UNIVERSIDADE DA BEIRA INTERIOR

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

Exame – Época Normal **Resolução** **2023/2024**

A. Análise de complexidade dos algoritmos

Considere o seguinte bloco de código em linguagem C:

```
for (k = 1; k < n; k = k + 2) {
   for (j = 1; j < k; j++)
      T = k + j;
}
```

1. Simule o bloco de código dado, considerando que **n** é um número inteiro **par muito grande**. Apresente os dados da simulação numa tabela cujo cabeçalho é o seguinte:

k	k < n (V/F)	j	j < k (V/F)	T
(k = 1) 1	1 < n V.	(j=1) 1	1 < 1 F.	
(k=k+2) 3	3 < n V.	(j=1) 1 (j++) 2 (j++) 3	1 < 3 V. 2 < 3 V. 3 < 3 F.	3+1 = 4 3+2 = 5
(k=k+2) 5	5 < n V.	(j=1) 1 (j++) 2 (j++) 3 (j++) 4 (j++) 5	1 < 5 V. 2 < 5 V. 3 < 5 V. 4 < 5 V. 5 < 5 F.	5+1 = 6 5+2 = 7 5+3 = 8 5+4 = 9
...	...	...	...	...
(k=k+2) n-1	n-1 < n V.	(j=1) 1 (j++) 2 ... (j++) n-2 (j++) n-1	1 < n-1 V. 2 < n-1 V. ... n-2 < n-1 V. n-1 < n-1 F.	(n-1)+1 = n (n-1)+2 = n+1 ... (n-1)+(n-2) = 2n-3
(k=k+2) n+1	n+1 < n F.	-	-	-

2. A partir dos resultados da simulação obtidos em 1, determine a **ordem de complexidade** do algoritmo associado ao bloco de código dado em cima. Incidir os cálculos sobre a operação que mais vezes é executada ("**j < k**"). Justifique, apresentando os cálculos efetuados.

T(n) = número de vezes que a operação dominante ("**j < k**") é executada

T(n) = 1 [k=1] + 3 [k=3] + 5 [k=5] + ... + (n-1) [k=n-1] + 0 [k=n+1] =

= 1 + 3 + 5 + ... + (n-1) [progressão aritmética com n/2 termos] =

= ((1 + (n-1)) / 2) x (n/2) = (n/2) x (n/2) = n^2 / 4

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

B. [2.50 val] Listas ligadas

Considere as seguintes declarações:

```
struct Nodo {
   int   Elemento;
   struct  Nodo  *Prox;
};
typedef  struct  Nodo *PNodo;
```

Considere também os seguintes protótipos de funções já implementadas:

```
int listaVazia (PNodo L);   // devolve 1 se lista L é vazia e 0 caso contrário
PNodo criarNodo (int X);   // devolve um ponteiro para um nodo com o elemento X
```

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** de elementos do tipo **inteiro** e um inteiro positivo **X** (X > 0)
- **devolva** a lista L após a **inserção** do nodo com o valor X antes do 1º elemento positivo da lista L (ou seja, de tal forma que X seja o elemento positivo mais próxima da cabeça da lista); ter em conta que a lista L pode ser vazia ou pode não ter elementos positivos.

Nota: Otimizar o algoritmo, de forma a que a lista L seja percorrida apenas uma vez.

```
PNodo inserirPositivo (PNodo L, int X) {
   PNodo  P, novo;
   novo = criarNodo(X);
   if (listaVazia(L) == 1) {
      L = novo;
      return L;
   }
   // verificar se o 1º elemento é positivo
   if (L->Elemento > 0) {  // inserir X no início da lista
      novo->Prox = L;
      L = novo;
      return L;
   }
   // procurar o 1º positivo, caso exista, sabendo que não está na 1ª posição
   P = L;
   while (P->Prox != NULL  &&  P->Prox->Elemento <= 0)
         P = P->Prox;
   // P aponta para o último nodo de L (se P->Prox == NULL) ou
   // P aponta para o nodo anterior ao 1º nodo com valor positivo (se P->Prox != NULL)
   novo->Prox = P->Prox;
   P->Prox = novo;
   return L;
}
```

C. [2.00 val] EAD Pilha

Considere as seguintes declarações de EAD Pilha de elementos do tipo inteiro:

struct NodoPilha { ... };
typedef struct NodoPilha *PNodoPilha;

e os seguintes protótipos de funções (operações básicas):

PNodoPilha criarPilha ();
int pilhaVazia (PNodoPilha S); // devolve **1** se S está vazia e **0** caso contrário
PNodoPilha push (int X, PNodoPilha S);
PNodoPilha pop (PNodoPilha S);
int topo (PNodoPilha S);

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

- **receba** uma **pilha S** já preenchida com valores inteiros não nulos (parâmetro da função),
- **devolva** dois resultados:
  - o elemento do topo de pilha S, caso exista (se não existe, devolver 0), e
  - a pilha S **sem** os elementos **negativos** (apenas os positivos, se existirem), mas mantendo a ordem inicial dos elementos em S

Ex: S = [-3, 12, -4, 5, 4, -5], topo(S) = -3 ==> resultado: top = -3 e S = [12, 5, 4]

```
PNodoPilha exercicioC (PNodoPilha S, int *top) {
    PNodoPilha Aux;
    Aux = criarPilha();
    *top = 0;
    if (pilhaVazia(S) == 1)
        return S;
    // S não está vazia, logo tem pelo menos um elemento
    *top = topo(S);
    while (pilhaVazia(S) == 0) {
        if(topo(S) > 0)
            Aux = push(topo(S), Aux);
        S = pop(S);
    }
    while (pilhaVazia(Aux) == 0) {
        S = push(topo(Aux), S);
        Aux = pop(Aux);
    }
    return S;
}
```

D. [1.50 val + 2.00 val] EAD Árvore Binária de Pesquisa (ABP)

Considere a seguinte declaração de EAD Árvore Binária de Pesquisa:

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

1. Implemente **uma função em C** que
- **receba** uma ABP **T**,
- **devolva** a quantidade de nodos de T sem filhos (folhas).

```
int numNodosSemFilhos (PNodoABP  T){
   int  numEsq, numDir;
   // caso terminal/base
   if (T == NULL)
      return 0;
   // caso geral
   numEsq = numNodosSemFilhos(T->Esquerda);
   numDir = numNodosSemFilhos(T->Direita);
   if (T->Esquerda == NULL  &&  T->Direita == NULL)
      return 1 + numEsq + numDir;
   else
      return numEsq + numDir;
}
```

---

2. Implemente **uma função em C** que
- **receba** uma ABP **T** e um número inteiro **N**
- **devolva** um ponteiro para o nodo-pai do nodo com **valor** no campo **Elemento igual** a **N**, caso exista; deve devolver **NULL** se não existir (T é vazia, não existe nenhum nodo com o valor N no campo Elemento ou se este nodo for a raiz de T)

Versão iterativa (não recursiva):

```
PNodoABP  pNodoPai_It (PNodoABP T, int N) {
   PNodoABP  P = NULL;
   if (T == NULL  ||  T->Elemento == N)
      return NULL;
   while (T != NULL && P == NULL) {
      // T->Elemento != N
      if (T->Elemento < N)
         if (T->Direita != NULL  &&  T->Direita->Elemento == N)
            P = T;
         else
            T = T->Direita;
      else  //  T->Elemento > N
         if (T->Esquerda != NULL  &&  T->Esquerda->Elemento == N)
            P = T;
         else
            T = T->Esquerda;
   }
   return P;
}
```

Versão recursiva:

```
PNodoABP  pNodoPai_Rec (PNodoABP T, int N) {
   if (T == NULL  ||  T->Elemento == N)
      return NULL;
   // T != NULL e T->Elemento != N
   if (T->Elemento < N)
      if (T->Direita != NULL)
         if (T->Direita->Elemento == N)
            return T;
         else
            return pNodoPai_Rec(T->Direita, N);
      else
         return NULL;
   // T-Elemento > N
   if (T->Esquerda != NULL)
      if (T->Esquerda->Elemento == N)
         return T;
      else
         return pNodoPai_Rec(T->Esquerda, N);
   else
      return NULL;
}
```

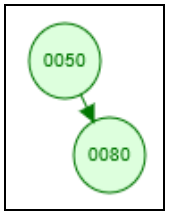
E. [2,00 val + 0,50 val] ABP Balanceadas/Equilibradas (AVL)

1. **Construa** uma **ABP do tipo AVL inserindo** os nodos com os seguintes valores (um a um e pela ordem apresentada): **50**, **80**, **90**, **70**, **60**, **40** e **75**. (**NOTA**: não é necessário colocar as alturas). Sempre que **inserir um nodo**, deve **redesenhar** a **ABP** obtida (acrescentar o nodo à árvore) e **verificar se continua balanceada/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,

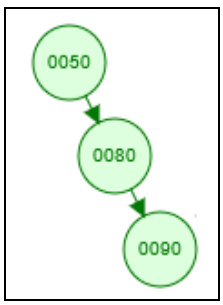
   2º) reequilibrar a árvore e redesenhá-la (apresentando todos as árvores intermédias obtidas)

Inserir **50** e **80**, por esta ordem:



A árvore está equilibrada em ambos os casos.

Inserir **90**:



A árvore fica desequilíbrada no nodo **50**, pois

- o seu filho esquerdo (NULL) tem altura -1
- o seu filho direito (80) tem altura 1
- altura(filho direito) – altura(filho esquerdo) = 1 - (-1) = 2

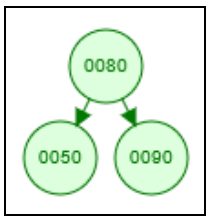
Como

- o filho direito de 50 (80) tem mais altura que o filho esquerdo de 50 (NULL), e

- o filho direito de 80 (90) tem mais altura que o filho esquero de 80 (NULL),

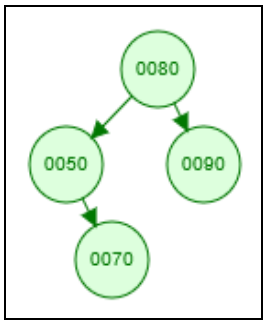
então reequilibrar a árvore aplicando uma **rotação simples à esquerda** do nodo 50 usando o nodo 80

Árvore após rotação simples à esquerda do nodo 50:



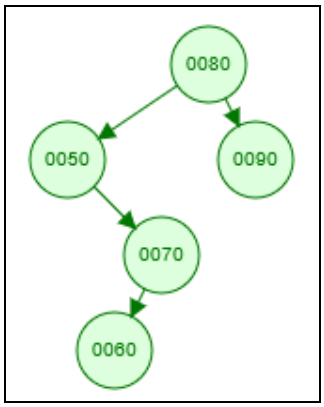
A árvore está equilibrada.

Inserir **70**:



A árvore está equilibrada.

Inserir **60**:



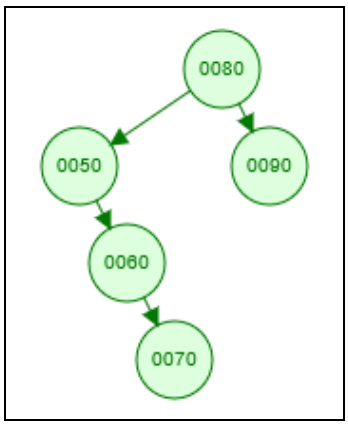
A árvore fica desequilíbrada no nodo **50**, pois

- o seu filho esquerdo (NULL) tem altura -1
- o seu filho direito (70) tem altura 1
- altura(filho direito) – altura(filho esquerdo) = 1 - (-1) = 2

Como

- o filho direito do nodo 50 (nodo 70) tem mais altura que o filho esquerdo de 50 (NULL), e
- o filho esquerdo de 70 (60) tem mais altura que o filho direito de 70 (NULL),

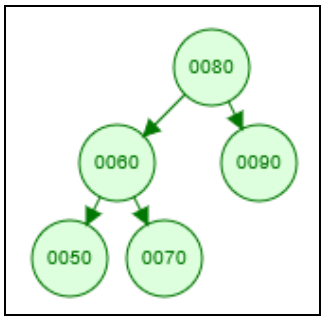
então reequilibrar a árvore aplicando uma **rotação dupla à esquerda** (rotação simples à direita seguida de rotação simples à esquerda), usando o nodo 60



1º) **rotação simples à direita** do nodo 70 usando o nodo **60**:

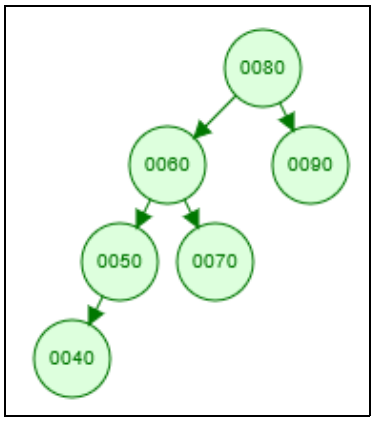
A árvore continua desequilibrada no nodo 50

2º) **rotação simples à esquerda** do nodo 50 usando o nodo **60**:



A árvore está equilibrada.

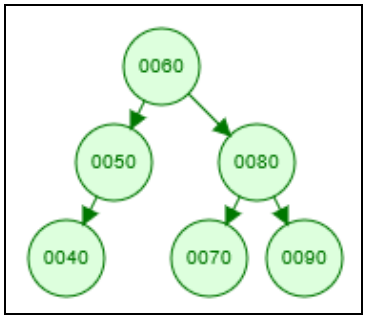
Inserir **40**:



A árvore fica desequilíbrada no nodo **80**, pois
- o seu filho esquerdo (nodo 60) tem altura 2
- o seu filho direito (nodo 90) tem altura 0
- altura(filho esquerdo) – altura(filho direito) = 2 - 0 = 2

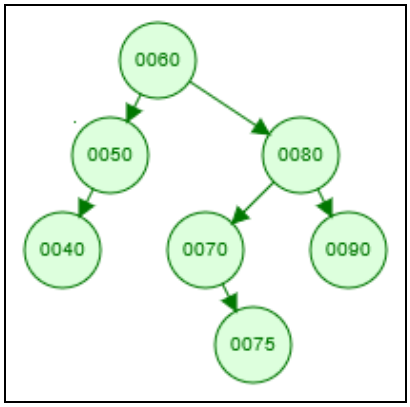
Como
- o filho esquerdo do nodo 80 (60) tem mais altura que o filho direito do nodo 80 (90), e
- o filho esquerdo do nodo 60 (50) tem mais altura que o filho direito do nodo 60 (70),

então reequilibrar a árvore aplicando uma **rotação simples à direita** do nodo 80 usando o nodo 60



A árvore está equilibrada.

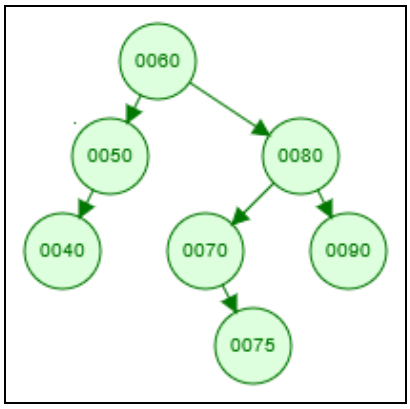
Inserir **75**:



A árvore está equilibrada.

---

2. **Remova** o nodo que é **filho direito** da **raiz** da ABP **obtida em 1.**, **redesenhe** a árvore obtida, **verifique** se continua equilibrada e **reequilibre-a**, caso seja necessário. Apresente a árvore final.

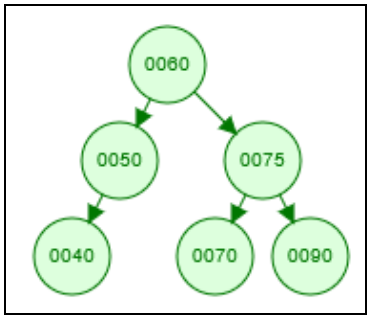


Remover o filho direito da raiz da árvore (nodo 80) implica substituir o elemento 80 por um dos seguintes elementos:

- 75 (maior elemento da subárvore com raiz em 80 menor que 80)
- 90 (menor elemento da subárvore com raiz em 80 maior que 80)

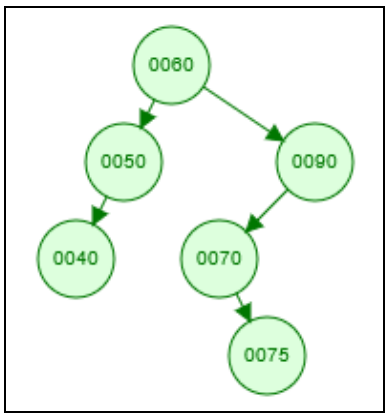
e de seguida remover o nodo substituto.

Caso 1: substituir pelo elemento 75



A árvore está equilibrada.

Caso 2: substituir pelo elemento 90



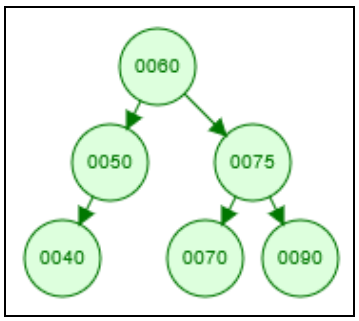
A árvore fica desequilíbrada no nodo **90**, pois

- o seu filho esquerdo (nodo 70) tem altura 1
- o seu filho direito (NULL) tem altura -1
- altura(filho esquerdo) – altura(filho direito) = 1 - (-1) = 2

Como

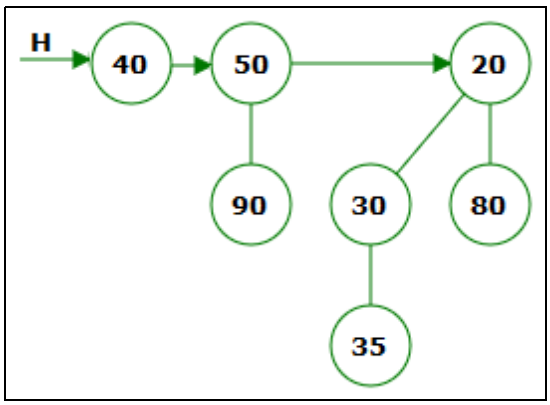
- o filho esquerdo do nodo 90 (70) tem mais altura que o filho direito do nodo 90 (NULL), e
- o filho direito do nodo 70 (75) tem mais altura que o filho esquerdo do nodo 70 (NULL),

então reequilibrar a árvore aplicando uma **rotação simples à esquerda** do nodo 90 usando o nodo 75



F. [1,50 val] Heaps

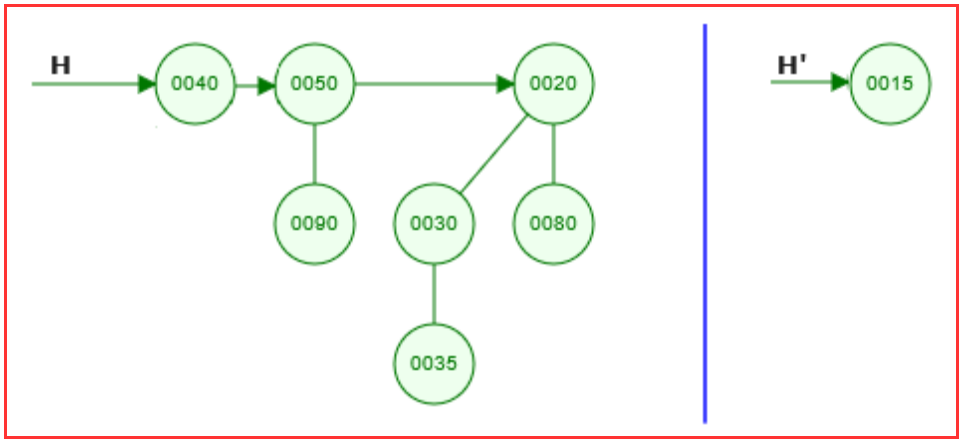
Considere o seguinte **heap Binomial H** com estrutura de minHeap:



1. Determine o resultado da operação **inserir** um elemento com o valor **15** no heap **H**. Justifique e apresente a evolução do heap desde a estrutura inicial (figura ao lado) até à final. Descreva o processo usado.

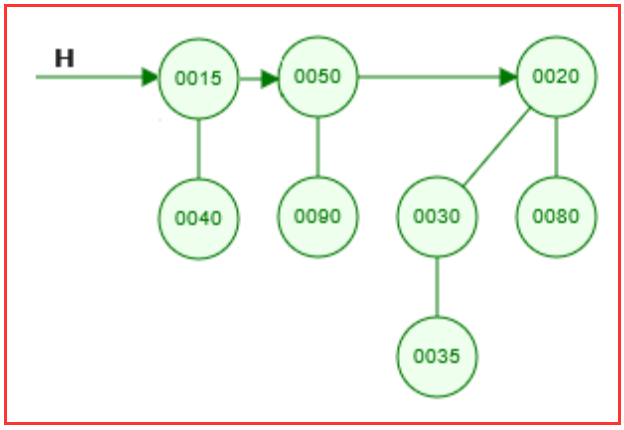
O processo é o seguinte:

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

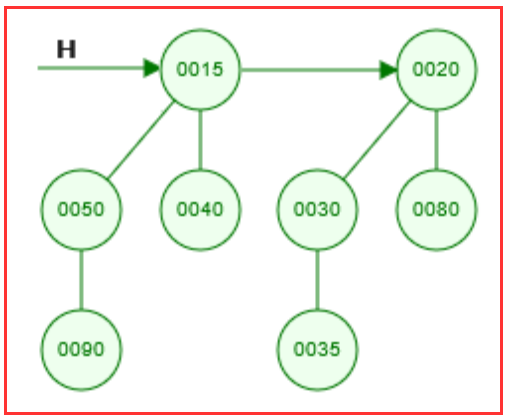


2º) aplicar a operação **união** entre os dois heaps, H e H':

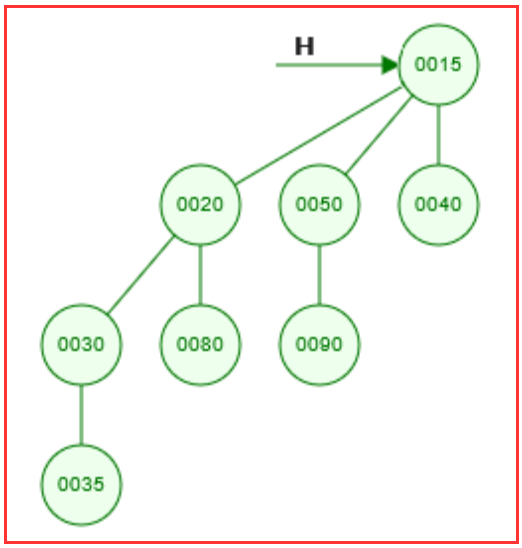
- como existem 2 árvores binomiais de grau 0 (B0) cujas raizes têm os valores 40 e 15, então aplicar a operação **união** entre árvores binomiais fomando uma árvore de grau 1 (B1) com raiz com o valor 15 (min{40, 15})



- como agora existem 2 árvores binomiais de grau 1 (B1) cujas raizes têm os valores 15 e 450, então aplicar a operação **união** entre árvores binomiais fomando uma árvore de grau 2 (B2) com raiz com o valor 15 (min{15, 40})

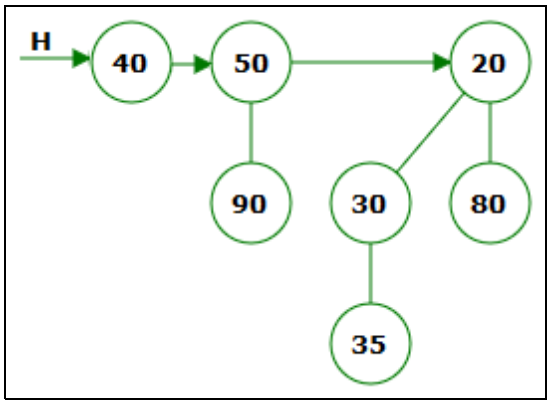


- como agora existem 2 árvores binomiais de grau 2 (B2) cujas raizes têm os valores 15 e 20, então aplicar a operação **união** entre árvores binomiais fomando uma árvore de grau 3 (B3) com raiz com o valor 15 (min{15, 20})



---

2. Determine o resultado da operação **"remover"** aplicado ao heap **H** (figura ao lado). Justifique e apresente a evolução do heap H desde a estrutura inicial (figura ao lado) até à final. Descreva o processo usado.



O processo é o seguinte:

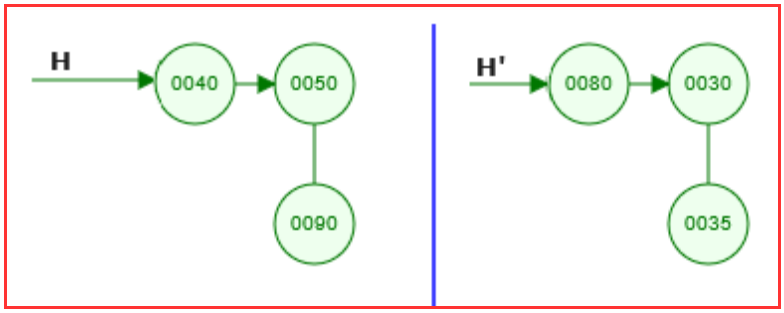
1º) extrair o menor valor entre as raizes do heap H: min{ 40, 50, 20 } = **20**

2º) remover o nodo raiz com o valor 20 (menor raiz),

- remover a árvore binomial com raiz com valor 20 do heap H, ficando H apenas com as árvores binomiais

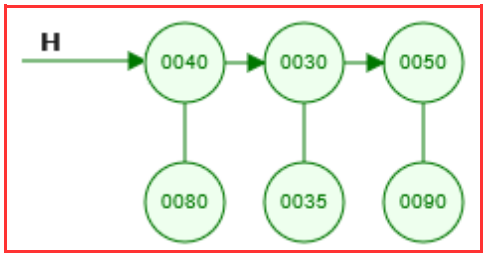
com raizes com os valores 40 (B0) e 50 (B1)

- criar um novo heap H' com os filhos da árvore com raiz com o valor 20

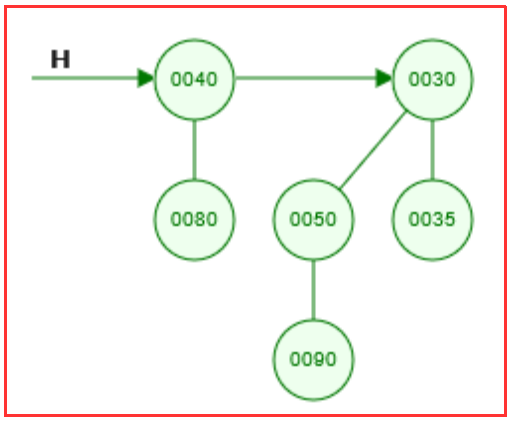


3º) aplicar a operação **união** entre os dois heaps, H e H':

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



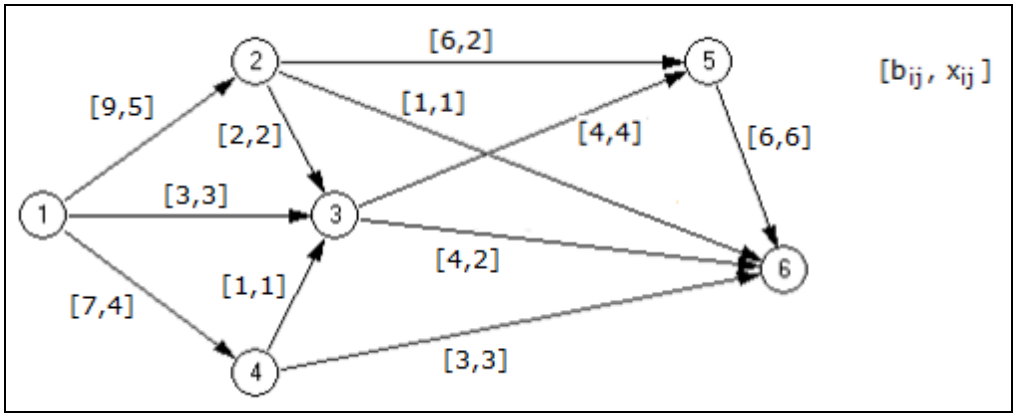
- como agora existem 3 árvores binomiais de grau 1 (B1) cujas raizes têm os valores 40, 30 e 50, então juntar as 2 árvores mais à direita formando uma árvore de grau 2 (B2) com raiz com o valor 30 (min{30, 50})



- a árvore resultante da operação "remover" é formada por 2 árvores binomiais de graus 1 (B1) e 2 (B2).

G. [3,00 val] Grafos

Considere a seguinte rede G = (A, N, C), onde o valor de cada arco corresponde ao seu custo ($\mathbf{C_{ij}}$):



1. Determine o fluxo atual da rede entre os nós 1 e 6. Justifique.

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

Como S = 1, fluxo que sai do nó 1 = x12 + x13 + x14 = 5 + 3 + 4 = 12

---

2. Determine o fluxo que atualmente sai do nó **2**. Determine o fluxo que chega ao nó **3**. Justifique.

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

---

3. Determine o **fluxo máximo** que pode ser enviado do nó **1** para o nó **6**, aplicando o algoritmo de **Ford-Fulkerson**. Simule este algoritmo e apresente os dados da simulação.

Apresente os dados da simulação, em especial: indicação dos **nós rotulados** e cálculo dos respetivos **rótulos**, indicação dos **nós rotulados e varridos**, evolução dos valores de **j** e de **k**.

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

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

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 <=> 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 < 6 SIM, logo R(5) = [2+, min{ 4, 6-2 }] = [2+, 4]

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{ 4, 4 }] = [5-, 4]

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{ 4, 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 já 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+, 4] => x25 = x25 + Q(6) = 2 + 2 = 4

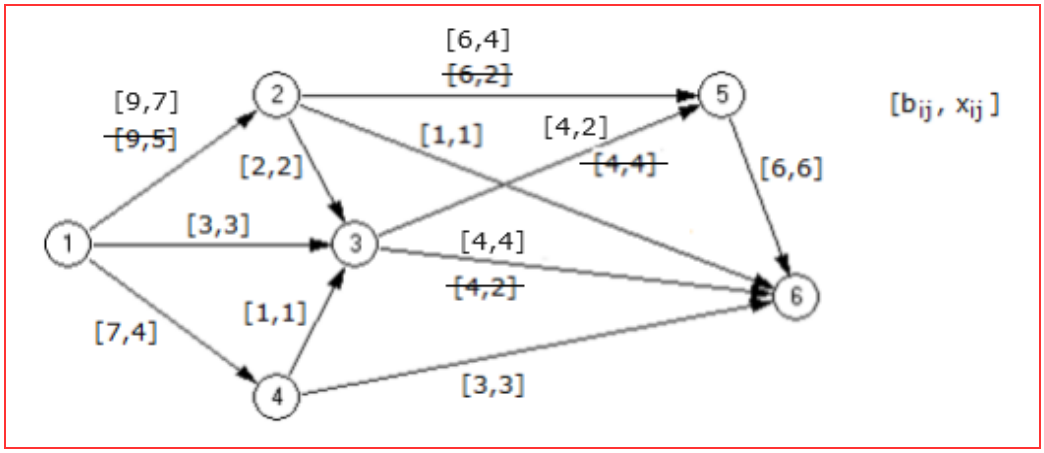
k = 2

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]	[5-, 2]	[1+, 3]	[2+, 2]	

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]

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 < 6 SIM, logo R(5) = [2+, min{ 4, 6-4 }] = [2+, 2]

6 --> x26 < b26 <=> 1 < 1 NÃO, logo não é possível rotular o nó 36

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 <=> 2 > 0 SIM, logo R(3) = [5-, min{ 2, 2 }] = [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 <=> 4 < 4 NÃO, logo não é possível rotular o nó 6

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 **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