Fundação Getulio Vargas
Estruturas de Dados e Algoritmos
Prof. Jorge Poco

Homework 2
Due: April 12, 2020

1 Questão 1

f_1 alternativa B : N
O tempo é linear pois são feitas n iterações cada uma com contribuição de $O(1)$.
f_2 alternativa E : N^3
f_2 chama f_1 dentro de um loop interno que para cada i ele itera i e loop externo vai de 0 até N.
f_3 alternativa H : $N!$
f_3 para cada valor indo de 0 até N a íesima iteração no loop retorna o valor da função f_1 para $N-1$, como fazemos isso N vezes, portanto o tempo de execução de f_3 é $N!$.
f_4 alternativa C : $nlogn$
$t(n) = 2t(\frac{n}{2}) + 3n$, $n = 2^k$
$t(2^k) = 2t(2^{k-1}) + 3.2^k$, utilizando o método da substituição, temos:
$s(k) = 2s(k-1) + 3.2^k$.
A equação característica é:
$(x-2)^2 = 0$
Sendo assim, $t(n) = nc_1 + c_2 nlogn$. O tempo de execução de f_4 é de $nlogn$.
f_6 alternativa G : 3^n
$f_6(n) = 3f_6(n-1)$. Pela equação característica, obtemos que o tempo computacional de f_6 é de 3^n.
f_7 alternativa A : $logn$
Supondo que n é uma potência de 2. Temos que $t(n) = t(\frac{n}{2}) + \Theta(1)$.
Utilizando uma árvore de recorrência, obtemos:
$t(n) = \Theta(1).logn + t(1)$

2 Questão 2

a) $T(n) = 2t(\frac{n}{2}) + n^k$, onde $k > 0$ é uma constante.
Usando o teorema mestre, temos que $a = b = 2$.
Assim, temos que $t(n) = O(nlogn)$ se $k = 1$, $t(n) = O(n^k)$ se $k > 1$ e $t(n) = O(n)$ se $k \in (0, k)$.
b)
$t(n) = 2t(\frac{n}{2}) + \frac{n}{logn}$. Suponhamos que n é uma potência de 2.
Desenhando a árvore, pra cada nível i temos um total de 2^i nós. Começamos no nível $i = 0$. Cada nível com exceção das chamadas de recursão tem uma contribuição de $2^i * \frac{\frac{n}{2^i}}{log\frac{n}{2^i}} = \frac{n}{logn - i}$.
Chegamos em $t(1)$ no último nível quando $1 = \frac{n}{2^i}, i = logn$. Assim, a altura da árvore é de logn + 1. No último nível temos uma quantidade $2^{logn} = n^{log2} = n$ de subproblemas, em que cada um contribui com um tempo constante c.

$$t(n) = \frac{n}{logn} + \frac{n}{log\frac{n}{2}} + ... + \frac{n}{log\frac{n}{2^{log(n-1)}}} + cn = n\sum_{i=0}^{logn-1} \frac{1}{logn - i} + cn$$

Como havíamos suposto que $n = 2^k$, onde $k = logn$, temos que: $t(n) = n\sum_{i=0}^{k-1} \frac{1}{k-i} + cn$
O somatório tem o comportamento de uma série harmônica, que para uma soma finita se aproxima de $log(k)$.
Logo, $t(n) = O(loglogn)$.

3 Questão 3

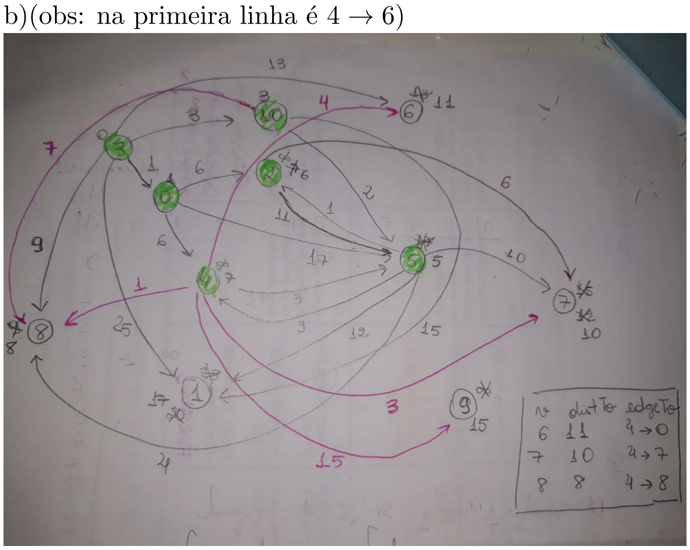
Queremos provar a afirmação: só há uma maneira de atribuir as chaves aos nós e formar uma BTS válida, recebidas n chaves e uma árvore binária com n nós.
Raciocínio usando indução:
Caso base: Recebo uma árvore binária com um nó e uma chave, de fato só existe uma maneira de atribuir a chave ao nó.
Hipótese de indução: Suponha que para uma dada árvore binária qualquer com k nós só haja uma maneira de atribuir as k chaves recebidas de modo que ela seja uma BTS válida.
Acrescentando um filho a uma folha nessa árvore de k nós e dada mais uma chave, removo os chaves uma vez atribuídas aos nós e conto quantos nós na árvore binária k + 1 estão à direita do root, suponha que seja r, e quantos estão à esquerda, l. Ordeno as chaves e adiciono a chave no root de acordo com o número correspondente que possui r maiores do que ele na lista e l menores. Observe que dada uma lista de números distintos não há como haver quantidades diferentes de valores maiores ou menores do que um número fixo.
Temos então duas sub árvores, uma à esquerda e uma à direita. Repetimos o processo pra os nós do próximo nível. Considere que como as subárvores possuem no máximo k nós e pela hipótese de indução só podemos atribuir as k chaves aos k nós de uma única forma, então temos que só existe uma maneira de distribuir as $k+1$ chaves para os $k+1$ nós como queríamos provar.

4 Questão 4

a) post order : $M, W, Y, I, P, S, E, B, O$
reverse post order: $O, B, E, S, P, I, Y, W, M$
b) A ordem BST começando por O é dada por:
$[[O], [B, E, Y], [S, W], [I, M, P]]$
c) Tal ordem não existe, pois ela só é possível se o grafo é um DAG, um grafo direcionado acíclico, o que não ocorre nesse caso pois há um ciclo entre os vértices E e S.

5 Questão 5

a) $3, 0, 10, 5, 2$
b)(obs: na primeira linha é $4 \rightarrow 6$)



6 Questão 6

H_1 é o conjunto exaustivo de todas as funções hash que mapeiam do universo U para os n buckets. Aqui considerando todas as possibilidades de mapeamento, temos um total de n^m funções possíveis. Assim, $|H_1| = n^m$. O conjunto exaustivo de todas as funções hash, apesar de ser uma família hash universal ocupa muitos bits para armazenar tantas funções. Para armazenar apenas uma função o custo é o $mlogn$, o que também é muito alto. Sendo assim, eu escolheria H_2, pois armazenar todo o H_1 ocupa muito espaço e armazenar cada função também ocupa muito espaço. Observe que o tamanho de H_2 é dado por $(p-1)p$, pois essas são as possíveis possibilidades para a e b. De fato, H_2 possui um tamanho muito menor.
Alternativas b e c.

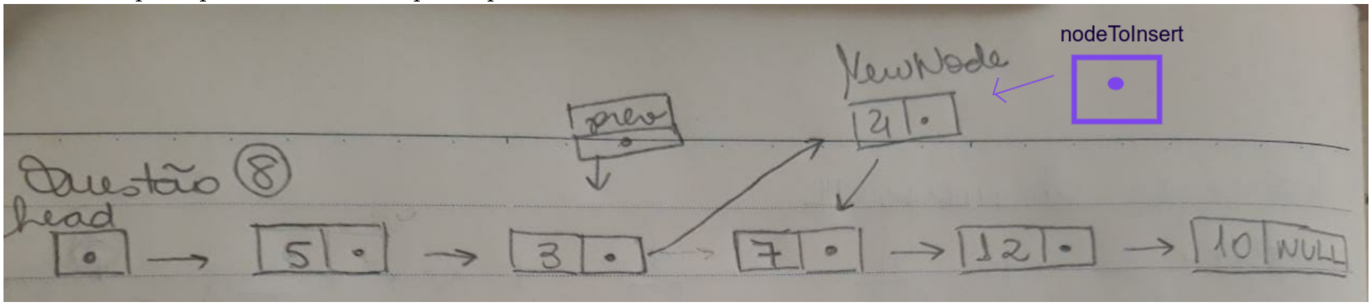
7 Questão 7

```
def encontrar_indice(A, left):
  // left controla o tamanho da lista existente na esquerda
  n = len(A)
  mid = math.floor(n/2)
  if len(A) == 0:
    return  print("Nenhum indice encontrado")
  if A[mid] == (mid + left):
    return print(A[mid])
  elif A[mid] > (mid + left):
  // Nao preciso procurar na lista da direita
    encontrar_indice(A[0: mid], left)
  elif A[mid] < (mid + left):
  // Preciso encontrar um modo de buscar na lista da direita preservando os indices
    left = left + mid
    encontrar_indice(A[mid:], left)
```

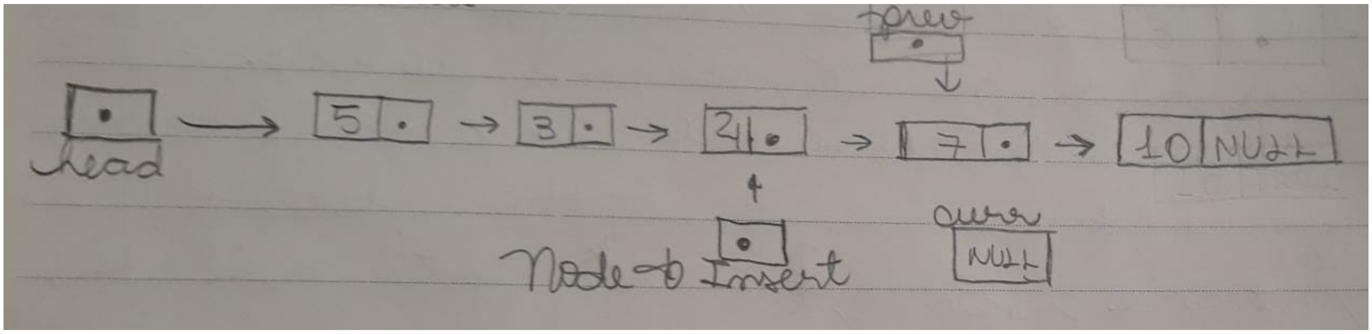
O algoritmo é $O(logn)$.

8 Questão 8

a) . Cria um ponteiro que aponta pra o nó 3.
. Cria um novo nó com item = 4 e cujo ponteiro aponta pra o nó 7.
. Faz com que o ponteiro do nó 3 aponte para o novo nó 4.



b) prev → item = 3
c) . Faz com que o ponteiro prev passe a apontar para o nó que o nó 3 aponta.
. Prev agora aponta para o nó 7.
. Cria um ponteiro que aponta para o nó 12.
. Faz 7 apontar para o nó que 12 aponta.
. deleta o nó para o qual curr aponta.
. Atribui null para o ponteiro curr.


prev
head
5
3
2
7
10
NULL
4
curr
NULL
Node to Insert