# Métodos de Programação II
# 2º Ano - LESI
# Exame da 1ª Chamada - Resolução

Edgar Sousa

12 de Julho de 2007

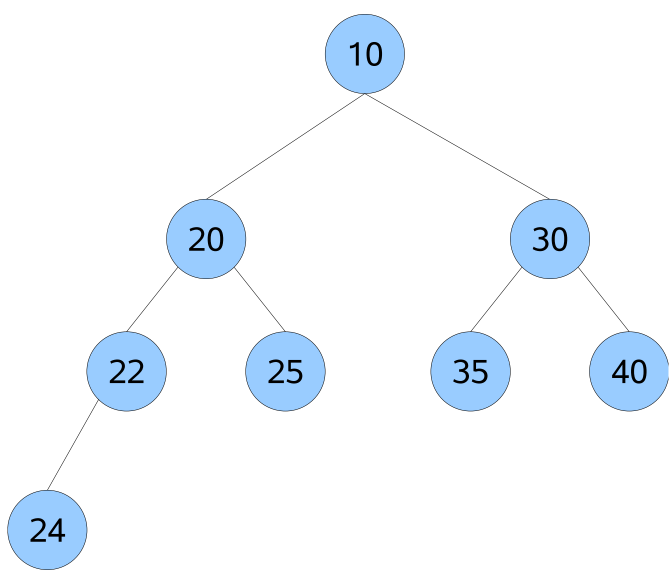
## Parte I

1. *Relembre a noção de* min-heap *binária.*

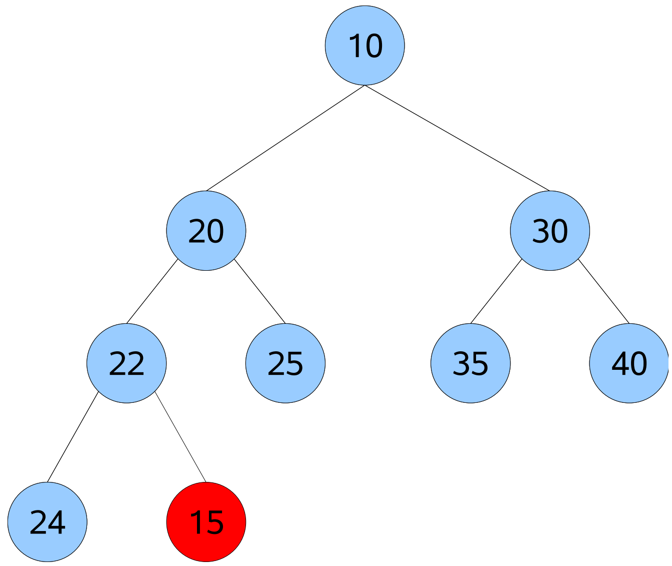
   (a) *Descreva, por palavras suas, as propriedades que esta estrutura de dados deve ter.*
   Uma *min-heap* binária é um tipo especial de árvore onde o valor em cada nodo é menor do que os valores dos filhos, e estes também observam esta propriedade.

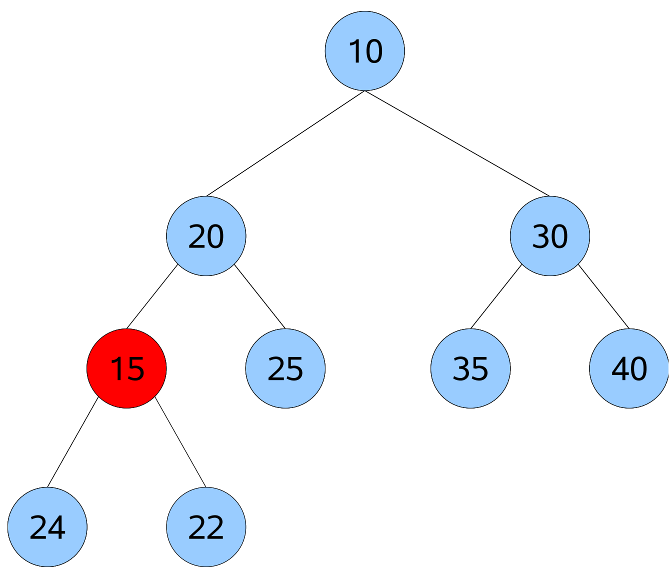
   (b) *Desenhe as três* heaps *de inteiros que resultam de:*

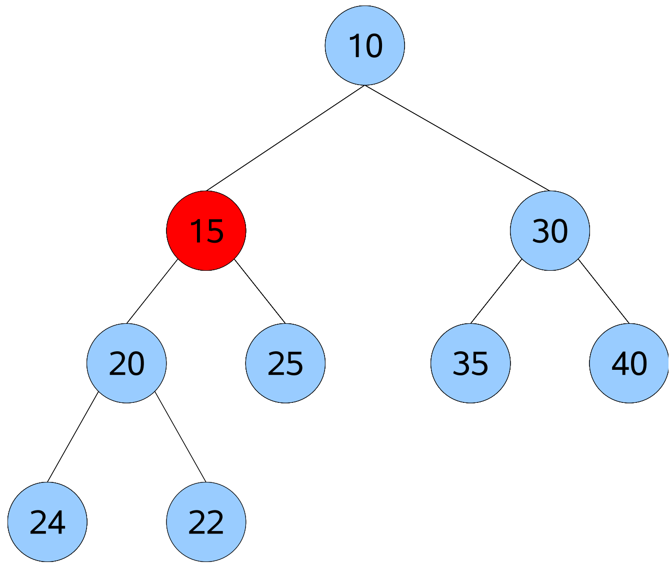
      i. *inserir consecutivamente os números* 10, 20, 30, 22, 25, 35, 40, 24



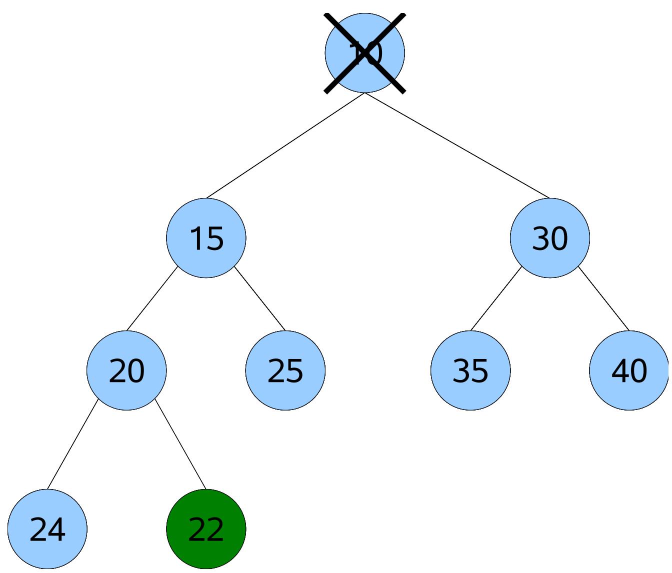

      ii. *inserir o número* 15 *na* heap *resultante;*

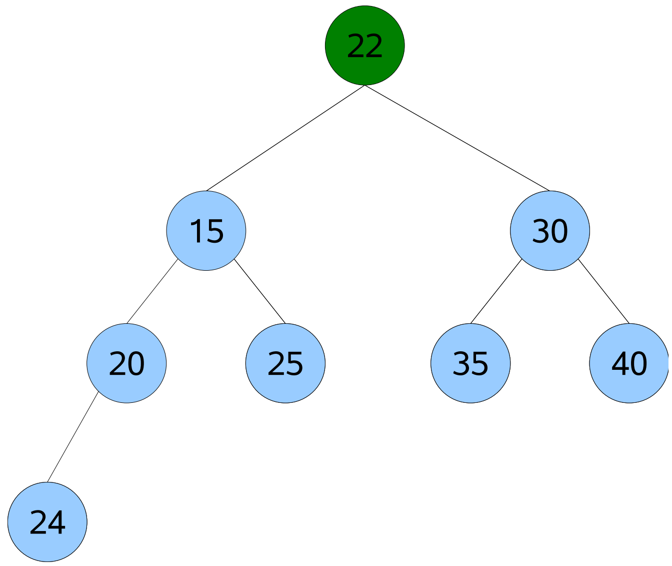

10
20
30
22
25
35
40
24
15

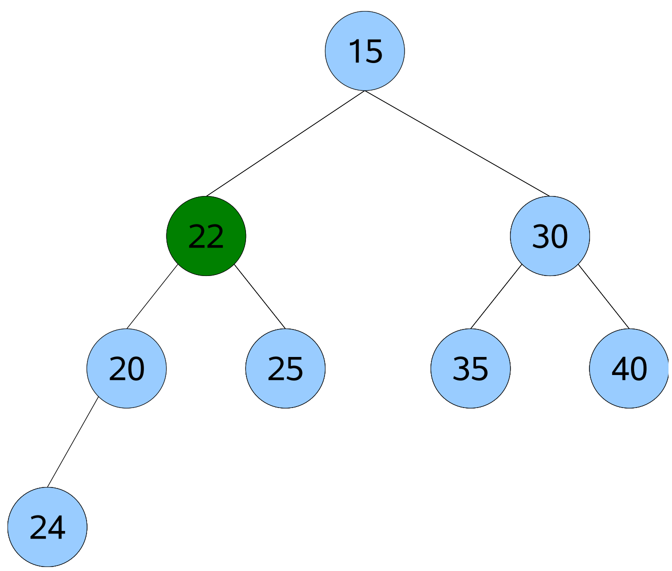

10
20
30
15
25
35
40
24
22

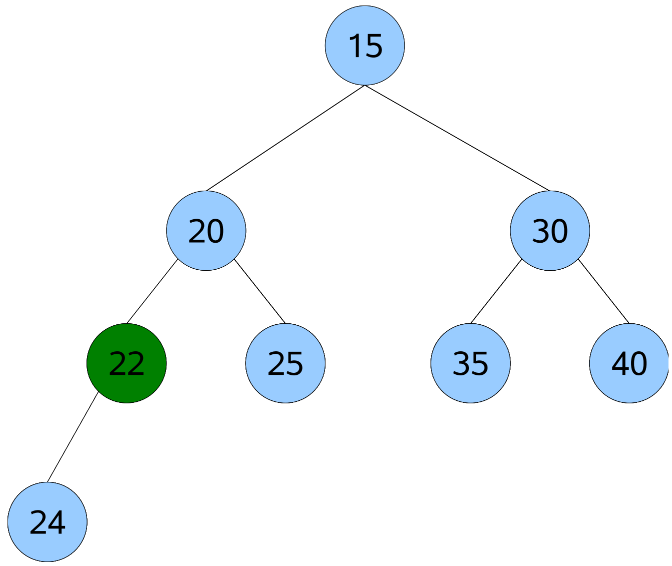



iii. *remover o mínimo da* heap




22
15
30
20
25
35
40
24


15
22
30
20
25
35
40
24




2. *Considere o seguinte programa:*

```
while(e<n){
    y=y*x;
    e=e+1;
}
```

*Prove que este ciclo termina e que preserva o seguinte invariante:* $y = x^e \wedge e \leq n$

Variante: $V = n - e$. É esta a quantidade que vai diminuindo de iteração em iteração do ciclo

$$\begin{array}{rcl}
\{I \wedge C_{ciclo} \wedge V = v_0\} & S_{ciclo} & \{I \wedge V < v_0\} \\
\{y = x^e \wedge e \leq n \wedge e < n \wedge V = v_0\} & y = y * x;\ e = e + 1 & \{y = x^e \wedge e \leq n \wedge V < v_0\} \\
\{y = x^e \wedge e \leq n \wedge e < n \wedge V = v_0\} & \Rightarrow & \{y = x^e \wedge e \leq n \wedge n - e < v_0\}[y \setminus y * x,\ e \setminus e + 1] \\
\{y = x^e \wedge e \leq n \wedge e < n \wedge V = v_0\} & \Rightarrow & \{y * x = x^{e+1} \wedge e + 1 \leq n \wedge n - (e + 1) < v_0\} \\
\{y = x^e \wedge e \leq n \wedge e < n \wedge V = v_0\} & \Rightarrow & \{y * x = x^e * x \wedge e < n \wedge n - e - 1 < v_0\} \\
\{y = x^e \wedge e \leq n \wedge e < n \wedge V = v_0\} & \Rightarrow & \{y = x^e \wedge e < n \wedge n - e - 1 < v_0\}
\end{array}$$

As duas primeiras condições são imediatas. No caso da terceira, temos que ter em conta que $v_0 = n - e$ e à qual se retirarmos uma unidade diminui. Prova-se assim que o invariante é preservado e que de facto o ciclo termina.

3. *Com base na seguinte implementação de uma árvore de procura, defina a função* `treeToArray` *que preenche um array com os elementos da árvore de procura, ordenados por ordem crescente. Note que esta função tem ainda um parâmetro (de entrada e saída) que indica qual a primeira posição livre do array.*

```
typedef struct Node {
    int elem;
    struct Node *esq, *dir;
} *Tree;

void treeToArray(Tree t, int A[], int *i);
```

```
void treeToArray(Tree t, int A[], int *i){
    if(t!=NULL) {
        //recursivamente passar para o array o lado esquerdo da árvore
        treeToArray(t->esq,A,i);
        //processar o elemento actual
        A[*i]=t->elem;
        *i+=1;
        //agora por fim processar o lado direito da árvore
        treeToArray(t->dir,A,i);
    }
    //travessia in-order: lado esquerdo - raiz - lado direito
}
```

4. *Considere a função* maxSort *que faz a ordenação de um array de tamanho* n.

```
void maxSort(int A[], int n){
    int i,j;

    for (i=n-1; i>0; i--)
        for (j=0; j<i; j++)
          if (A[j] > A[i])
            swap(A,j,i);
}
```

Atribuindo um tempo constante $C_1$ para atribuições, $C_2$ para comparações e $C_3$ para a função *swap* temos o seguinte:

$$T(n) = C_1 + \sum_{i=1}^{n-1}(C_2 + C_1 + C_1 + \sum_{j=0}^{i-1}(C_2 + C_1 + C_2 + C_3))$$

O primeiro somatório representa o ciclo *for* exterior e o segundo representa o *for* interior. Fazendo uma mudança de variável (para simplificar os cálculos) $m = n - 1$ podemos ter:

$$T(m) = C_1 + \sum_{i=0}^{m-1}(C_2 + 2C_1 + \sum_{j=0}^{i-1}(2C_2 + C_1 + C_3)) = C_1 + m(C_2 + 2C_1) + \frac{m(m+1)}{2}(2C_2 + C_1 + C_3)$$

$T(n) = C_1 + (n+1)(C_2 + 2C_1) + \dfrac{(n+1)^2 + (n+1)}{2}(2C_2 + C_1 + C_3)$. Convertendo as constantes $C_i$ para outras, simplifica a expressão e fica:

$T(n) = K_1 n^2 + K_2 n + K_3$. Logo podemos dizer que a função executa no pior caso em tempo $O(n^2)$.

5. *Indique, justificando, a solução da seguinte recorrência:*

$$T(n) = \begin{cases} c & se \quad n \leq 1 \\ 4T(n/2) & se \quad n > 1 \end{cases}$$

N.º de níveis da árvore: $\dfrac{n}{2^k} = 1 \Leftrightarrow n = 2^k \Leftrightarrow k = \log_2 n$ ou seja, níveis desde 0 até $(\log_2 n) - 1$.

N.º de nodos em cada nível: começa com um e quadriplica em cada nível: $4^i$, com $i$ a começar em 0.

Tempo de EXTRA de execução de cada nodo: 0. Tempo de execução das FOLHAS: $c$

$$T(n) = tempo\_nodos + tempo\_folhas = \left(\sum_{i=0}^{(\log_2 n)-2} 4^i \times 0\right) + 4^{(\log_2 n)-1} \times c = 4^{(\log_2 n)-1} \times c$$

6. *Assuma que a função* ssSP *faz o cálculo dos caminhos mais curtos com origem num dado vértice, segundo o algoritmo de Dijkstra num grafo pesado.*

```
void ssSP(Grafo g, int v, int pai[], int dist[]);
```

*Note que, dados o grafo* **g** *e o vértice de origem* **v***, esta função guarda a árvore dos caminhos mais curtos em* **pai***, e as respectivas distâncias em* **dist***.*

*Usando a função* **ssSP***, defina a função* **camMaisCurto** *que dado o grafo* **g***, o vértice origem* **v** *e o vértice destino* **d***, imprima no* **stdout** *a sequência de vértices do caminho mais curto (da origem para o destino), um vértice por linha. Esta função devolve* **1** *se não houver caminho, e* **0** *se existir.*

```
int camMaisCurto (Grafo g, int v, int d);
```

```
int camMaisCurto (Grafo g, int v, int d){
    int *pai  = malloc(sizeof(int)*g->num_nodos);
    int *dist = malloc(sizeof(int)*g->num_nodos);

    ssSP(g,v,pai,dist);

    Stack s = new_stack();
    while(d!=v){
        push(s,d);
        d=pai[d];
    }
    if(d==v){
        push(s,d);
        while(!empty_stack(s)){
          printf("%d\n",pop(s));
        }
        return 0;
    }
    return 1;
}
```