UNIVERSIDADE DA BEIRA INTERIOR

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

Exame Época Normal **Resolução** **2024/2025**

A. Análise de complexidade dos algoritmos

1.

k	k < n (S/N)	j	j <= k (S/N)	printf
(k=1) **1**	1 < n (S)	(j=1) **1**	1 <= 1 (S)	OLA
		(j++) **2**	2 <= 1 (N)	
(k++) **2**	2 < n (S)	(j=1) **1**	1 <= 2 (S)	OLA
		(j++) **2**	2 <= 2 (S)	OLA
		(j++) **3**	3 <= 2 (N)	
...	...	...	...	...
(k++) **n-1**	n-1 < n (S)	(j=1) **1**	1 <= n-1 (S)	OLA
		(j++) **2**	2 <= n-1 (S)	OLA
		...	...	...
		(j++) **n-1**	n-1 <= n-1 (S)	OLA
		(j++) **n**	n <= n-1 (N)	
(k++) **n**	n < n (N)			

2.

Para determinar a ordem de complexidade do algoritmo, basta contabilizar o número de vezes que a operação dominante é executada. Analisando a tabela de simulação, conclui-se que a operação dominante é a comparação "j <= k".

T(n) = **2** [k=1] + **3** [k=2] + ... + **n** [k=n-1] + **0** [k=n] =

2 + **3** + ... + **n** [progressão aritmética com n-1 termos] =

((**2** + **n**) / 2) x (**n-1**) = (1 + n / 2) x (n - 1) = n - 1 + (n^2 / 2) – (n / 2) =

(n^2 / 2) + (n / 2) - 1

Logo, a ordem de complexidade é: **O(n^2)** (ou: **T(n) = O(n^2)**)

Podia-se também contabilizar todas as operações, somando o número de vezes que cada uma das operações é executada.

B. Listas ligadas

1.

```
int numeroElementosEntreXeY (PNodoLista L, int X, int Y) {   // int = INFOLista
   int  cont;
   if (L == NULL)  // caso base/terminal
      return  0;
   cont = numeroElementosEntreXeY(L->Prox, X, Y);   // caso geral
   if (L->Elemento >= X  &&  L->Elemento <= Y)
      return  cont + 1;
   else
      return  cont;
}
```

2.

```
PNodoLista  removerMaiorElemento  (PNodoLista  L) {
   int  maior;
   PNodoLista  PAntMaior, P, PAnt;
   if (L == NULL)
      return  L;
   maior = L->Elemento;    // o 1º maior é o que está na cabeça da lista
   PAntMaior = NULL;       // ponteiro para o antecessor do maior elemento (nodo a remover)
   P = L->Prox;            // ponteiro para percorrer todos os nodos da lista
   PAnt = L;               // ponteiro para o nodo anterior ao nodo apontado por P
   while (P != NULL) {
      if (P->Elemento > maior) {
         maior = P->Elemento;
         PAntMaior = PAnt;
      }
      PAnt = P;
      P = P->Prox;
   }
   if (PAntMaior == NULL) {   // o nodo a remover é a cabeça da lista
      P = L;
      L = L->Prox;
   }
   else {   // o nodo a remover não é a cabeça da lista
      P = PAntMaior->Prox;
      PAntMaior->Prox = P->Prox;
   }
   P = libertarNodoLista(P);
   return  L;
}
```

C. Pilhas

```
int numeroElementoIguaisFundo (PNodoPilha *S)   // int = INFOPilha
{
   int  fundo, cont;
   PNodoPilha  SAux;
   if (pilhaVazia(*S) == 0)
      return 0;
   SAux = criarPilha();
   while (pilhaVazia(*S) == 0) {
      SAux = push(topo(*S), SAux);
      *S = pop(*S);
   }
   cont = 0;
   fundo = topo(SAux);
   while (pilhaVazia(SAux) == 0) {
      if (topo(SAux) == fundo)
         cont = cont + 1;
      *S = push(topo(SAux), *S);
      SAux = pop(SAux);
   }
   return cont;
}
```

D. Árvores Binárias de Pesquisa (ABP)

1.

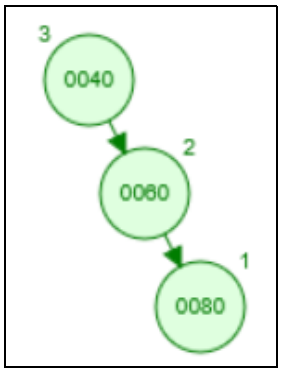
```
int numeroElementosPositivos (PNodoABP T)
{
   int  numEsq, numDir;
   if (T == NULL)
      return  0;
   if (T->Elemento > 0) {
      numEsq = numeroElementosPositivos(T->Esquerda);
      numDir = numeroElementosPositivos(T->Direita);
      return  numEsq + numDir + 1;
   }
   else
      return  numeroElementosPositivos(T->Direita);
}
```

2.

```
PNodoABP paiNodoMaiorMenorX_it (PNodoABP T, int X)
{
   PNodoABP  pai, paiFinal, P;

   // 1º caso: T é vazia
   if (T == NULL)
      return  NULL;

   // 2º caso: o menor elemento maior que X é a raiz
   if (T->Elemento < X  &&  T->Direita == NULL)
      return  NULL;

   // 3º caso: o menor elemento maior que X se existe não é a raiz
   pai = T;
   paiFinal = NULL;
   if (T->Elemento < X)
      P = T->Direita;      // T->Direita != NULL
   else
      P = T->Esquerda;
   while (P != NULL) {
      if (P->Elemento < X){
         paiFinal = pai;
         pai = P;
         P = P->Direita;      // P->Direita != NULL
      }
      else {
         pai = P;
         P = P->Esquerda;
      }
   }
   return  paiFinal;
}
```

E. ABP Balanceadas/Equilibradas (AVL)

Inserir **40, 60** e **80**, por esta ordem:



A árvore fica **equilibrada** com a inserção dos nodos **40** e **60**

A árvore fica **desequilíbrada** com a inserção do nodo **80**

O **desequilibrio** acontece na árvore com raiz no nodo **40**, pois

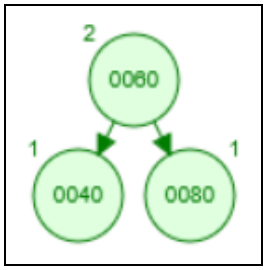
- o seu filho direito (nodo **60**) tem altura 2
- o seu filho esquerdo (NULL) tem altura 0
- altura(filho direito) – altura(filho esquerdo) = 2 - 0 = 2

Todos as subárvores com raiz nos nodos descendentes do nodo **40** estão equilibradas

Como

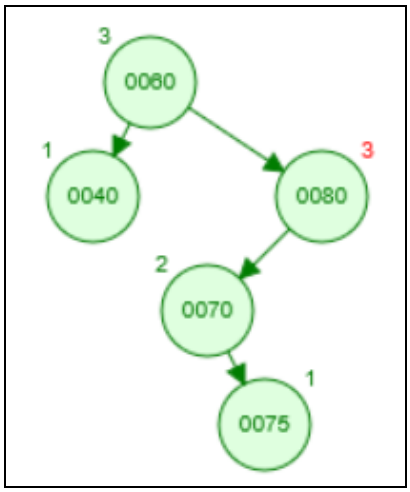
- o filho **direito** do nodo **40** (nodo **60**) tem **mais altura** que o filho esquerdo do nodo **40** (**NULL**), e
- o filho **direito** do nodo **60** (nodo **80**) tem **mais altura** que o filho esquerdo do **60** (**NULL**),

então **reequilibrar** a árvore aplicando uma **rotação simples à esquerda** do nodo **40**, usando o nodo **60**



A árvore está equilibrada.

Inserir **70** e **75**, por esta ordem:



A árvore fica equilibrada com a inserção dos nodos **70**

A árvore fica **desequilíbrada** com a inserção do nodo **75**

O desequilibrio acontece na árvore com raiz no nodo **80**, pois

- o seu filho esquerdo (nodo **70**) tem altura 2
- o seu filho direito (**NULL**) tem altura 0
- altura(filho esquerdo) – altura(filho direito) = 2 - 0 = 2

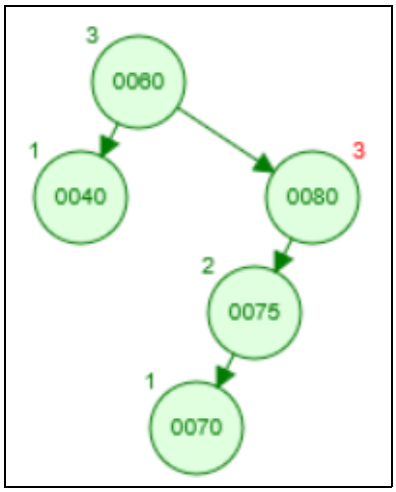
Todos as subárvores com raiz nos nodos descendentes do nodo **60** estão equilibradas

Como

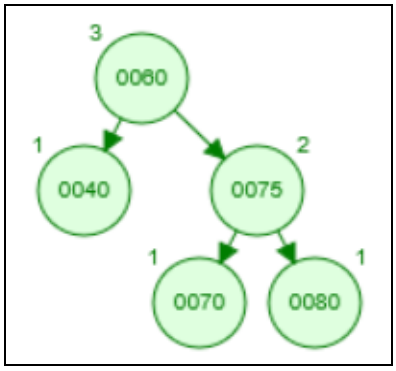
- o filho **esquerdo** do nodo **80** (nodo **70**) tem **mais altura** que o filho direito do nodo **80** (**NULL**), e
- o filho **direito** do nodo **70** (nodo **75**) tem **mais altura** que o filho esquerdo do nodo **70** (**NULL**),

então **reequilibrar** a árvore aplicando uma **rotação dupla à direita** do nodo **80** usando o nodo **75**

1º) **rotação simples à esquerda** do nodo **70** usando o nodo **75**:

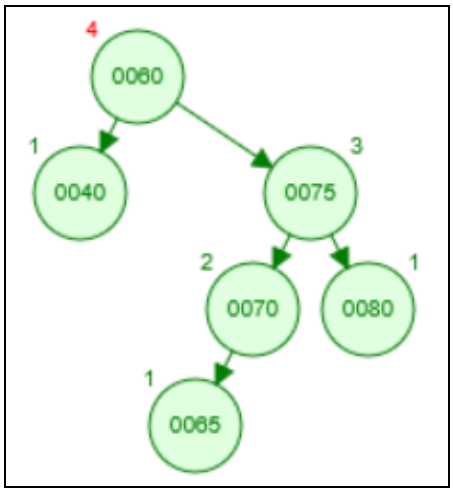


2º) **rotação simples à direita** do nodo **80** usando o nodo **75**:



A árvore está equilibrada.

Inserir **65**:



A árvore fica **desequilíbrada** com a inserção do nodo **65**

O desequilibrio acontece na árvore com raiz no nodo **60**, pois

- o seu filho direito (nodo **75**) tem altura 3
- o seu filho esquerdo (**NULL**) tem altura 1
- altura(filho direito) – altura(filho esquerdo) = 3 - 1 = 2

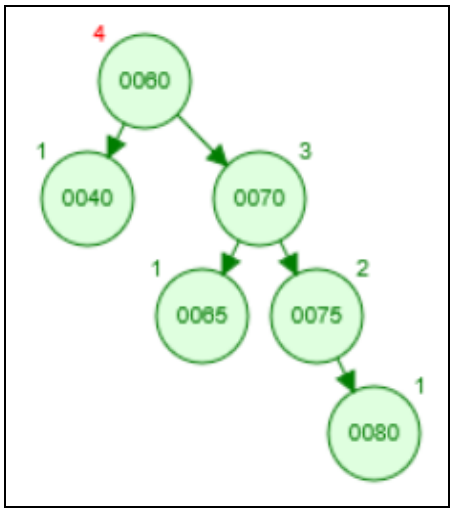
Todos as subárvores com raiz nos nodos descendentes do nodo **60** estão equilibradas

Como

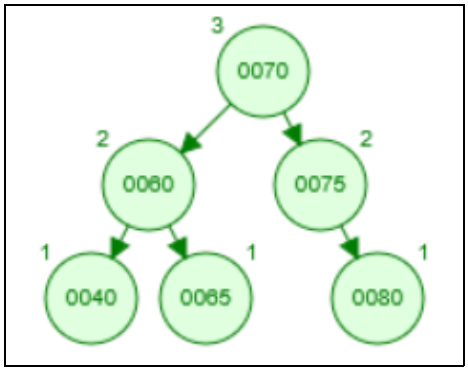
- o filho **direito** do nodo **60** (nodo **75**) tem **mais altura** que o filho esquerdo do nodo **60** (nodo **40**), e
- o filho **esquerdo** do nodo **75** (nodo **70**) tem **mais altura** que o filho direito do nodo **75** (nodo **80**),

então **reequilibrar** a árvore aplicando uma **rotação dupla à esquerda** do nodo **60** usando o nodo **70**

1º) **rotação simples à direita** do nodo **75** usando o nodo **70**:



2º) **rotação simples à esquerda** do nodo **60** usando o nodo **70**:



A árvore está equilibrada.

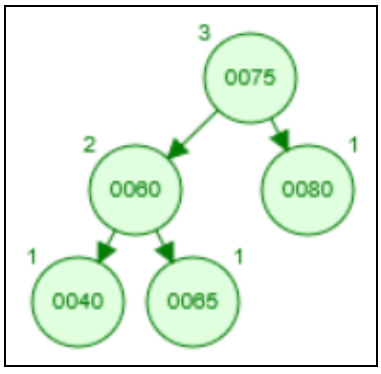
Remover a raiz da ABP (nodo 70):

Remover a raiz da árvore (nodo **70**) implica substituir o elemento **70** por um dos seguintes elementos:

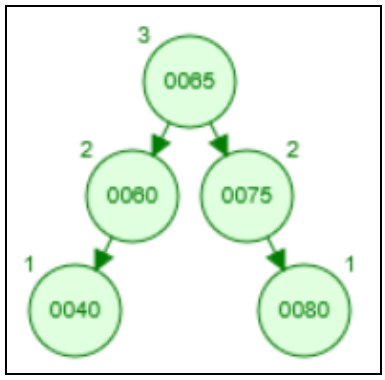
- **75** (menor elemento da subárvore direita – subárvore com os maiores que 70) [Caso 1]
- **65** (maior elemento da subárvore esquerda – subárvore com os menores que 70) [Caso 2]

e de seguida remover o nodo substituto.

Caso 1: substituir pelo elemento **75**



Caso 2: substituir pelo elemento **65**

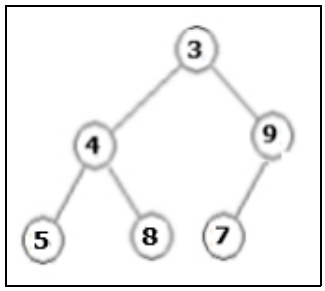


Em ambos os casos, a árvore está equilibrada.

F. Heaps

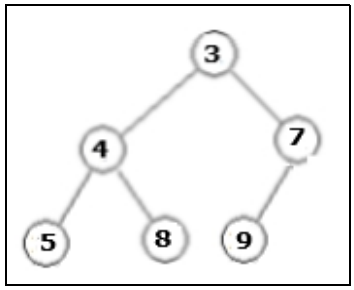
1.

a) inserir um nodo com o valor **7** como filho esquerdo do nodo com o valor **9**



b) comparar o nodo com o valor **7** com o seu pai, e caso tenha menor (melhor) valor trocá-los. Fazer isto até um nodo com valor **7** ter valor maior (pior) do que o seu pai ou chegar à raiz da minHeap:

i) como 7 < 9 (melhor que o seu pai), então trocá-los



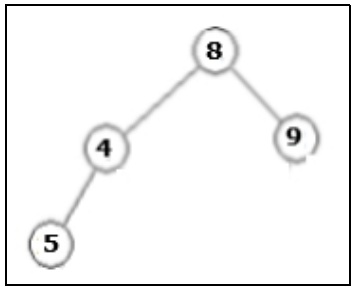
ii) como 7 > 3 (pior que o seu pai), então parar o processo.

2.

1º Extrair/tomar o elemento que se encontra na raiz do minHeap original: **3**

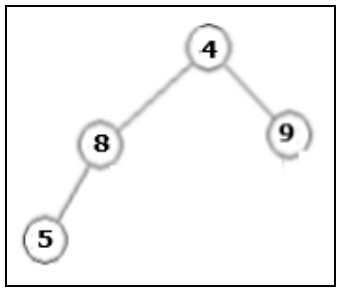
2º Remover o nó que se encontra na raiz da árvore, da seguinte forma:

a) copiar para a raiz (com valor **3**) o elemento do nodo mais à direita do último nível do minHeap (com valor **8**) e remover este último nodo

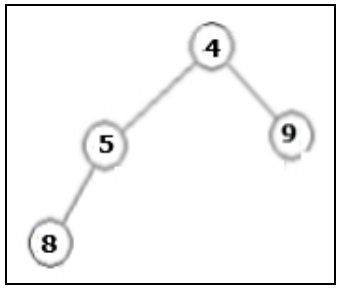


b) repor as propriedade de minHeap: comparar o nodo com valor **8** (agora raiz) com o menor (melhor) dos seus filhos e, caso seja maior (pior) do que esse filho, então trocá-los; fazer isto até o nodo com valor **8** ser menor (melhor) que os seus filhos ou atingir o último nível do minHeap:

i) como 8 > 4 (4 = min{4, 9}) (pior que o melhor dos seus filhos), então trocar o elemento com valor 8 pelo elemento com valor 4



ii) como 8 > 5 (5 = min{5}) (pior que o seu filho), então trocar o elemento com valor 8 pelo elemento com valor 5



iii) como o nodo 8 não tem filhos, então terminar processo.

D. Grafos

1.

Passo 1:

S = 5

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
R	∞	∞	∞	∞	0	2	∞	13
Pai	5	5	5	5	5	5	5	5

Permanentes ← 5

Temporários ← 1 - 2 – 3 – 4 – 6 – 7 - 8

Passos 2 e 3:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
R	∞ 21 **20**	∞ **11**	∞ 22 **15**	∞ **8**	**0**	**2**	∞ **16**	**13**
Pai	5 2 **3**	5 **4**	5 6 **2**	5 **6**	**5**	**5**	5 **6**	**5**

Permanentes ← 5 - 6 - 4 - 2 - 8 - 3 - 7 - 1

Temporários ← 1 - 2 - 3 - 4 - 6 - 7 - 8

Temporários ≠ ∅ ⇒ Continuar

k ← 6 ---> R(6) = 2 = minimo entre os rótulos dos nós temporários (o nó 6 passa a Permanente)

j ← 3, 4, 7, 5 (adjacentes ao nó 6)

3: R(6) + C(6,3) < R(3) ? ⇔ 2 + 20 < ∞ SIM ⇒ **R(3) = 22 e Pai(3) = k = 6**

4: R(6) + C(6,4) < R(4) ? ⇔ 2 + 6 < ∞ SIM ⇒ **R(4) = 8 e Pai(4) = k = 6**

7: R(6) + C(6,7) < R(7) ? ⇔ 2 + 14 < ∞ SIM ⇒ **R(7) = 16 e Pai(7) = k = 6**

5: não pode ser rotulado, pois é Permanente

Temporários ≠ ∅ ⇒ Continuar

k ← 4 ---> R(4) = 6 = minimo entre os rótulos dos nós temporários (o nó 4 passa a Permanente)

j ← 2, 6 (adjacentes ao nó 4)

2: R(4) + C(4,2) < R(2) ? ⇔ 8 + 3 < ∞ SIM ⇒ **R(2) = 11 e Pai(2) = k = 4**

6: não pode ser rotulado, pois é Permanente

Temporários ≠ ∅ ⇒ Continuar

k ← 2 ---> R(2) = 11 = minimo entre os rótulos dos nós temporários (o nó 2 passa a Permanente)

j ← 1, 3, 4 (adjacentes ao nó 2)

1: R(2) + C(2,1) < R(1) ? ⇔ 11 + 10 < ∞ SIM ⇒ **R(1) = 21 e Pai(1) = k = 2**

3: R(2) + C(2,3) < R(3) ? ⇔ 11 + 4 < 22 SIM ⇒ **R(3) = 15 e Pai(3) = k = 2**

4: não pode ser rotulado, pois já é Permanente

Temporários ≠ ∅ ⇒ Continuar

k ← 8 ---> R(8) = 13 = minimo entre os rótulos dos nós temporários (o nó 8 passa a Permanente)

j ← 5, 7 (adjacentes ao nó 8)

5, 7: não podem ser rotulados, pois são Permanentes

Temporários ≠ ∅ ⇒ Continuar

k ← 3 ---> R(3) = 15 = minimo entre os rótulos dos nós temporários (o nó 3 passa a Permanente)

j ← 1, 2, 6 (adjacentes ao nó 3)

1: R(3) + C(3,1) < R(1) ? ⇔ 15 + 5 < 21 SIM ⇒ **R(1) = 20 e Pai(1) = k = 3**

2, 6: não podem ser rotulados, pois são Permanentes

Temporários ≠ ∅ ⇒ Continuar

k ← 7 ---> R(7) = 16 = minimo entre os rótulos dos nós temporários (o nó 7 passa a Permanente)

j ← 1, 6, 8 (adjacentes ao nó 7)

1: R(7) + C(7,1) < R(1) ? ⇔ 16 + 8 < 20 NÃO

6, 8: não podem ser rotulados, pois são Permanentes

Temporários ≠ ∅ ⇒ Continuar

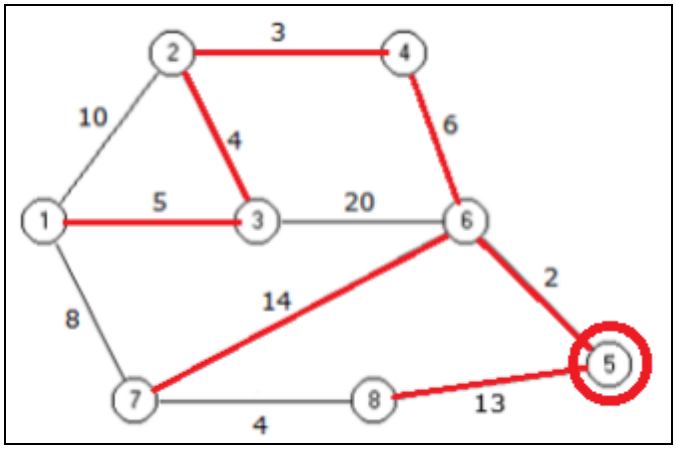
k ← 1 ---> R(1) = 20 = minimo entre os rótulos dos nós temporários (o nó 1 passa a Permanente)

j ← 2, 3, 7 (adjacentes ao nó 1)

2, 3, 7: não podem ser rotulados, pois são Permanentes

Temporários = ∅ ⇒ PARAR

Árvore dos Caminhos Mais Curtos do nó **5** (S = 5) para os **restantes nós**, é a seguinte:



2.

Caminho Mais Curto do nó **5** para o nó **1**:

Custo = R(1) = **20**

p = { 1 --> Pai(1)=3 --> Pai(3)=2 --> Pai(2)=4 --> Pai(4)=6 --> Pai(6)=5 } = **[5, 6, 4, 2, 3, 1]**