UNIVERSIDADE DA BEIRA INTERIOR

Algoritmos e Estruturas de Dados **2º Semestre**

Exame Época Recurso (16 val) **2 h + 30 min** **2025/2026**

Nota: nas perguntas que impliquem a criação de código, os algoritmos que lhes estão associadas devem ser otimizados (executarem o menor número possível de operações/ações).

A. [0.75 val + 0.50 val] Análise de complexidade dos algoritmos

Considere o seguinte bloco de código em linguagem C:

```
...
for (k = 0; k < n; k++) {
   for (j = 1; j <= 3; j++)
      printf("OLA");
}
...
```

1. Simule o bloco de código dado (apresente apenas as 2 primeiras e as 2 últimas iterações de cada um dos ciclos), considerando que **n** é um número inteiro **muito grande**.

Apresente os dados da simulação numa tabela com a seguinte estrutura:

k	k < n [S/N]	j	j <= 3 [S/N]	printf [S/N]
0	0 < n [S]	...	...	...
...	...	...	...	...

2. A partir dos resultados da simulação obtidos em 1, determine a **ordem de complexidade** do algoritmo associado ao bloco de código dado (considerar apenas a operação dominante). Justifique.

B. [1.50 val] Algoritmos de ordenação e de pesquisa

Implementar uma **função** em C que dado um array **A** com **N** números **inteiros positivos** (pode haver repetidos) **ordenados** por **ordem crescente** (da primeira para a última posição), **devolva** a **quantidade** de elementos do array **A** com **valores iguais** ao **segundo menor valor** de **A**, caso exista.

C. [1.25 val + 2.00 val] Listas ligadas

Considere as seguintes declarações associadas a EAD Lista:

```
typedef int DadosLista;  // DadosLista = int
struct NodoLista {
   DadosLista Elemento;
   struct NodoLista *Prox;
};
typedef struct NodoLista *PNodoLista;
```

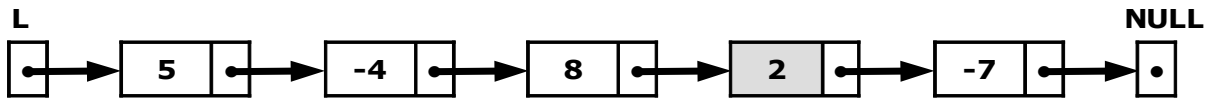
assim como os seguintes protótipos de funções já implementadas (operações básicas sobre a EAD Lista):

```
int listaVazia (PNodoLista L);  // devolve 1 se lista L está vazia e 0 caso contrário
PNodoLista libertarNodoLista (PNodoLista P);  // liberta a memória apontada por P e devolve NULL
```

Usando as definições e as funções dadas (protótipos), resolva as seguintes questões:

1. **Implemente** uma **função recursiva em C** que
   - **receba** uma lista **L** com elementos do tipo **DadosLista** (números inteiros) e um número inteiro **A**,
   - **devolva** a **soma** dos elementos de **L** que são **positivos não nulos** e **maiores que A**.
2. **Usando** ou **não** as funções fornecidas (em cima), mas **não podendo** usar outras funções, **implemente uma função iterativa (não recursiva) em C** que
   - **receba** uma lista **L** com elementos do tipo **DadosLista** (números inteiros),
   - **devolva** a lista **L** após **remover** o nodo **sucessor** do nodo com o **maior** elemento da lista **L**; se a lista **L** estiver vazia ou se o maior elemento não tem sucessor, então devolver a lista **L** inalterada.

Ex: é o nodo com **2** que será removido, visto ser o sucessor do nodo com **8** (**8** é o maior da lista).



D. [2.00 val] Pilhas

Considere as seguintes declarações associadas a EAD Pilha:

```
typedef  int  DadosPilha;     //  DadosPilha = int
struct NodoPilha { ... };
typedef struct NodoPilha *PNodoPilha;
```

assim como os seguintes protótipos de funções já implementadas (operações básicas sobre a EAD Pilha):

```
PNodoPilha criarPilha ();
int pilhaVazia (PNodoPilha S);
PNodoPilha push (DadosPilha X, PNodoPilha S);
PNodoPilha pop (PNodoPilha S);
DadosPilha topo (PNodoPilha S);
```

Usando as operações básicas sobre uma EAD Pilha, implemente uma **função em C** que

- **receba** uma **pilha S** com números **inteiros positivos não nulos** (**> 0**),
- **devolva** a **soma** de elementos da pilha **S** com **valores diferentes** dos elementos que **estão no topo e no fundo da pilha**, mas devolvendo a pilha **S** como no início; se S vazia, devolver **0**.

Ex: S = [3, 7, 2, 5, 2, 5], com topo(S) = 3 => resultados: **11** (7 + 2 + 2) e **S** = [3, 7, 2, 5, 2, 5]

E. [2.00 val] Árvores Binárias de Pesquisa (ABP)

Considere as seguintes declarações associadas de EAD Árvore Binária de Pesquisa:

```
typedef  int  DadosABP;   //  DadosABP = int
struct NodoABP {
   DadosABP  Elemento;
   struct  NodoABP  *Esquerda;
   struct  NodoABP  *Direita;
};
typedef  struct  NodoABP *PNodoABP;
```

Implemente **uma função recursiva** em C que

- **receba** uma ABP **T** com números **inteiros positivos** (**>= 0**) e um número **inteiro positivo X**,
- **devolva** o **maior** elemento de **T menor que X**; se não existir, então devolver **-1**.

F. [2.25 val] ABP Balanceadas/Equilibradas (AVL)

Equilibrar uma **ABP do tipo AVL**, consiste em aplicar uma das 4 operações de **rotação**: simples à esquerda, simples à direita, dupla à esquerda e dupla à direita.

Construa uma **ABP do tipo AVL** da seguinte forma:

a) **insira** os nodos com os seguintes valores (um de cada vez e pela ordem apresentada):

90, 30, 40, 70, 60 e 80 (**NOTA**: ao desenhar as ABP não é necessário colocar as alturas)

Sempre que **inserir um nodo**, deve-o **acrescentar** à ABP e **verificar se continua equilibrada**; caso **não fique equilibrada**, deve

1º) indicar o nodo onde se deteta o desequilíbrio e o tipo de rotação a aplicar para reequilibrá-la,

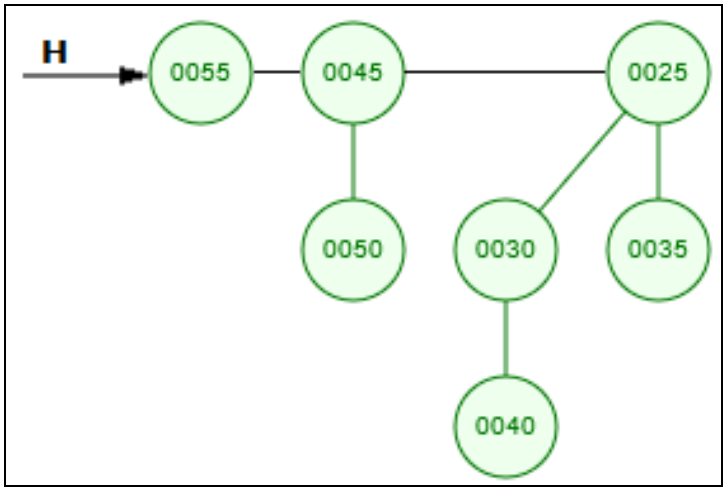
2º) reequilibrar a árvore e redesenhá-la (apresentando todos os passos efetuados)

b) **remova** a **raiz** da ABP obtida em **a)** e **redesenhe** a ABP obtida com a remoção, **verifique** se continua equilibrada e **reequilibre-a**, caso seja necessário. Apresente a árvore final.

NOTA: em caso de rotação dupla, apresentar (desenhar) as duas rotações que lhe estão associadas.

G. [1.50 val] Heaps (Filas de Prioridade)

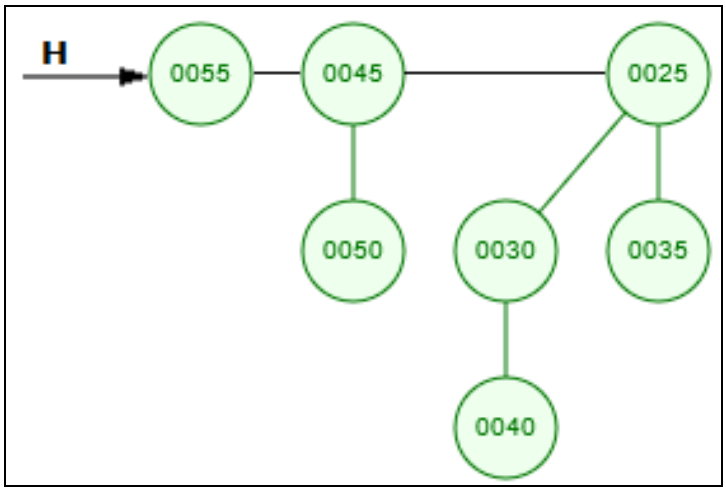
1. Considere o seguinte **heap Binomial H** com estrutura de **minHeap**:



Determine e **desenhe** o heap Binomial resultante da operação **"inserir"** um nodo com valor **10** no heap H. Justifique, apresentando a evolução do heap desde a sua estrutura inicial até à final.

NOTA: Sempre que redesenhar o heap Binomial, descreva o processo aplicado para o obter.

2. Considere o seguinte **heap Binomial H** com estrutura de **minHeap**:

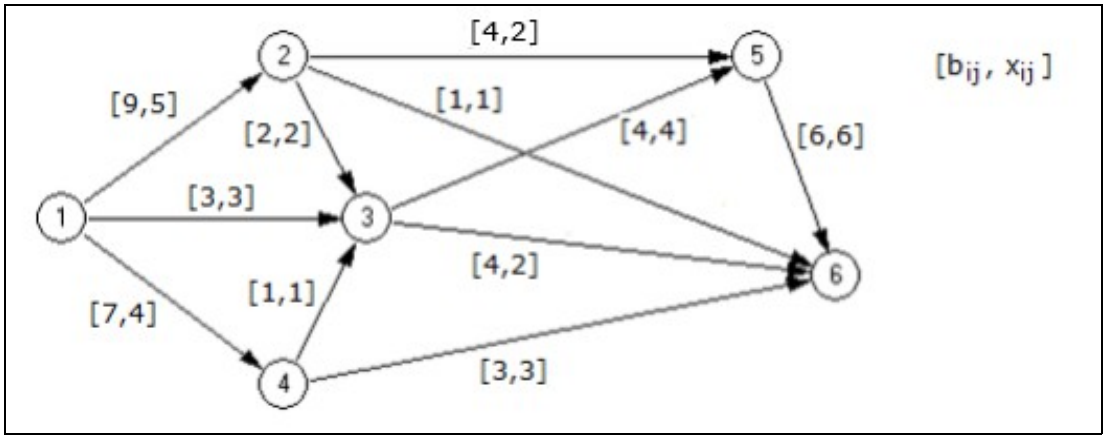


Determine e **desenhe** o heap Binomial resultante da operação **"remover"** aplicada ao heap H dado. Justifique, apresentando a evolução do heap desde a sua estrutura inicial até à final.

NOTA: Sempre que redesenhar o heap Binomial, descreva o processo aplicado para o obter.

H. [0.25 val + 2.00 val] Grafos

Considere a rede G = (N, A, B) em baixo que representa a quantidade de fluxo que passa pelos seus arcos desde o nó **1** até ao nó **6**: os valores apresentados em cada arco estão associados à capacidade do arco ($\mathbf{b_{ij}}$) e ao fluxo que passa atualmente nesse arco ($\mathbf{x_{ij}}$).



1. Determine os fluxos que atualmente **sai do nó 2** e o fluxo que **chega no nó 3**. Justifique.
2. Determine o **fluxo máximo** que pode ser enviado do nó **1** para o nó **6**, simulando o algoritmo de **Ford-Fulkerson**. Apresente os dados da simulação: todos os valores das estruturas de dados **R**, **Nós rotulados não varridos** e **Nós rotulados varridos**, e das variáveis **j** e **k**.

   Esquema para apresentar os dados da simulação:

	1	2	3	4	5	6
R						

Nós **rotulados não varridos** ←

Nós **rotulados varridos** ←

j ← ?

k ← …

k ← …

Algoritmo de Ford-Fulkerson:

Passo 1.
R(S) ← [S^+, ∞] (S é rotulado com S^+ e ∞)
S fica rotulado e não varrido

Passo 2. (processo de rotulação)
j (rotulado e não varrido) ← [i^+, Q(j)] ou [i^-, Q(j)]
para (todo **k** ∈ **N** não rotulado, tal que (j, k) ∈ A e $x_{jk} < b_{jk}$) **repetir**
R(k) ← [j^+, Q(k)], com Q(k) = min{ Q(j), $b_{jk} - x_{jk}$ }
fim_para
para (todo **k** ∈ **N** não rotulado, tal que (k, j) ∈ A e $x_{kj} > 0$) **repetir**
R(k) ← [j^-, Q(k)], com Q(k) = min{ Q(j), x_{kj} }
fim_para
j fica **rotulado** e **varrido**
todos os nós **k** ficam **rotulados** e **não varridos**
se (T está rotulado) **ou** (não é possível rotular T) **então**
(foi determinado um c.a.f. ⇒ **passar** ao **Passo 3**) **ou**
(não existe c.a.f. — o fluxo atual é máximo ⇒ **PARAR**)
senão
regressar ao **Passo 2** (início)
fim_se

Passo 3. (mudança de fluxo)
como (R(T) = [k^+, Q(T)]) **então**
x_{kT} ← x_{kT} + Q(T)
enquanto (k ≠ S) **repetir**
se (R(k) = [j^+, Q(k)]) **então**
x_{jk} ← x_{jk} + Q(T)
fim_se
se (R(k) = [j^-, Q(k)]) **então**
x_{kj} ← x_{kj} – Q(T)
fim_se
k ← j
fim_enquanto
apagar os rótulos
regressar ao **Passo 1**