1. Especificação (3 valores)

A função `lastSeen` calcula a última posição de um array onde um dado elemento ocorre. No caso do elemento não existir no array, a função retorna (-1).

```
int lastSeen (int x, int v[], int N){
    // Pre: N >= 0
    int i= N-1;
    while (i>=0 && v[i] != x)
        // Inv:
        i=i-1;
    // Pos: ???
    return i;
}
```

1. Complete a especificação da função de forma a refletir a descrição acima (bem como a implementação apresentada).
2. Defina um invariante apropriado para provar a correcção parcial desta função.

2. Complexidade de algoritmos recursivos (3 valores)

A função `diffConseqR` calcula o comprimento do maior sub-array sem elementos repetidos. Para isso usa a função `lastSeen` que calcula a última posição de um array onde um dado elemento ocorre (no caso do elemento não existir no array, a função retorna (-1)).

```
int diffConseqR (int v[], int N){
      int p;
      if (N<2) return N;
      p = lastSeen (v[N-1],v,N-1);
      return max(1+diffConseqR (v+p+1, N-p-2),
                    diffConseqR(v,N-1));
}
```

Escreva e resolva uma recorrência que traduza o número de acessos ao array v no caso em que o array não tem elementos repetidos (e por isso as invocações da função `lastSeen` retornam sempre -1).

Número: 104098 Nome: Eduarda Rafaela Martins Vieira

3. Complexidade de algoritmos iterativos (3 valores)

A função `diffConseq` calcula o comprimento do maior sub-array sem elementos repetidos.
Com o objectivo de analisar a complexidade desta função em termos de número de acessos ao array.

0,5 1. Identifique o melhor e pior casos da execução desta função.
1 2. Para o pior caso identificado, calcule o número de acessos ao array.

```
int diffConseq (int v[], int N){
      int r=0,c=1,i;
      for (i=1; i<N; i++){
            c = min (1+c, i-lastSeen(v[i], v, i));
            r = max (r, c);
      }
      return r;
}
```

4. Min-Heap (3 valores)

Assumindo que existem funções
- `void bubbleUp (int h[], int i)` que faz o bubble-up do elemento da posição i
- `void bubbleDown (int h[], int N, int i)` que faz o bubble-down do elemento da posição i

Defina a função `void heapify (int v[], int N)` que transforma o array v numa *min-heap*.
Qual a complexidade da função apresentada.

4.

```
void heapify (int v[], int N){

    int i;
    for (i=(N-2)/2 ; i>=0 ; i--){
```

5. Estruturas de Dados (5 valores)

MinHeap Considere que se usa um array dinâmico para armazenar uma *min-heap*

```
typedef struct dynArr {
    int size, used;
    int *values;
} DynArr;
```

No qual os campos `size` e `used` correspondem ao tamanho do array `values` e ao tamanho da *min-heap*, respectivamente. Considere ainda a min-heap com `size=12`, `used=10` e em que os primeiros 10 elementos do array `values` têm os seguintes valores:

0	1	2	3	4	5	6	7	8	9	10	11
10	20	50	30	60	70	100	40	80	90	...	...
										25	

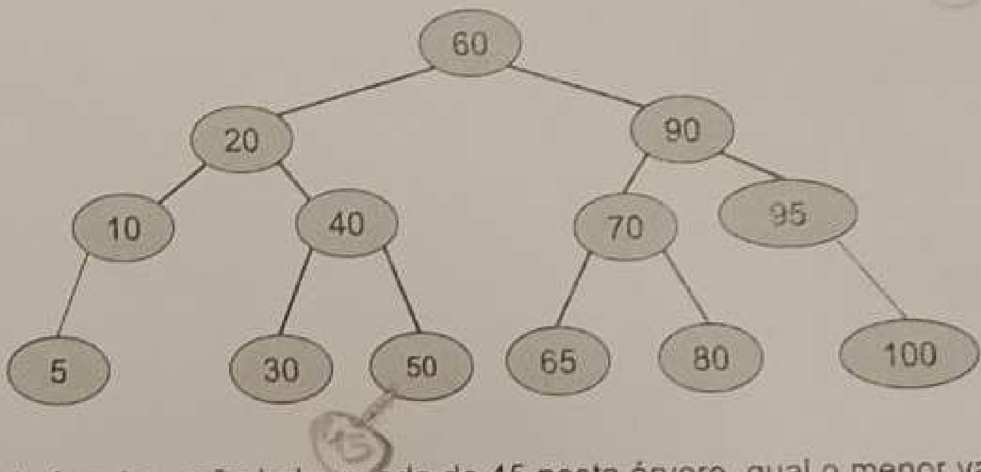
1. Considere que se acrescenta um novo (i.e. diferente de todos os que lá estão) elemento x a esta *min-heap*, em que (`0 < x < 100`). Qual o número **médio** de *swaps* que serão feitos.
   Resposta:
2. De forma a fazer análise amortizada do custo (escritas no array) da inserção, definiu-se como função de potencial `phi (x) = 2*x.used - x.size`.
   Qual o custo amortizado de, na tabela apresentada, acrescentar o valor 25
   (considere que cada swap tem um custo de 2).
   Resposta:

Tabelas de Hash Considere uma tabela de hash de inteiros (`hsize=10, hash (x) = x%10`), implementada em open-address com linear probing.
Considere ainda que a tabela tem o seguinte conteúdo

0	1	2	3	4	5	6	7	8	9
U	F	U	U	D	U	U	F	F	U
20	74	42	23	14	52	36	17	12	19

1. A inserção do valor 35 será feita em que posição?
   Resposta:
2. A consulta do valor 74 quantas posições da tabela consulta?
   Resposta:

Árvores AVL Considere a árvore balanceada abaixo.



Após a inserção balanceada de 45 nesta árvore, qual o menor valor dos nodos do nível 2 (assuma que a raiz se encontra no nível 1)
Resposta:

Grafos Seja g um grafo não orientado, não pesado e ligado com 15 arestas. A invocação `breadth_first (g,0,vis,pais)` preenche o array `pais` com os seguintes valores

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
pais =	-1	5	6	8	0	6	7	11	4	6	6	13	13	8	13

Qual a distância (número mínimo de arestas) do caminho entre os vértices 0 e 7?
Resposta:

6. Grafos (3 valores)

Considere o tipo habitual para representar árvores binárias

```
typedef struct nodo {int valor; struct nodo *esq, *dir;} *ABin;
```

Adapte o algoritmo de travessia *breadth-first* de forma a definir a função `int porNivel (ABin a, int v[], int N)` que preenche o array `v` com os elementos da árvore `a`, por níveis. A função deverá preencher um máximo de `N` elementos e retornar o número de elementos preenchidos.

Algoritmos C.
Recurso 2026

1.1

// pos: $i == -1$ && $\forall_{0<j<N}$ $v[j] != x$

// $0 <= i$ && $i < N$ && $v[i] == x$ && $\forall_{i<j<N}$ $v[j] != x$

1.2.

$\forall_{i<j<N}$ $v[j] != x$

2.

$$T(N) = \begin{cases} 0 & \text{se } N=0 \\ (N-1) + 2 \times T(N-1) & \text{, se } N>0 \end{cases}$$

$T(1) = 0$

$T(2) = 1$

$T(3) = 4$

$T(4) = 3 + 2\times 4 = 11$

$T(5) = 26$

$T(N) = 2^N - N - 1$

3.1

Pior caso: Não existe valores repetidos. A função lastrun percorre todo o array e faz i acessos ao array.

Melhor caso: encontra o elemento na 1ª comparação. (o melhor caso não está 100% certo)

3.2

acessos fora lastrun: $v[i]$ (1x) → faz $(N-1)$ vezes

acessos dentro lastrun: $\sum_{i=1}^{N-1} i = \frac{(N-1)\times N}{2}$

Nº total de acessos: $T(N) = (N-1) + (N^2 - N)/2$

Complexidade: $\Theta(N^2)$

4.

```
void heapify (int v[], int N) {
int i;
for( i = (N-2)/2 ; i>=0 ; i--) {
   bubbleDown (v, N, i);
   }
}
```

complexidade: $\Theta(N)$

5.1

pai: $(i-1)/2$

$100-60=40$ ~~[illegible]~~ $(x>60) \rightarrow 0$ swaps (fica no $i=10$)

$60-20=40$ $(20<x<60) \rightarrow 1$ swap ($i=4$) vai para

$20-10=10$ $(10<x<20) \rightarrow 2$ swaps ($i=1$)

$10-0=10$ $(x<10) \rightarrow 3$ swaps ($i=0$)

$$\frac{40\times 0 + 40\times 1 + 10\times 2 + 10\times 3}{100} = 0{,}9$$

5.2

size = 12

used $i_{inicio} = 10$

$phi(x)_{inicial} = 2\times 10 - 12 = 8$

$phi(x)_{final} = 2\times 11 - 12 = 10$

$$c_{real} = \underset{\text{inicio}}{25} + \underset{\text{custo}}{2} = 27$$

$c_{amortizado} = 27 + ((10-8)) = 29$ $\left(c_r + (phi_f - phi_i)\right)$

tabela hash

linear probing: $h(x,i) = h'(x) + i \bmod n$

1. M = 10 elementos

35 mod 10 = 5 × Used (U)

36 mod 10 = 6 × Used

37 mod 10 = 7 ✓ free (F)

35 será posto na posição 7.

2.

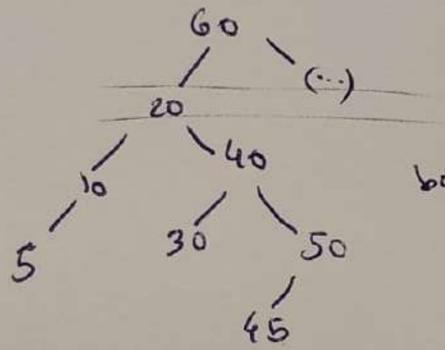
[illegible]

74 mod 10 = 4

4 → 5 → 6 → 7 → 8 → 9 → 0 → 1   8 posições

AVL

1. 20



balanceada na mesma

Grafos

7 (11)
↳ 11 (13)
↳ 13 (8)
↳ 8 (4)
↳ 4 (0)

$0 \xrightarrow{1} 4 \xrightarrow{2} 8 \xrightarrow{3} 13 \xrightarrow{4} 11 \xrightarrow{5} 7$

distância 5 arestas

```c
typedef struct nodo {
    int valor;
    struct nodo *esq, *dir;
} *ABin;

int porNivel(ABin a, int v[], int N) {
    if (a == NULL || N <= 0) return 0;

    int preenchidos = 0;

    // Criamos uma fila auxiliar para armazenar ponteiros de nós
    // No pior caso (árvore muito larga), a fila pode precisar de espaço significativo
    ABin *fila = malloc(sizeof(struct nodo*) * N * 2);
    int frente = 0, tras = 0;

    // Colocamos a raiz na fila
    fila[tras++] = a;

    while (frente < tras && preenchidos < N) {
        // Retira o nó da frente da fila
        ABin atual = fila[frente++];

        // Guarda o valor no array de destino
        v[preenchidos++] = atual->valor;

        // Adiciona os filhos à fila se eles existirem
        if (atual->esq != NULL) {
            fila[tras++] = atual->esq;
        }
        if (atual->dir != NULL) {
            fila[tras++] = atual->dir;
        }
    }

    free(fila);
    return preenchidos;
}
```