cada inserção tem complexidade $\mathcal{O}(n)$. A complexidade total é:

$$\sum_{n=1}^{N} \mathcal{O}(n) = \mathcal{O}(N^2)$$

Item b): Novamente, a complexidade é resultado da soma da complexidade das duas operações, e novamente a segunda parte pode ser feita em tempo $\mathcal{O}(N)$. A primeira parte, no entanto, tem uma complexidade diferente. De fato, agora a árvore tem no pior caso uma altura que é $\mathcal{O}(\log n)$, onde n é o número de elementos já inseridos. A complexidade total é:

$$\sum_{n=1}^{N} \mathcal{O}(\log n) = \mathcal{O}(N \log N)$$

Exercício 25

O processo de remoção consiste em substituir temporariamente o elemento da primeira posição com o da última (único removível sem movimentação no vetor). Em seguida, reparar o heap de cima para baixo (com a função FixHeapDown vista em sala).

Antes do reparo o heap é:

0 7 9 5 3 2 8 4 3 2 2

O reparo no nível, que troca o 0 pelo 9, produz o heap:

9 7 0 5 3 2 8 4 3 2 2

No nível 2 o reparo troca o 8 pelo 0:

9 7 8 5 3 2 0 4 3 2 2

Finalmente, não há mais descendentes do nó 0, e este é o resultado final.

Exercício 26:

Em um heap, o *menor* elemento necessariamente não possui descendentes. Assim, *qualquer* posição na qual ele não possui nenhum descendente é válida para ele.

Deste modo, a *primeira* posição válida para o menor elemento é imediatamente após o pai da *última* subárvore do heap.

Em uma linguagem com índices baseados em zero, os índices legais para um vetor de tamanho N vão de 0 a $N-1$. O índice do último elemento válido é assim $N-1$. O endereço de seu pai é $\lfloor (N-2)/2 \rfloor$. Finalmente, o endereço do *primeiro* elemento sem descendentes é $\lfloor (N-2)/2 \rfloor + 1$. Assim, o menor elemento pode estar em qualquer posição de $\lfloor (N-2)/2 \rfloor + 1$ a $N-1$.

O algoritmo varreria o espaço de $\lfloor (N-2)/2 \rfloor + 1$ a $N-1$ a busca do seu menor elemento. Não há como melhorar este algoritmo, visto que qualquer posição neste intervalo é válida para o menor elemento.

A complexidade é $\mathcal{O}(N/2)$ que é $\mathcal{O}(N)$.

Exercício 27

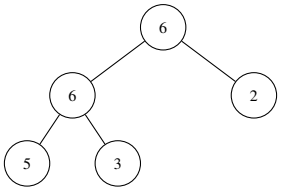
Sabe-se que o número total de nós na altura h do heap é de no máximo 2^{H-h}, onde H é a altura de *todo* o heap.

Ora, mas o número total de elementos de um heap de altura H é limitado superiormente por $2^H - 1$.

Assim, $\lceil N/2^h \rceil$ é o limite superior para elementos no heap com altura h.

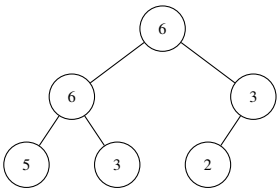
Exercício 28:

a)

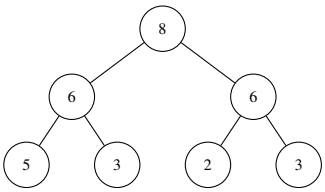


b)

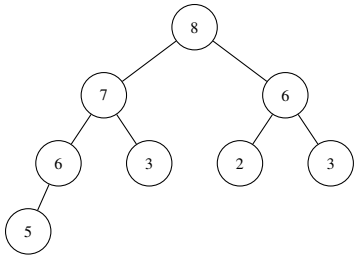
(a) Inserção de 3:



(b) Inserção de 8:

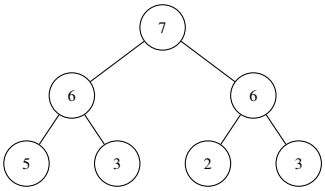


(c) Inserção de 7:

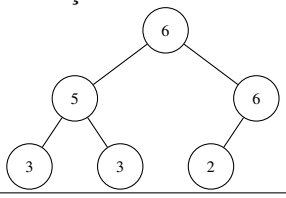


c)

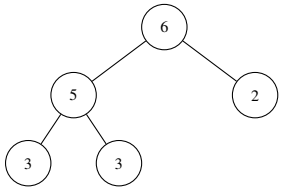
(a) Remoção de 8:



(b) Remoção de 7^1:



(c) Remoção de 6:



A correção do nó raiz pode seguir tanto pela esquerda quanto pela direita. Tomou-se aqui o caminho da esquerda. *Ambos estão corretos.*

Exercício 29:

O heapsort *não* é um algoritmo estável. Para demonstrá-lo, basta um contraexemplo (naturalmente, exemplos *não* são suficientes para demonstrar a estabilidade de um algoritmo). Tomemos o exemplo do enunciado, e consideremos o heap construído com a função FixHeapDown: Após a construção do heap o vetor é: `jhHeEhBbcb`. Após a ordenação o vetor é: `BbbcEehHhj`. Nota-se

que a ordem relativa dos elementos `b`, `b` e `B` é distinta da original (entre outras alterações).

Exercício 30:

A última posição no heap pode ser removida sem prejuízo da estrutura do mesmo. Deseja-se no entanto remover um elemento *arbitrário*. Ora, mas ao se trocar o último elemento de posição com o elemento que deseja-se substituir, gera-se um heap com um elemento potencialmente "errado", menor do que seus pais mas possivelmente menor que seus filhos, exatamente a solução que pode ser corrigida por `FixHeapDown`, que o faz na complexidade desejada. O código é:

```
def remove_from_heap(a, end, pos):
      temp = a[pos]
      a[pos] = a[end]
      a[end] = temp
      end -= 1
      fix_heap_down(a, pos, end)
      return end
```