

# Homework 3

## 1. Análisis de Algoritmos [0.5]

Para cada função à esquerda, dê a melhor ordem correspondente de crescimento do tempo de execução à direita. Você pode usar uma resposta mais de uma vez ou não.

- |       |  |               |
|-------|--|---------------|
| --B-- | public static int f1(int N) {<br>int x = 0;<br>for (int i = 0; i < N; i++)<br>x++;<br>return x;<br>}                                       | A. $\log N$   |
| ----- | public static int f2(int N) {<br>int x = 0;<br>for (int i = 0; i < N; i++)<br>for (int j = 0; j < i; j++)<br>x += f1(j);<br>return x;<br>} | B. $N$        |
| ----- | public static int f3(int N) {<br>if (N == 0) return 1;<br>int x = 0;<br>for (int i = 0; i < N; i++)<br>x += f3(N-1);<br>return x;<br>}     | C. $N \log N$ |
| ----- | public static int f4(int N) {<br>if (N == 0) return 0;<br>return f4(N/2) + f1(N) + f1(N) + f1(N) + f4(N/2);<br>}                           | D. $N^2$      |
| ----- | public static int f6(int N) {<br>if (N == 0) return 1;<br>return f6(N-1) + f6(N-1) + f6(N-1);<br>}   | E. $N^3$      |
| ----- | public static int f7(int N) {<br>int x = 0;<br>while (N > 0) {<br>x++;<br>N = N / 2;<br>}<br>return x;<br>}                                | F. $2^N$      |
|       |  | G. $3^N$      |
|       |  | H. $N!$       |

## 2. Recursion [1.5]

Resolva as recorrências abaixo dando limites superiores apertados da forma  $T(n) = O(f(n))$  para uma função apropriada  $f$ . Você não precisa provar que os limites superiores são apertados ou fornecem limites inferiores ( $T(n) = \Omega(f(n))$ ). Você pode usar qualquer método da classe. Mostre seu trabalho. Se desejar, você pode assumir que  $n$  inicialmente tem o formato  $n = a^i$ , para uma constante apropriada  $a$ .

Nota: log refere-se à base de  $\log_2$ .

- a.  $T(n) = 2T(n/2) + n^k$  onde  $k > 0$  é uma constante.

[Dica: use o teorema mestre e considere casos dependendo do valor de  $k$ .]

- b.  $T(n) = 2T(n/2) + \frac{n}{\log n}$

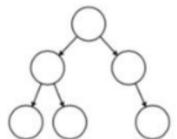
[Dica: desenhe a árvore de recursão.]

## 3. Binary Search Tree [1.5]

Você recebe uma árvore binária com  $n$  nós e um conjunto de  $n$  chaves distintas (números).

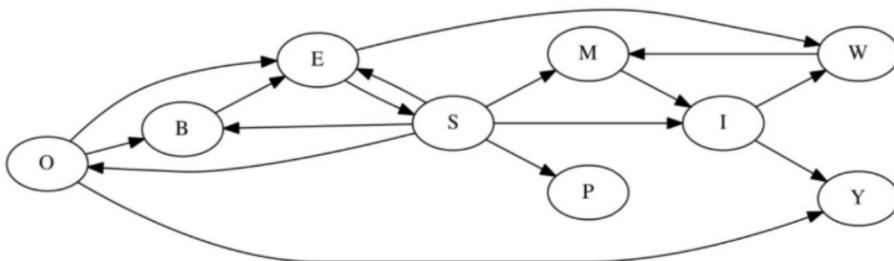
Prove ou refute: há exatamente uma maneira de atribuir as chaves aos nós, de modo que a árvore resultante é uma árvore de pesquisa binária válida (*binary search tree*).

**Exemplo:** Você recebe a árvore binária desenhada à direita e o conjunto de chaves  $\{1, 2, 3, 4, 5, 6\}$ . A questão pergunta se existe exatamente uma maneira de atribuir as chaves aos nós, de modo que a árvore seja uma árvore de pesquisa binária. (Se você provar a declaração, deve ser para qualquer entrada e não apenas este exemplo).  
[Dica: prove por indução]



## 4. Pesquisa en Grafos [1.5]

Considere o dígrafo abaixo. Suponha que as listas de adjacências estejam em ordem. Por exemplo, siga a aresta  $O \rightarrow B$  antes de seguir  $O \rightarrow E$ .



- Dê o postorder reverso do grafo quando visitado pelo DFS (*reverse postorder of the graph*). Comece pelo vértice  $O$ .
- Dê a ordem em que os vértices são enfileirados (“enqueued”) usando o BFS (de acordo com nossos slides, quando o nó é removido da lista  $L$ ). Comece pelo vértice  $O$ .
- Forneça um *topological sort* diferente do postorder reverso. Se não existe tal ordem, explique o porquê.

## 5. Dijkstra [1.5]

Suponha que você esteja executando o algoritmo de Dijkstra no dígrafo ponderado pela aresta (abaixo à esquerda), começando de um vértice fonte  $s$ . A tabela (abaixo à direita) fornece os valores  $\text{edgeTo}[]$  e  $\text{distTo}[]$  imediatamente após o vértice 2 ter sido excluído da fila de prioridade e relaxado.  
(Dica:  $\text{distTo}$  é o valor que cada nó possui, inicialmente é infinito e é atualizado em cada iteração. Cada vez que o  $\text{distTo}$  é atualizado, no  $\text{edgeTo}$  salvamos a aresta de onde atualizamos o peso, ou seja, o antecessor no caminho mais curto.)

<i>edge</i>	<i>weight</i>	<i>edge</i>	<i>weight</i>	<i>v</i>	<i>distTo[]</i>	<i>edgeTo[]</i>
$0 \rightarrow 2$	6.0	$5 \rightarrow 1$	12.0	0	1.0	$3 \rightarrow 0$
$0 \rightarrow 4$	6.0	$5 \rightarrow 2$	1.0	1	17.0	$5 \rightarrow 1$
$0 \rightarrow 5$	17.0	$5 \rightarrow 4$	3.0	2	6.0	$5 \rightarrow 2$
$1 \rightarrow 3$	17.0	$5 \rightarrow 7$	10.0	3	0.0	<i>null</i>
$2 \rightarrow 5$	11.0	$5 \rightarrow 8$	4.0	4	7.0	$0 \rightarrow 4$
$2 \rightarrow 7$	6.0	$6 \rightarrow 0$	12.0	5	5.0	$10 \rightarrow 5$
$3 \rightarrow 0$	1.0	$6 \rightarrow 1$	5.0	6	13.0	$3 \rightarrow 6$
$3 \rightarrow 10$	3.0	$6 \rightarrow 2$	1.0	7	12.0	$2 \rightarrow 7$
$3 \rightarrow 1$	25.0	$6 \rightarrow 4$	9.0	8	9.0	$3 \rightarrow 8$
$3 \rightarrow 6$	13.0	$6 \rightarrow 9$	4.0	9	$\infty$	<i>null</i>
$3 \rightarrow 8$	9.0	$7 \rightarrow 1$	7.0	10	3.0	$3 \rightarrow 10$
$4 \rightarrow 5$	3.0	$7 \rightarrow 5$	11.0			
$4 \rightarrow 6$	4.0	$7 \rightarrow 9$	6.0			
$4 \rightarrow 7$	3.0	$10 \rightarrow 1$	15.0			
$4 \rightarrow 8$	1.0	$10 \rightarrow 5$	2.0			
$4 \rightarrow 9$	15.0	$10 \rightarrow 8$	7.0			

- a. Dê a ordem na qual os 5 primeiros vértices foram excluídos da fila de prioridades e relaxados.
- b. Modifique a tabela (acima à direita) para mostrar os valores das matrizes  $\text{edgeTo}[]$  e  $\text{distTo}[]$  imediatamente após o próximo vértice ter sido excluído da fila de prioridades e relaxado. Circule esses valores que mudaram.

## 6. Algoritmos Aleatórios [0.5]

Seja  $U$  um universo de tamanho  $m$ , onde  $m$  é primo, e considere as seguintes duas famílias de hash que fazem hash de  $U$  em  $n$  baldes (*buckets*), onde  $n$  é muito menor que  $m$ . Primeiro, considere  $H_1$ , que é o conjunto de todas as funções de  $U$  para  $\{1, \dots, n\}$ :

$$H_1 = \{h \mid h : U \rightarrow \{1, \dots, n\}\}$$

Segundo, seja  $p = m$  (então  $p$  é primo, uma vez que assumimos que  $m$  seja primo), e escolha  $H_2$  para ser

$$H_2 = \{h_{a,b} \mid a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\}\},$$

onde  $h_{a,b} = ((ax + b) \bmod p) \bmod n$ . Você deseja implementar uma tabela de hash usando uma dessas duas famílias. Por que você escolheria  $H_2$  acima de  $H_1$  para essa tarefa?

- a.  $H_1$  não é uma família hash universal.
- b. Armazenar um elemento de  $H_1$  ocupa muito espaço.
- c. Armazenar todo o  $H_1$  ocupa muito espaço.

## 7. Desenho de Algoritmos [1.5]

Dada um vetor ordenado (sorted array)  $A$  de  $n$  inteiros distintos, alguns dos quais podem ser negativos, dê um algoritmo para encontrar um índice  $i$  tal que  $1 \leq i \leq n$  e  $A[i] = i$  desde que tal índice exista. Se houver muitos desses índices, o algoritmo pode retornar qualquer um deles.

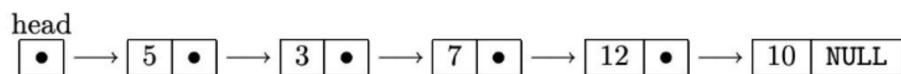
(Dica: pense em uma pesquisa binária)

## 8. Linked Lists [1.5]

Considere a lista com os seguintes membros privados:

```
class List{
    /* public members here ... */
    private:
        struct Node{
            ListDataType item; // the data of the node
            Node* next;       // points to the next node of the list
        };
        Node* head;          // point to first node in the list
};
```

Considere a *linked list* de ints representada pelo diagrama a seguir:



- a. Desenhe um diagrama da lista acima após as seguintes linhas de código terem sido executadas:

```
Node* prev = head->next;
Node* nodeToInsert = new Node;
nodeToInsert->item = 4;
nodeToInsert->next = prev->next;
prev->next = nodeToInsert;
```

- b. Suponha que o código representado acima na parte (a) tenha sido executado. Qual é o valor de `prev-> item`?
- c. Além do código acima, suponha que o código a seguir seja executado. Desenhe um diagrama da lista depois que esse código for executado também.

```
prev = prev->next;
prev = prev->next;
Node* curr = prev->next;
prev->next = curr->next;
delete curr;
curr = NULL;
```