UNIVERSIDADE DA BEIRA INTERIOR

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

Exame Época de Recurso **Resolução** **2025/2026**

NOTA: A forma como esta resolução se apresenta, para além de mostrar uma possível solução para cada questão, serve também para explicar os métodos e algoritmos usados. Assim sendo, não seria necessário, em especial nalgumas questões, detalhar tanto as respostas.

A. Análise de complexidade dos algoritmos

1.

k	k < n [S/N]	j	j <= 3 [S/N]	printf [S/N]
(k=0) 0	0 < n [S]	(j=1) 1	1 <= 3 [S]	OLA
		(j++) 2	2 <= 3 [S]	OLA
		(j++) 3	3 <= 3 [S]	OLA
		(j++) 4	4 <= 3 [N]	
(k++) 1	1 < n [S]	(j=1) 1	1 <= 3 [S]	OLA
		(j++) 2	2 <= 3 [S]	OLA
		(j++) 3	3 <= 3 [S]	OLA
		(j++) 4	4 <= 3 [N]	
...	...	...	...	...
(k++) n-1	n-1 < n [S]	(j=1) 1	1 <= 3 [S]	OLA
		(j++) 2	2 <= 3 [S]	OLA
		(j++) 3	3 <= 3 [S]	OLA
		(j++) 4	4 <= 3 [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 <= 3**".

T(n) = **4** [k=0] + **4** [k=1] + ... + **4** [k=n-1] = **4 x n => T(n)** = **O(n)**

B. Algoritmos de ordenação e de pesquisa

```
int quantidadeSegundosMenores (int A[], int N)
{
   int  k, cont, min1, min2;
   min1 = A[0];  // menor valor
   k = 1;
   while (k < N  &&  A[k] == min1)
      k = k + 1;
   if (k < N) {
      min2 = A[k];   // 2º menor valor
      cont = 1;
      k = k + 1;
      while (k < N  &&  A[k] == min2){
         cont++;
         k++;
      }
      return cont;
   }
   else
      return 0;
}
```

C. Listas ligadas

1.

```
int somaElementosPosMaioresA (PNodoLista L, int A) {
   int  soma;
   // caso base/terminal
   if (L == NULL)
      return  0;
   // caso geral
   soma = somaElementosPosMaioresA(L->Prox, A);
   if (L->Elemento > 0  &&  L->Elemento > A)
      return  L->Elemento + soma;
   else
      return  soma;
}
```

2.

```
PNodoLista remover (PNodoLista L) {
   PNodoLista  P, PMaior;
   if (L == NULL)
      return  L;
   PMaior = L;  // procurar o maior elemento e marcá-lo com PMaior
   P = L->Prox;
   while (P != NULL) {
      if (P->Elemento > PMaior->Elemento)
         PMaior = P;
      P = P->Prox;
   }
   // inserir o novo elemento como antecessor do maior
   if (PMaior->Prox != NULL) {  // o nodo com o maior elemento tem sucessor (não é o último)
      P = PMaior->Prox;   // marcar o nodo a remover para libertar a memória que ocupa
      PMaior->Prox = PMaior->Prox->Prox;   // ligar o maior elemento ao sucessor do seu sucessor
      P = libertarNodoLista(P);  // libertar o nodo removido
   }
   return  L;
}
```

D. Pilhas

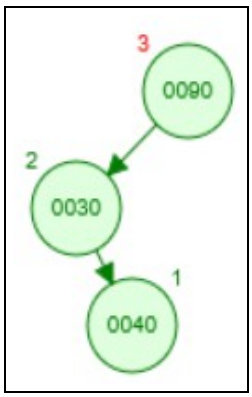
```
int somaElementosDiferentesTopoFundo (PNodoPilha *S) {
   PNodoPilha  SAux;
   int  soma, fundo, top;
   if (pilhaVazia(*S) == 1)  // se a pilha está vazia, a soma é 0
      return  -1;
   SAux = criarPilha();
   top = topo(*S);   // guardar o elemento do topo da pilha S
   // determinar o fundo da pilha e construir uma pilha auxiliar com os elementos de S
   while (pilhaVazia(*S) == 0) {
      SAux = push(topo(*S), SAux);
      *S = pop(*S);
   }
   fundo = topo(SAux);    // guardar o elemento do fundo da pilha S = elemento do topo de SAux
   soma = 0;
   while (pilhaVazia(SAux) == 0) {  // reconstruir a pilha S e
      if(topo(SAux) != top  &&  topo(SAux) != fundo)
         soma = soma + topo(SAux);  // somar os elementos diferentes de top e de fundo
      *S = push(topo(SAux), *S);
      SAux = pop(SAux);
   }
   return  soma;
}
```

E. Árvores Binárias de Pesquisa (ABP)

```
int maiorMenorX (PNodoABP T, int X)
{
   int  max;
   // caso base/terminal
   if (T == NULL)
      return  -1;
   // caso geral
   if (T->Elemento < X){
      max = maiorMenorX(T->Direita, X);
      if (max == -1)
         return  T->Elemento;
      else
         return  max;
   }
   else
      return  maiorMenorX(T->Esquerda, X);
}
```

F. ABP Balanceadas/Equilibradas (AVL)

Inserir **90, 30** e **40**, por esta ordem:



A árvore fica **equilibrada** com a inserção dos nodos **90** e **30**

A árvore fica **desequilibrada** com a inserção do nodo **40**

O **desequilíbrio** acontece no nodo **90**, pois

- o seu filho direito (**NULL**) tem altura 0
- o seu filho esquerdo (**30**) tem altura 2
- altura(filho esquerdo) – altura(filho direito) = 2 - 0 = 2

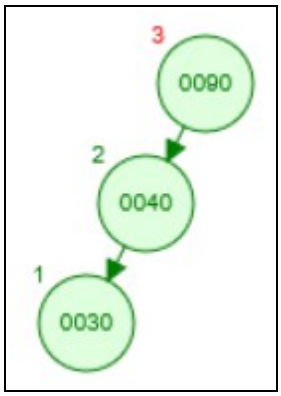
Todos as subárvores com raiz nos nodos descendentes do nodo **90** estão equilibradas

Como:

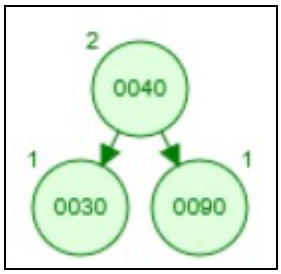
- o filho **esquerdo** do nodo **90** (**30**) tem **mais altura** que o filho **direito** do nodo **90** (**NULL**), e
- o filho **direito** do nodo **30** (**40**) tem **mais altura** que o filho **esquerdo** do nodo **30** (**NULL**),

então **reequilibrar** a árvore aplicando uma **rotação dupla à direita** do nodo **90**

1º) **rotação simples à esquerda** do nodo **30** usando o nodo **40**:

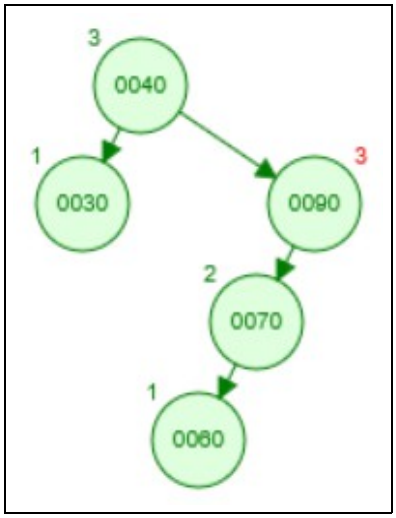


2º) **rotação simples à direita** do nodo **90** usando o nodo **40**:



A árvore está equilibrada.

Inserir **70** e **60**, por esta ordem:



A árvore fica equilibrada com a inserção do nodo **70**

A árvore fica **desequilibrada** com a inserção do nodo **60**

O desequilíbrio acontece no nodo **90**, pois

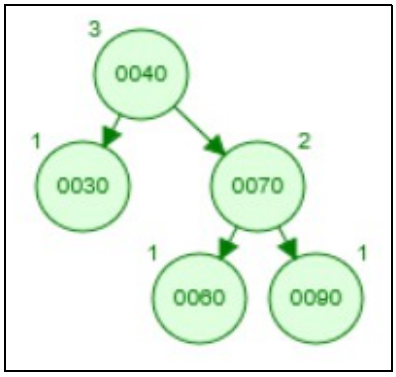
- o seu filho **direito** (**NULL**) tem altura o
- o seu filho **esquerdo** (**70**) tem altura 2
- altura(filho esquerdo) – altura(filho direito) = 2 - 0 = 2

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

Como:

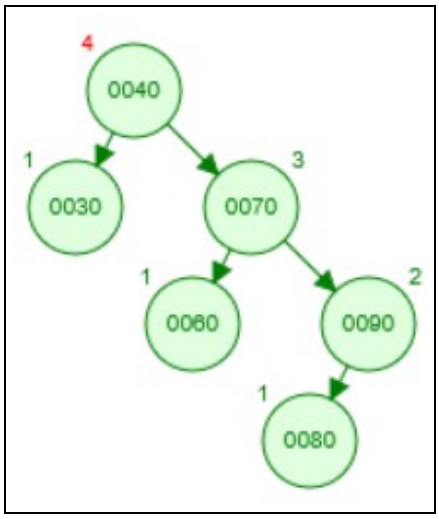
- o filho **esquerdo** do nodo **90** (**70**) tem **mais altura** que o filho **direito** do nodo **90** (**NULL**), e
- o filho **esquerdo** do nodo **70** (**60**) tem **mais altura** que o filho **direito** do nodo **70** (**NULL**),

então **reequilibrar** a árvore aplicando uma **rotação simples à direita** do nodo **90** usando o nodo **70**



A árvore está equilibrada.

Inserir **80**:



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

O **desequilíbrio** acontece no nodo **40**, pois

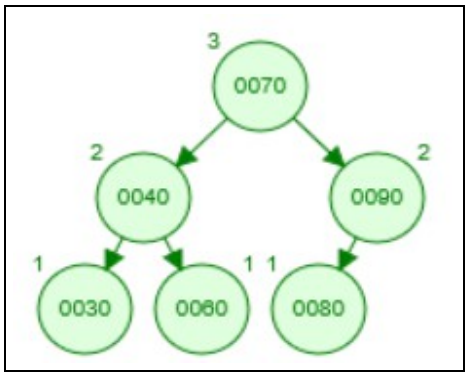
- o seu filho **direito** (**70**) tem altura 3
- o seu filho **esquerdo** (**30**) tem altura 1
- altura(filho direito) – altura(filho esquerdo) = 3 - 1 = 2

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

Como:

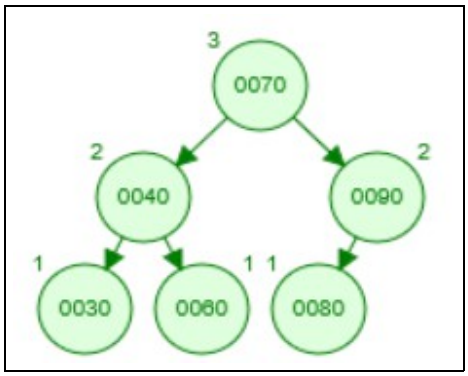
- o filho **direito** do nodo **40** (**70**) tem **mais altura** que o filho **esquerdo** do nodo **40** (**30**), e
- o filho **direito** do nodo **70** (**90**) tem **mais altura** que o filho **esquerdo** do nodo **70** (**60**),

então **reequilibrar** a árvore aplicando uma **rotação simples à esquerda** do nodo 40 usando o nodo **70**



A árvore está equilibrada.

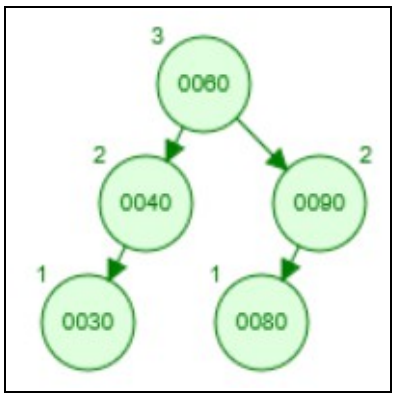
Remover o nodo 70 (raiz da ABP obtida)



Como o nodo **70** tem dois filhos, a remoção deste nodo pode ser feito de duas formas:

1º caso:

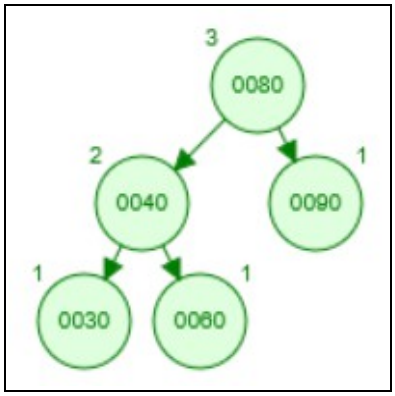
- copiar o valor do nodo com maior valor menor que **70** (**60**) para o nodo com **70** (raiz passa a **60**),
- remover o nodo (folha) com **60**



A árvore continua **equilibrada**

2º caso:

- copiar o valor do nodo com menor valor maior que **70** (**80**) para o nodo com **70** (raiz passa a **80**),
- remover o nodo (folha) com **80**

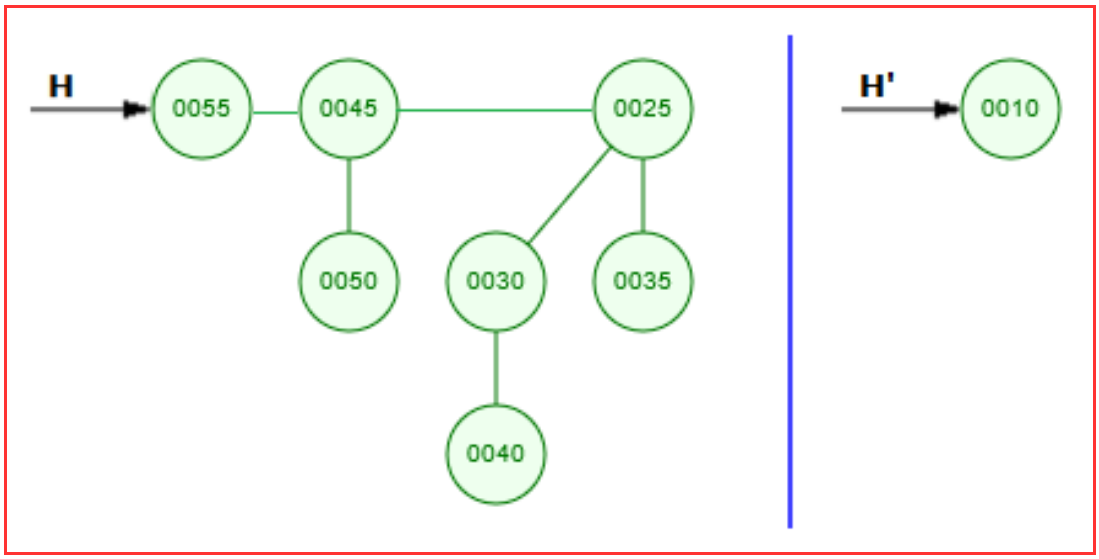


A árvore continua **equilibrada**

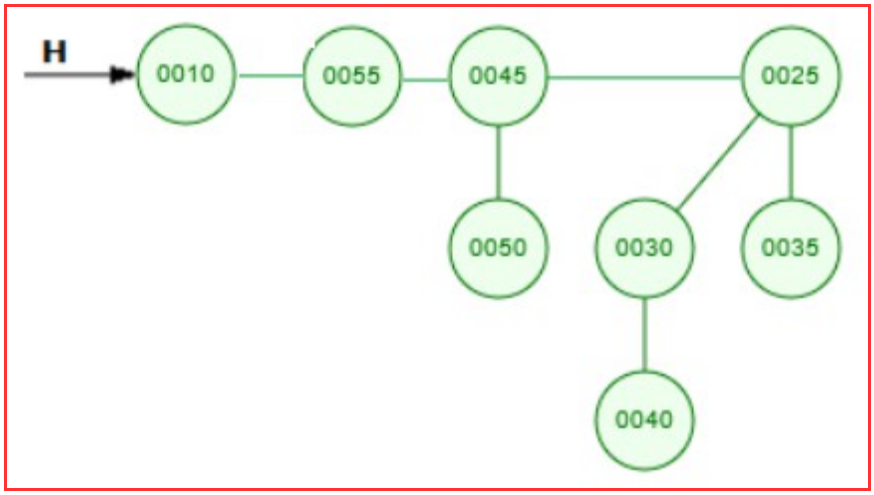
G. Heaps (Filas de Prioridade)

1. O processo resultante da operação **"inserir"** um nodo com valor **10** no heap H é o seguinte:

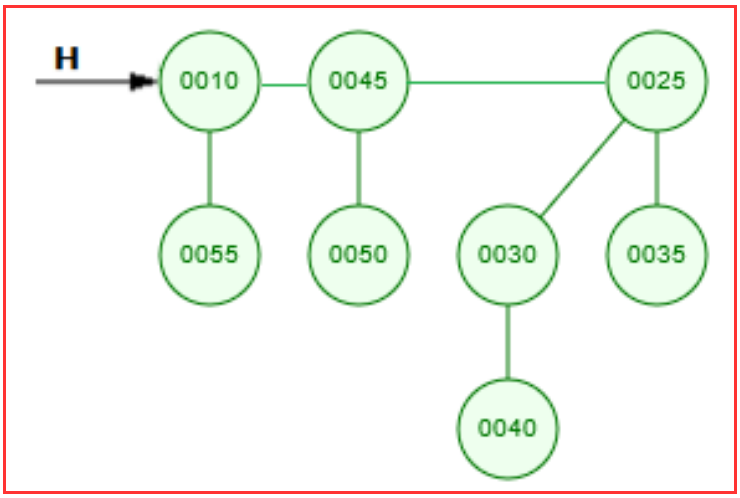
1º) criar um heap H' apenas com um nodo com o valor **10**



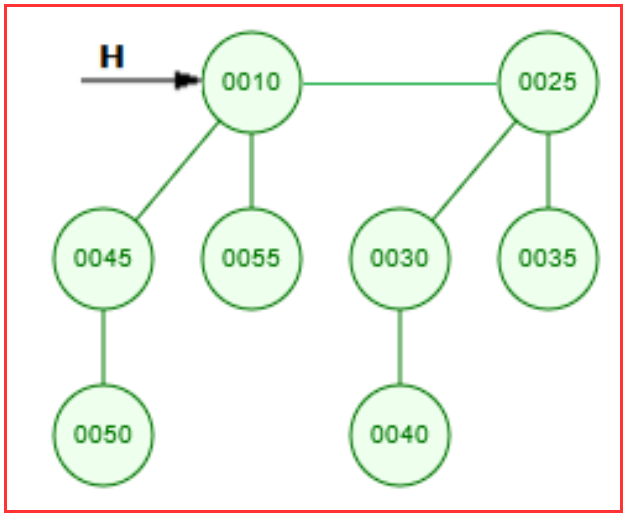
2º) aplicar a operação **união** entre os dois heaps, H e H':



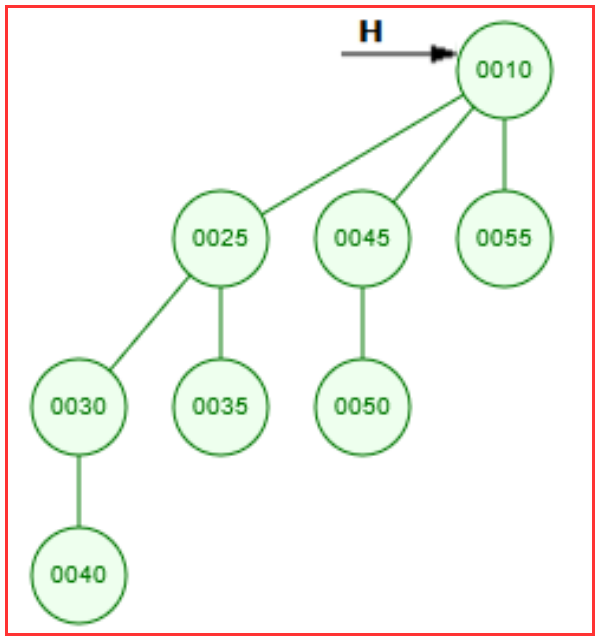
- como existem 2 árvores binomiais de grau 0 (B0) cujas raízes têm os valores 10 e 55, então aplicar a operação **união** entre árvores binomiais formando uma árvore de grau 1 (B1) com raiz com o valor 10 (min{10, 55})



- como agora existem 2 árvores binomiais de grau 1 (B1) cujas raízes têm os valores 10 e 45, então aplicar a operação **união** entre árvores binomiais formando uma árvore de grau 2 (B2) com raiz com o valor 10 (min{10, 45})



- como agora existem 2 árvores binomiais de grau 2 (B2) cujas raízes têm os valores 10 e 25, então aplicar a operação **união** entre árvores binomiais formando uma árvore de grau 3 (B3) com raiz com o valor 10 (min{10, 25})



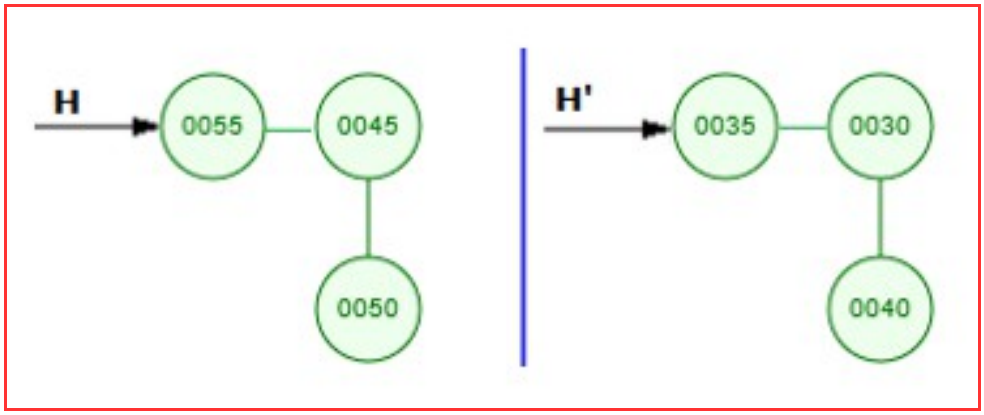
2. O processo resultante da operação **"remover"** aplicado ao heap H original é o seguinte:

1º) extrair o menor valor entre as raízes do heap H: min{ 55, 45, 25 } = **25**

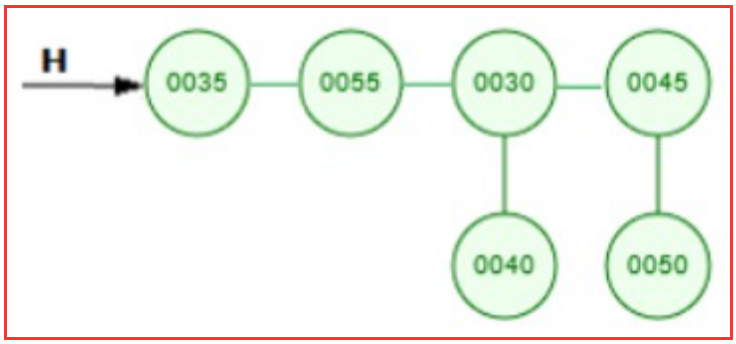
2º) remover o nodo raiz com o valor **25** (menor raiz),

- remover a árvore binomial com raiz com valor 25 do heap H, ficando H apenas com as árvores binomiais com raízes com os valores 55 (B0) e 45 (B1)

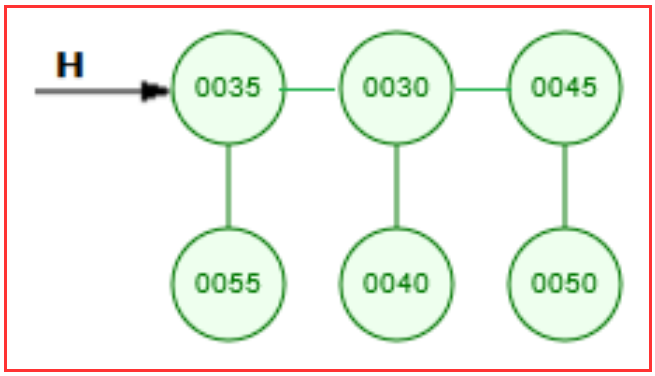
– criar um novo heap H' com os filhos da árvore com raiz com o valor 25



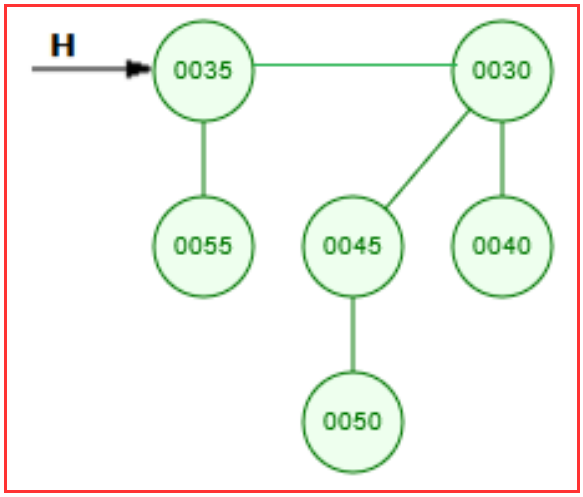
3º) aplicar a operação **união** entre os dois heaps, H e H':



- como existem 2 árvores binomiais de grau 0 (B0) cujas raízes têm os valores 35 e 55, então aplicar a operação **união** entre árvores binomiais formando uma árvore de grau 1 (B1) com raiz com o valor 35 (min{35, 55})



- como agora existem 3 árvores binomiais de grau 1 (B1) cujas raízes têm os valores 35, 30 e 45, então juntar as 2 árvores mais à direita formando uma árvore de grau 2 (B2) com raiz com o valor 30 (min{30, 45})
- a árvore resultante da operação "remover" é formada por duas árvores binomiais de graus 1 (B1) e 2 (B2).



H. Grafos

1.

Fluxo que sai do nó 2 = x23 + x25 + x26 = 2 + 2 + 1 = 5

Fluxo que chega ao nó 3 = x13 + x23 + x43 = 3 + 2 + 1 = 6

2.

	1	2	3	4	5	6
R(k)	[1+, ∞]	[1+, 4]	[5-, 2]	[1+, 3]	[2+, 2]	[3+, 2]

Nós **rotulados** e **não varridos** ← 1, 2, 4, 5, 3, 6

Nós **rotulados** e **varridos** ← 1, 2, 4, 5, 3

Passo1:

R(1) ← [1+, ∞]

Passo 2:

j ← 1 [1 rotulado e não varrido]

k ← 2, 3, 4 (arcos de 1 para k)

2 --> x12 < b12 <=> 5 < 9 SIM, logo **R(2) = [1+, min{ ∞, 9-5 }] = [1+, 4]**

3 --> x13 < b13 <=> 3 < 3 NÃO, logo não é possível rotular o nó 3

4 --> x14 < b14 <=> 4 < 7 SIM, logo **R(4) = [1+, min{ ∞, 7-4 }] = [1+, 3]**

j ← 2 [2 rotulado e não varrido] (podia ser o nó 4)

k ← 3, 5, 6 (arcos de 2 para k)

3 --> x23 < b23 <=> 2 < 2 NÃO, logo não é possível rotular o nó 3

5 --> x25 < b25 <=> 2 < 4 SIM, logo **R(5) = [2+, min{ 4, 4-2 }] = [2+, 2]**

6 --> x26 < b26 <=> 1 < 1 NÃO, logo não é possível rotular o nó 6

k ← 1 (arcos de k para 2)

1 --> nó 1 já rotulado

j ← 4 [4 rotulado e não varrido] (podia ser o nó 5)

k ← 3, 6 (arcos de 4 para k)

3 --> x43 < b43 <=> 1 < 1 NÃO, logo não é possível rotular o nó 3

6 --> x46 < b46 <=> 3 < 3 NÃO, logo não é possível rotular o nó 6

k ← 1 (arcos de k para 4)

1 --> nó 1 já rotulado

j ← 5 [5 rotulado e não varrido]

k ← 6 (arcos de 5 para k)

6 --> x56 < b56 <=> 6 < 6 NÃO, logo não é possível rotular o nó 6

k ← 2, 3 (arcos de k para 5)

2 --> nó 2 já rotulado

3 --> x35 > 0 <=> 4 > 0 SIM, logo **R(3) = [5-, min{ 2, 4 }] = [5-, 2]**

j ← 3 [3 rotulado e não varrido]

k ← 5, 6 (arcos de 3 para k)

5 --> nó 5 já rotulado

6 --> x36 < b36 <=> 2 < 4 SIM, logo **R(6) = [3+, min{ 2, 4-2 }] = [3+, 2]**

k ← 1, 2, 4 (arcos de k para 5)

1 --> nó 1 já rotulado

2 --> nó 2 já rotulado

4 --> nó 4 já rotulado

Como o nó **T = 6 está rotulado**, foi encontrado um **c.a.f.**; logo, passar ao **Passo 3**.

Passo 3:

R(T=6) = [3+, 2] => k = 3 e Q(6) = 2

x36 = x36 + Q(6) = 2 + 2 = 4

k=3 ≠ S=1 SIM

R(3) = [5-, 4] => **x35 = x35 - Q(6) = 4 - 2 = 2**

k = 5

k=5 ≠ S=1 SIM

R(5) = [2+, 2] => **x25 = x25 + Q(6) = 2 + 2 = 4**

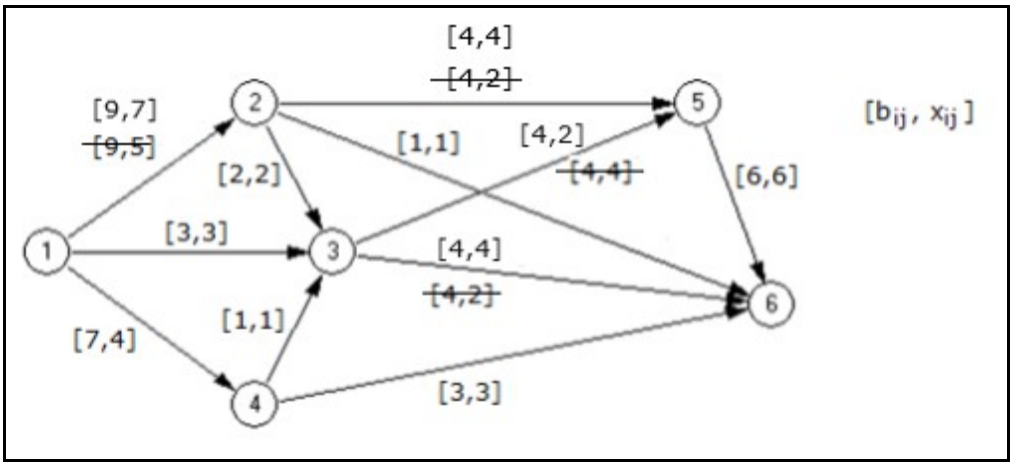
k = 2

k=2 ≠ S=1 SIM

R(2) = [1+, 4] => **x12 = x12 + Q(6) = 5 + 2 = 7**

k = 1

k=1 ≠ S=1 NÃO => Atualização da rede terminado



Nova iteração do algoritmo de Ford-Fulkerson, aplicado à rede atualizada

	1	2	3	4	5	6
R(k)	[1+, ∞]	[1+, 2]		[1+, 3]		

Nós **rotulados** e **não varridos** ← 1, 2, 4, 5, 3

Nós **rotulados** e **varridos** ← 1, 2, 4, 5, 3

Passo1:

R1 ← [1+, ∞]

Passo 2:

j ← 1 [1 rotulado e não varrido]

k ← 2, 3, 4 (arcos de 1 para k)

2 --> x12 < b12 <=> 7 < 9 SIM, logo **R(2) = [1+, min{ ∞, 9-7 }] = [1+, 2]**

3 --> x13 < b13 <=> 3 < 3 NÃO, logo não é possível rotular o nó 3

4 --> x14 < b14 <=> 4 < 7 SIM, logo **R(4) = [1+, min{ ∞, 7-4 }] = [1+, 3]**

k ←

j ← 2 [2 rotulado e não varrido] (podia ser o nó 4)

k ← 3, 5, 6 (arcos de 2 para k)

3 --> x23 < b23 <=> 2 < 2 NÃO, logo não é possível rotular o nó 3

5 --> x25 < b25 <=> 4 < 4 NÃO, logo não é possível rotular o nó 5

6 --> x26 < b26 <=> 1 < 1 NÃO, logo não é possível rotular o nó 6

k ← 1 (arcos de k para 2)

1 --> nó 1 já rotulado

j ← 4 [4 rotulado e não varrido] (podia ser o nó 5)

k ← 3, 6 (arcos de 4 para k)

3 --> x43 < b43 <=> 1 < 1 NÃO, logo não é possível rotular o nó 3

6 --> x46 < b46 <=> 3 < 3 NÃO, logo não é possível rotular o nó 6

k ← 1 (arcos de k para 4)

1 --> nó 1 já rotulado

Como o nó T = 6 **não está rotulado e não é possível rotular** (todos os nós rotulados estão varridos), então **não existe c.a.f.**; logo, o fluxo atual é máximo ==> PARAR

Fluxo máximo = fluxo que sai do nó S = fluxo que chega ao nó T

Como S = 1, fluxo que sai do nó 1 = x12 + x13 + x14 = 7 + 3 + 4 = 14
