



Universidade Federal do ABC
Centro de Matemática, Computação e Cognição

Heap

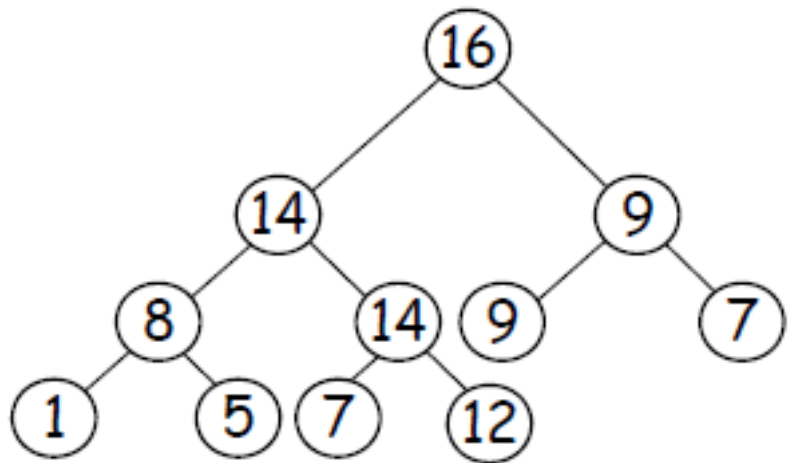
Monael Pinheiro Ribeiro, D.Sc.

Heap

- Definição

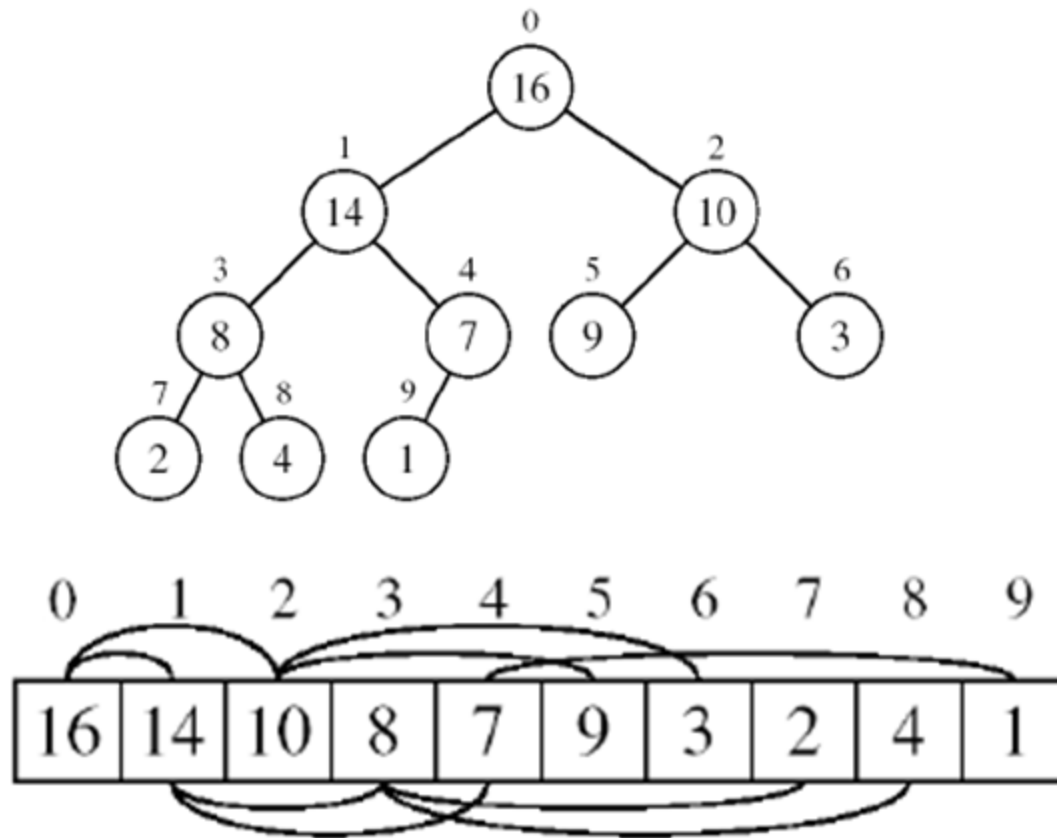
- Heap é uma árvore binária com duas propriedades:
 - Balanceamento: É uma árvore completa, com a eventual exceção do último nível, onde as folhas estão sempre nas posições mais à esquerda.
 - Estrutural: o valor armazenado em cada nó não é menor que os de seus filhos.

Há também o caso análogo, em que o valor de cada nó não é maior que os de seus filhos



Representação de Heap com vetores

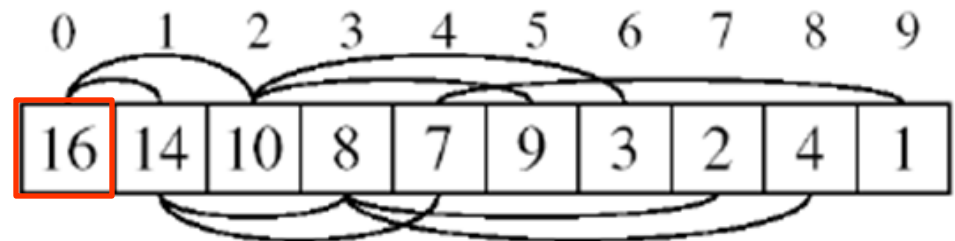
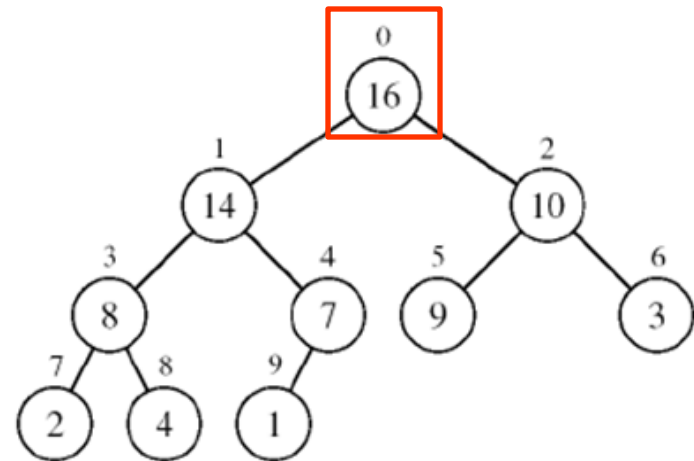
- Armazenamento de um heap com n elementos em um vetor v :



Relações de Heap representado em vetor

- Raiz está em $v[0]$:

A raiz da árvore está sempre no índice $i=0$.

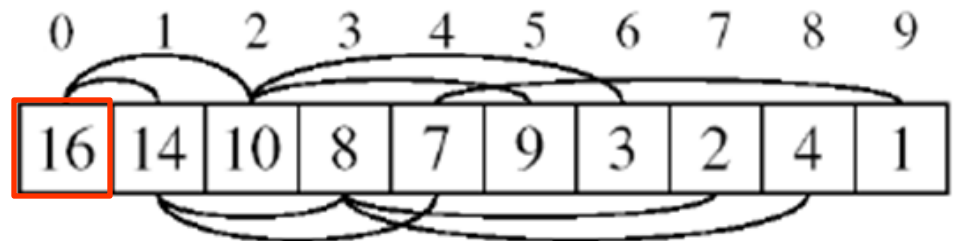
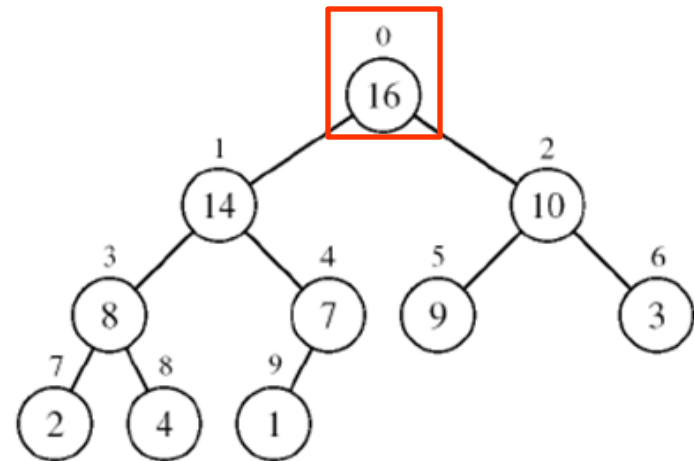


Relações de Heap representado em vetor

- Raiz está em $v[0]$:

A raiz da árvore está sempre no índice $i=0$.

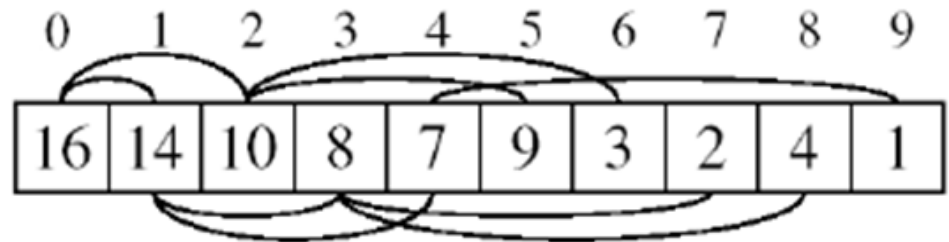
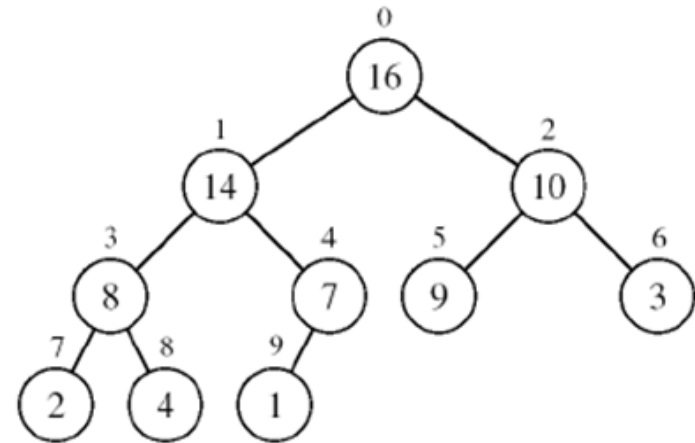
Uma implicação importante desta relação é que o **maior** elemento da coleção sempre estará no índice **0** do vetor.



Relações de Heap representado em vetor

- Filho Esquerdo:

O filho Esquerdo de um nó i está sempre no índice $2*i+1$.

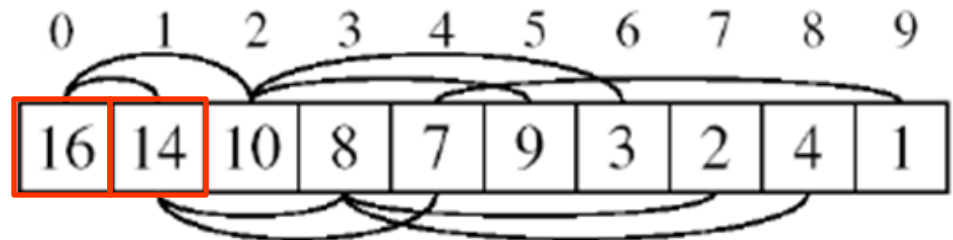
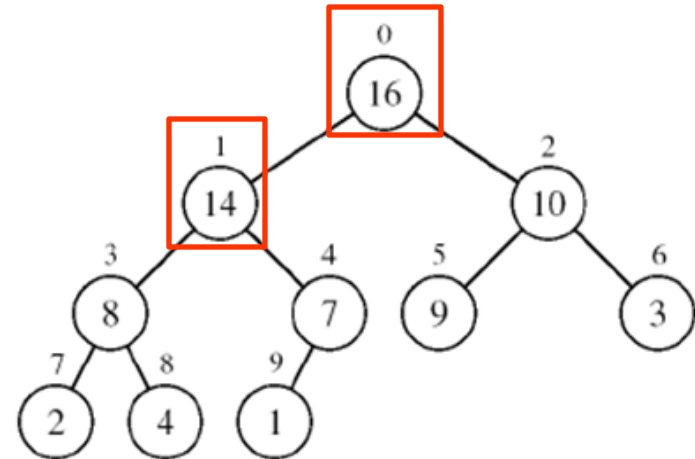


Relações de Heap representado em vetor

- Filho Esquerdo:

O filho Esquerdo de um nó i está sempre no índice $2*i+1$.

Filho Esquerdo de 0: $2*0+1 = 1$



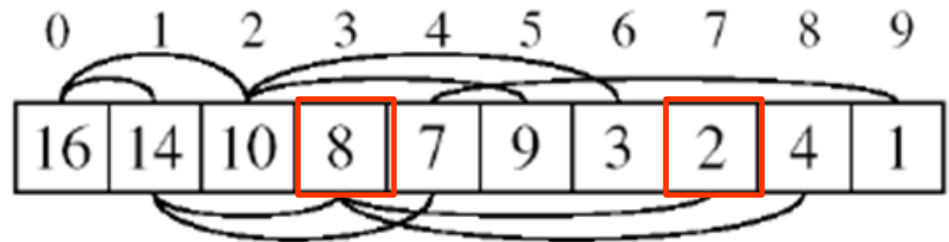
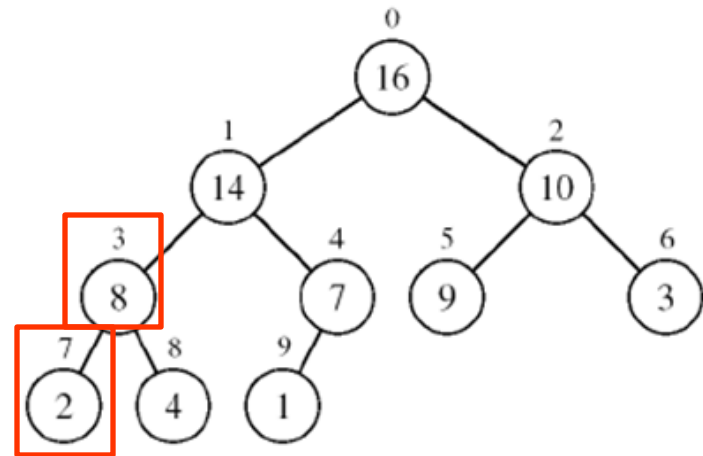
Relações de Heap representado em vetor

- Filho Esquerdo:

O filho Esquerdo de um nó i está sempre no índice $2*i+1$.

Filho Esquerdo de 0: $2*0+1 = 1$

Filho Esquerdo de 3: $2*3+1 = 7$

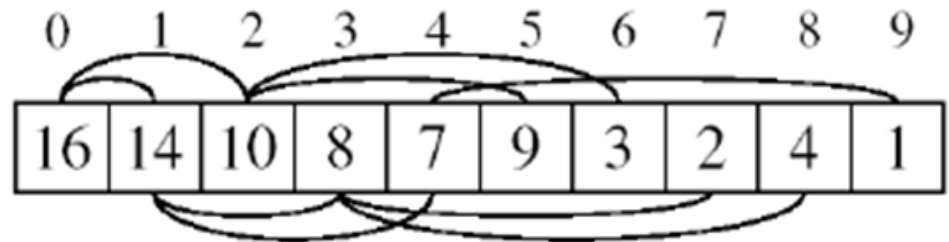
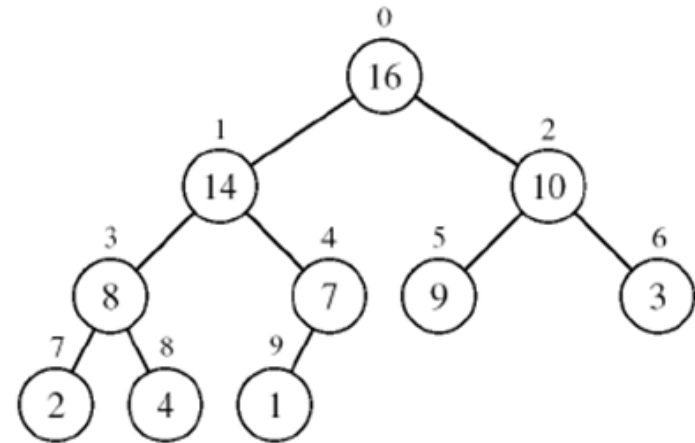


Relações de Heap representado em vetor

- Filho Direito:

O filho Direito de um nó i está sempre no índice $2*i+2$.

Filho Direito de 0: $2*0+2 = 2$

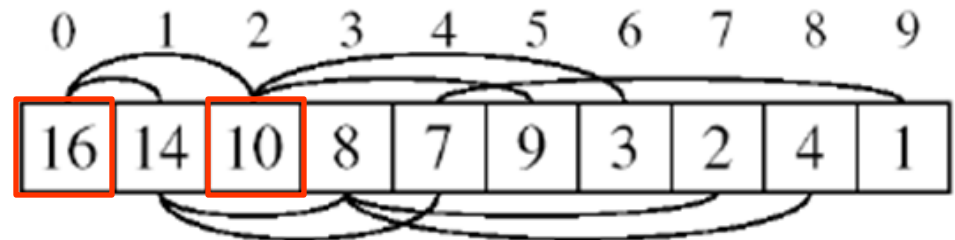
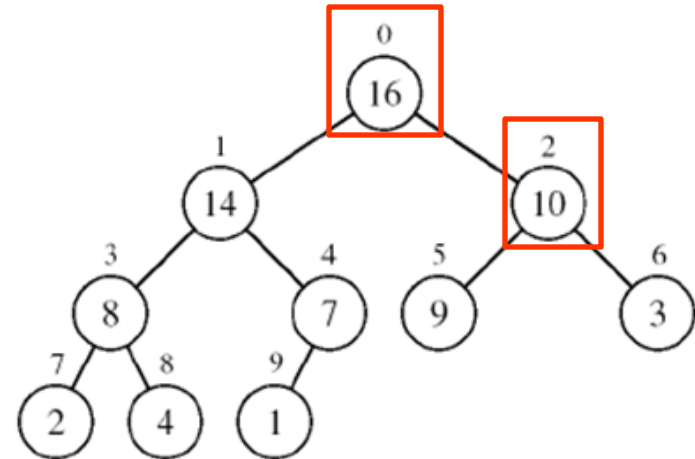


Relações de Heap representado em vetor

- Filho Direito:

O filho Direito de um nó i está sempre no índice $2*i+2$.

Filho Direito de 0: $2*0+2 = 2$



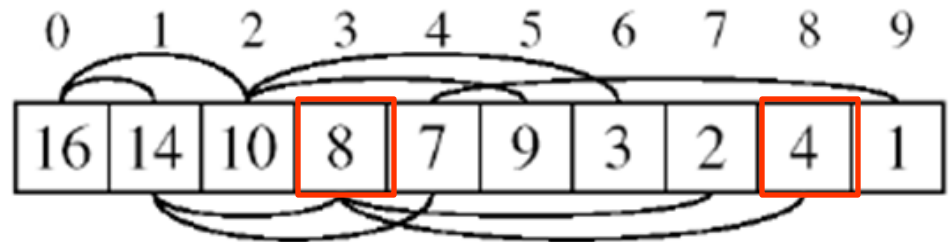
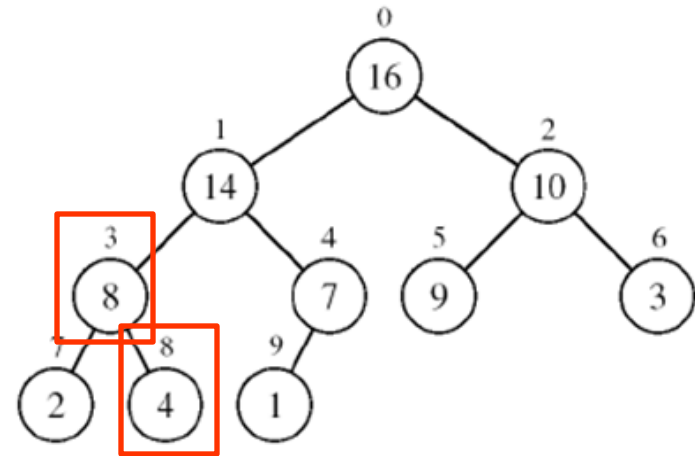
Relações de Heap representado em vetor

- Filho Direito:

O filho Direito de um nó i está sempre no índice $2*i+2$.

Filho Direito de 0: $2*0+2 = 2$

Filho Direito de 3: $2*3+2 = 8$

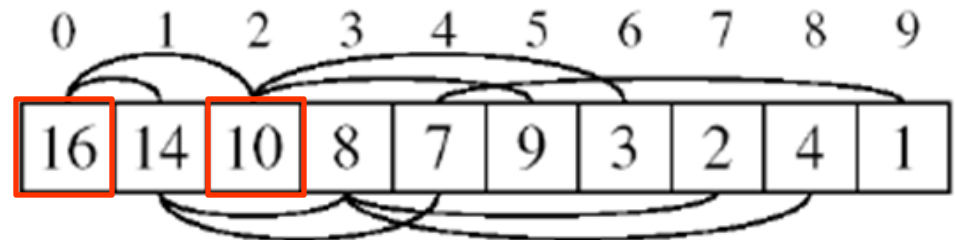
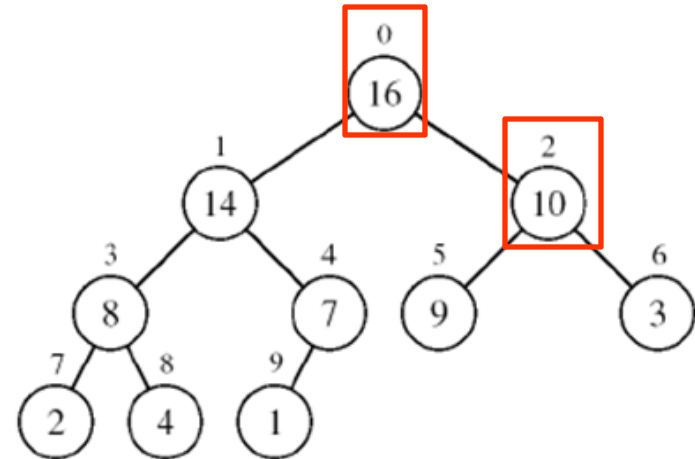


Relações de Heap representado em vetor

- Pai:

O pai de um nó i está sempre no índice $\lfloor (i-1)/2 \rfloor$.

Pai de 2: $\lfloor (2-1)/2 \rfloor = 0$



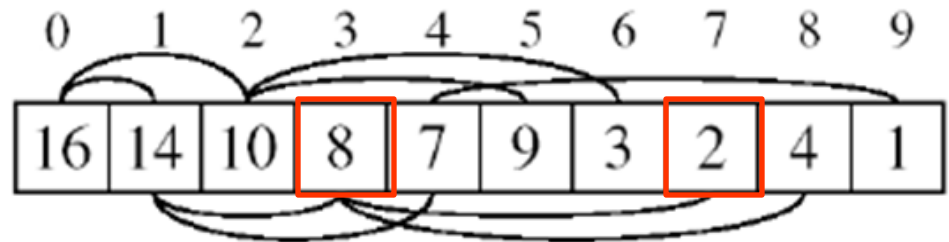
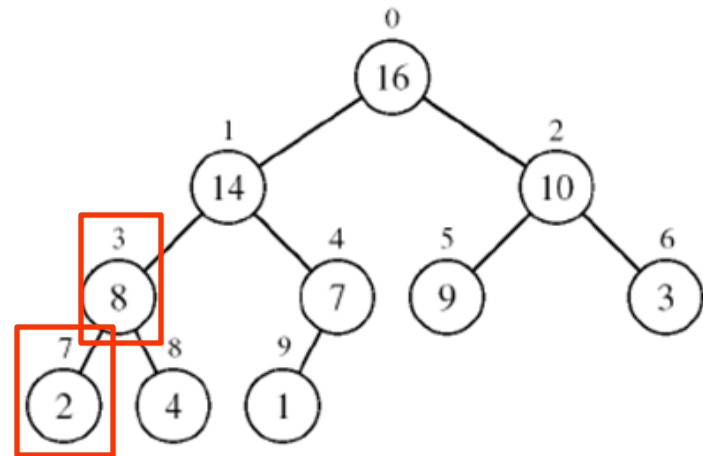
Relações de Heap representado em vetor

- Pai:

O pai de um nó i está sempre no índice $\lfloor (i-1)/2 \rfloor$.

Pai de 2: $\lfloor (2-1)/2 \rfloor = 0$

Pai de 7: $\lfloor (7-1)/2 \rfloor = 3$

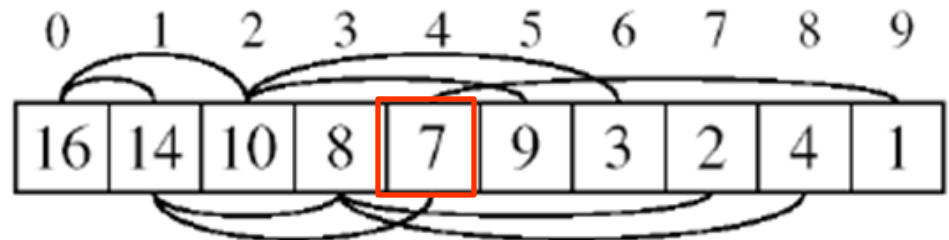
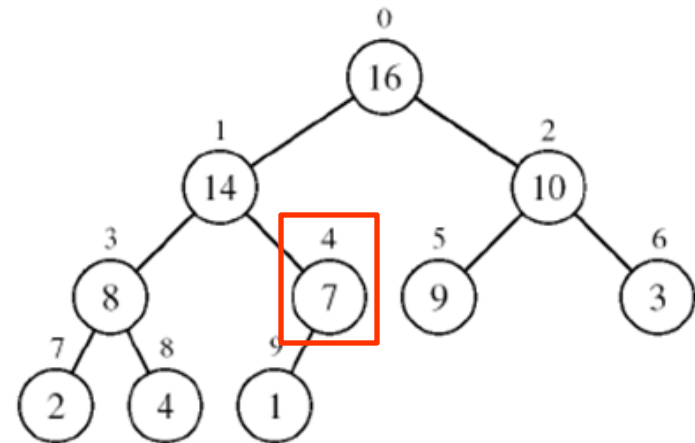


Relações de Heap representado em vetor

- Último Pai:

O elemento que é último pai da árvore sempre está no índice $\lfloor n/2 \rfloor - 1$:

Para a árvore exemplo, temos $n=10$: último pai: $\lfloor 10/2 \rfloor - 1 = 4$

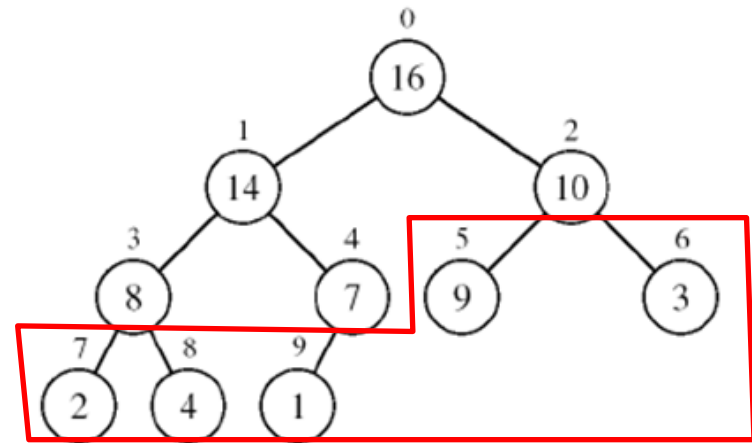


Relações de Heap representado em vetor

- Último Pai:

O elemento que é último pai da árvore sempre está no índice $\lfloor n/2 \rfloor - 1$:

Para a árvore exemplo, temos $n=10$: último pai: $\lfloor 10/2 \rfloor - 1 = 4$

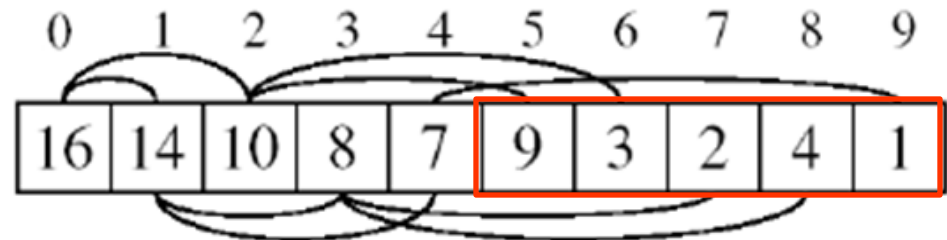


- Folhas:

Consequência da definição de último pai, temos que qualquer nó com índice i :

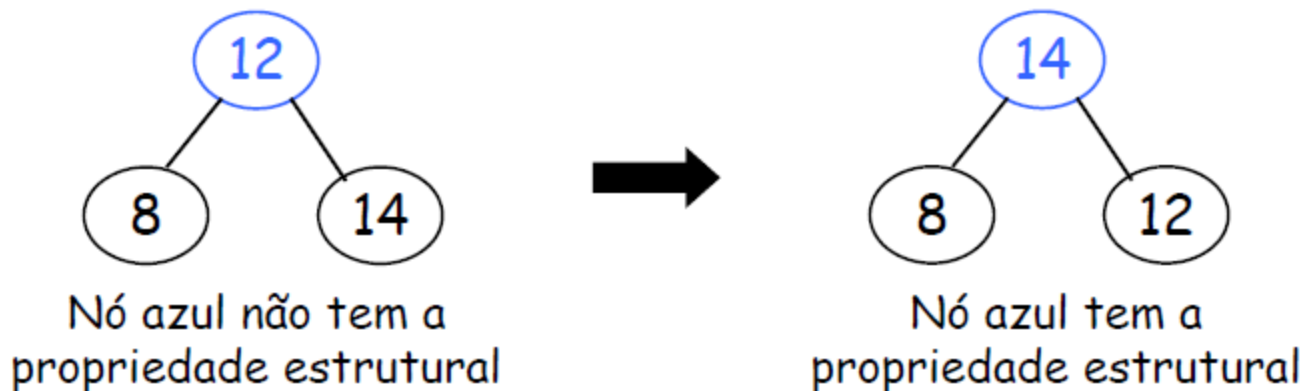
$$\lfloor n/2 \rfloor \leq i < n$$

trata-se de nó folha.



Perda da Propriedade Estrutural

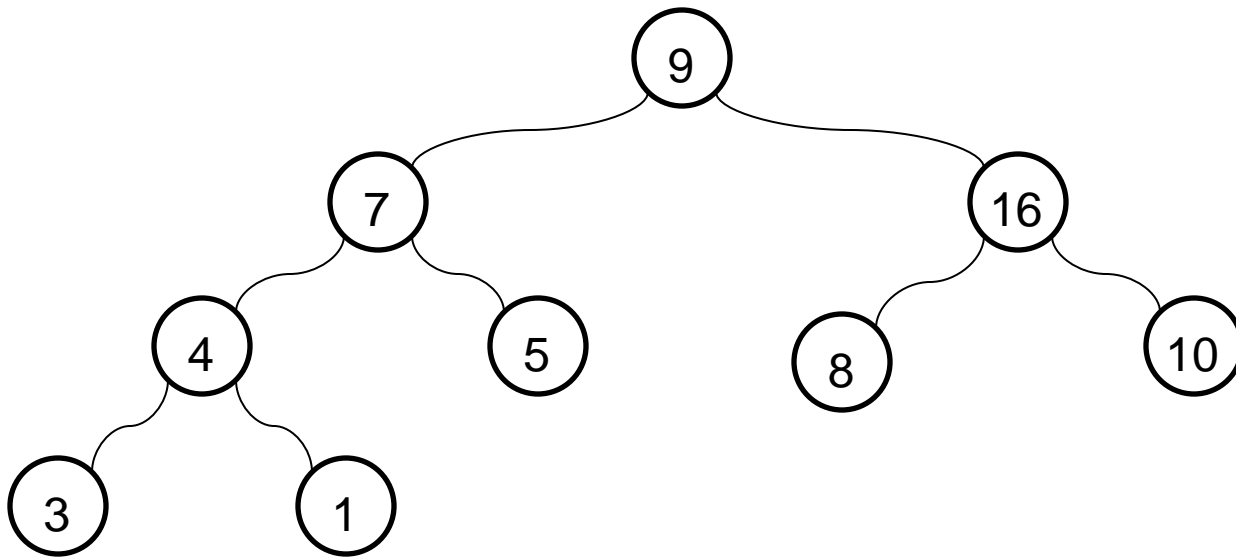
- Caso um nó de um heap perca a sua propriedade estrutural, poderá recuperá-la trocando de valor com o seu filho maior.
- Isso pode ser feito através do algoritmo PENEIRAR (Sift).



- Uma vez que o filho trocou de lugar com o pai, a subárvore que protagonizou a troca pode ter perdido a propriedade estrutural de heap, e também precisará invocar PENEIRAR para ela.

Peneirar

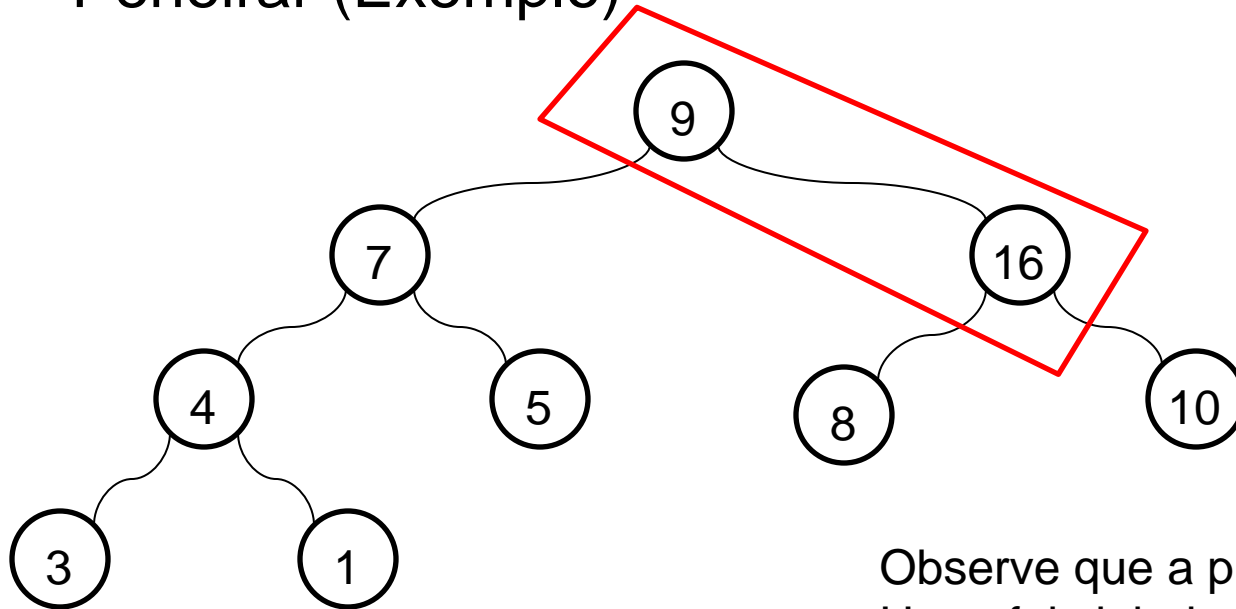
- Peneirar (Exemplo)



0	1	2	3	4	5	6	7	8
9	7	16	4	5	8	10	3	1

Peneirar

- Peneirar (Exemplo)

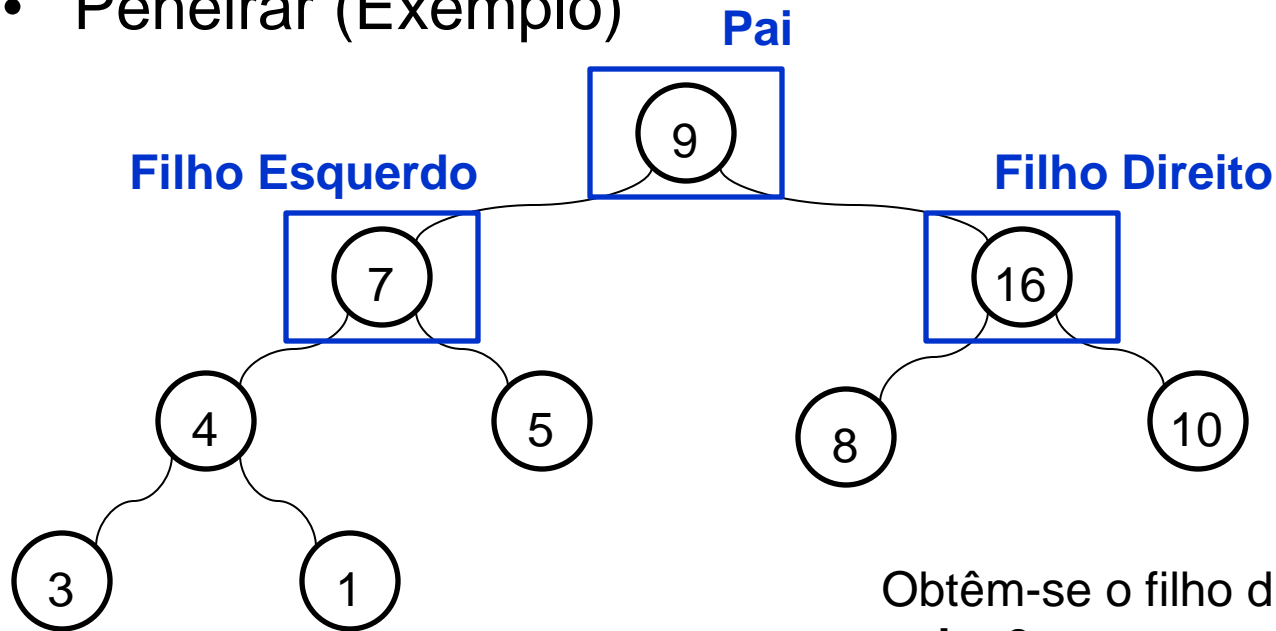


Observe que a propriedade estrutural de Heap foi violada no ponto destacado. Deste modo, invoca-se **Peneirar** para **pai = 0**

0	1	2	3	4	5	6	7	8
9	7	16	4	5	8	10	3	1

Peneirar

- Peneirar (Exemplo)

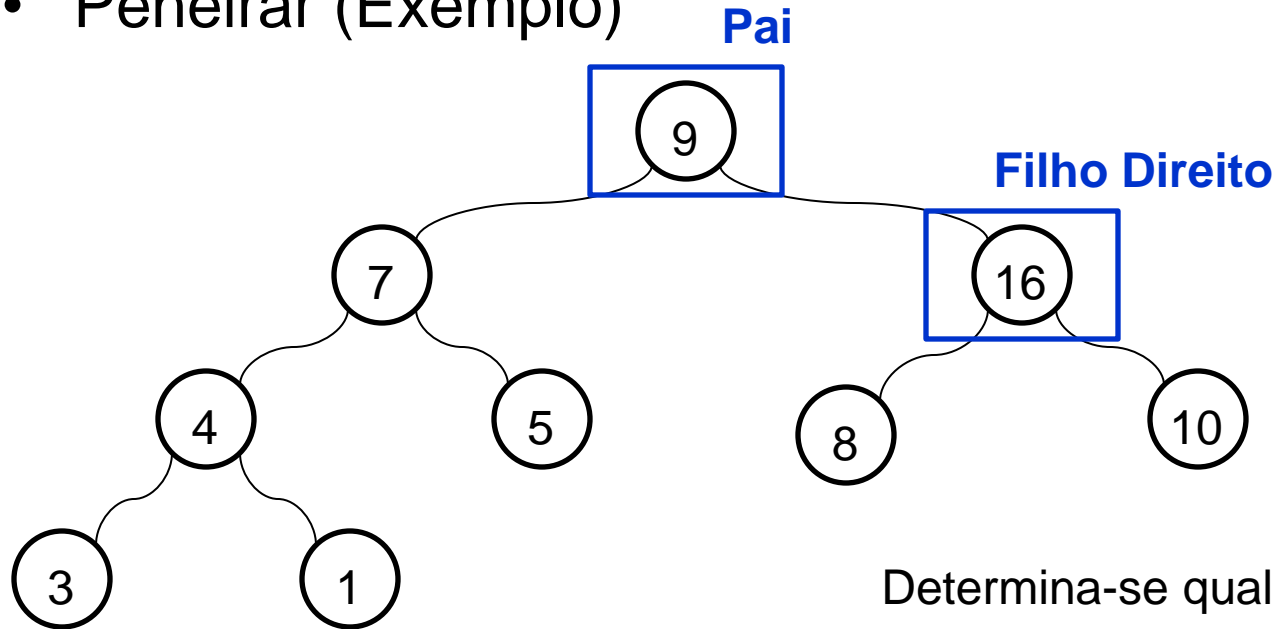


Obtêm-se o filho direito e esquerdo de
pai = 0

0	1	2	3	4	5	6	7	8
9	7	16	4	5	8	10	3	1

Peneirar

- Peneirar (Exemplo)

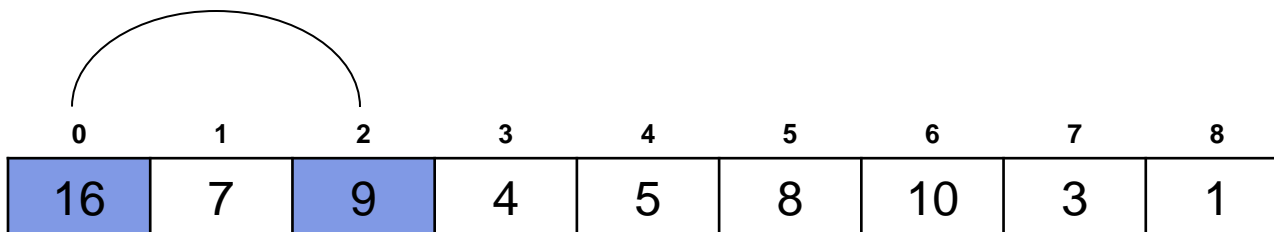
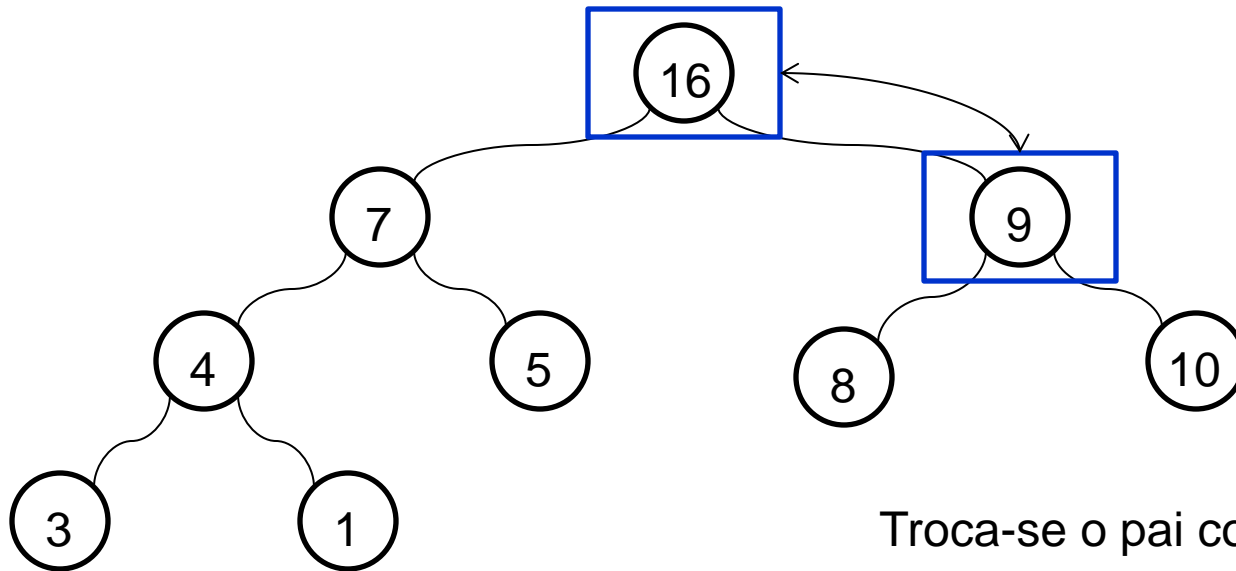


Determina-se qual dos filhos é maior.

0	1	2	3	4	5	6	7	8
9	7	16	4	5	8	10	3	1

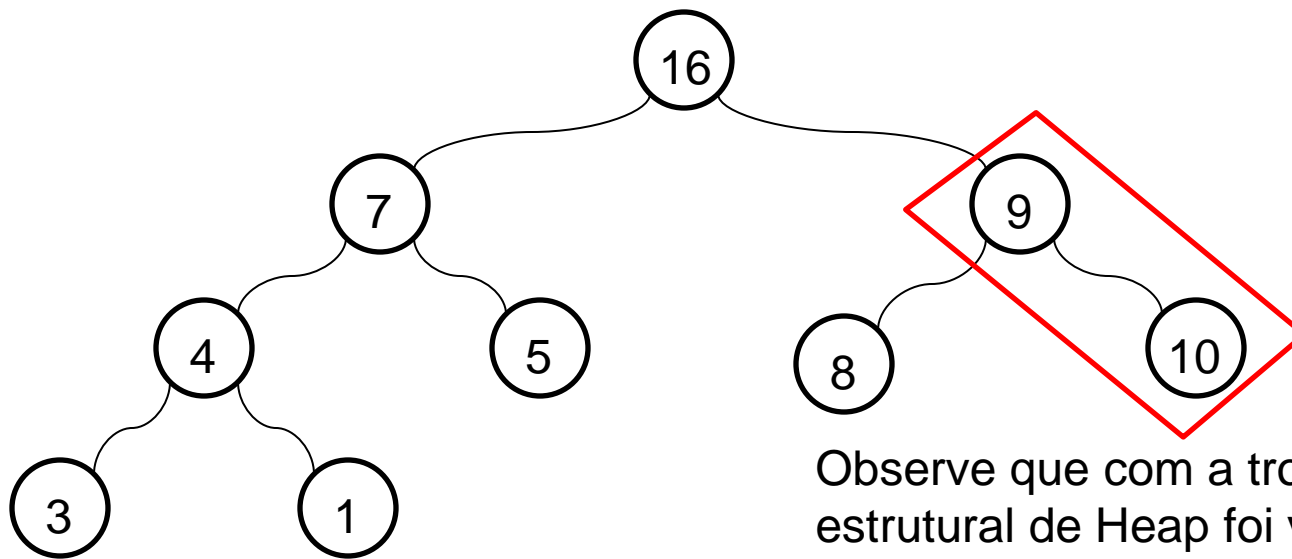
Peneirar

- Peneirar (Exemplo)



Peneirar

- Peneirar (Exemplo)



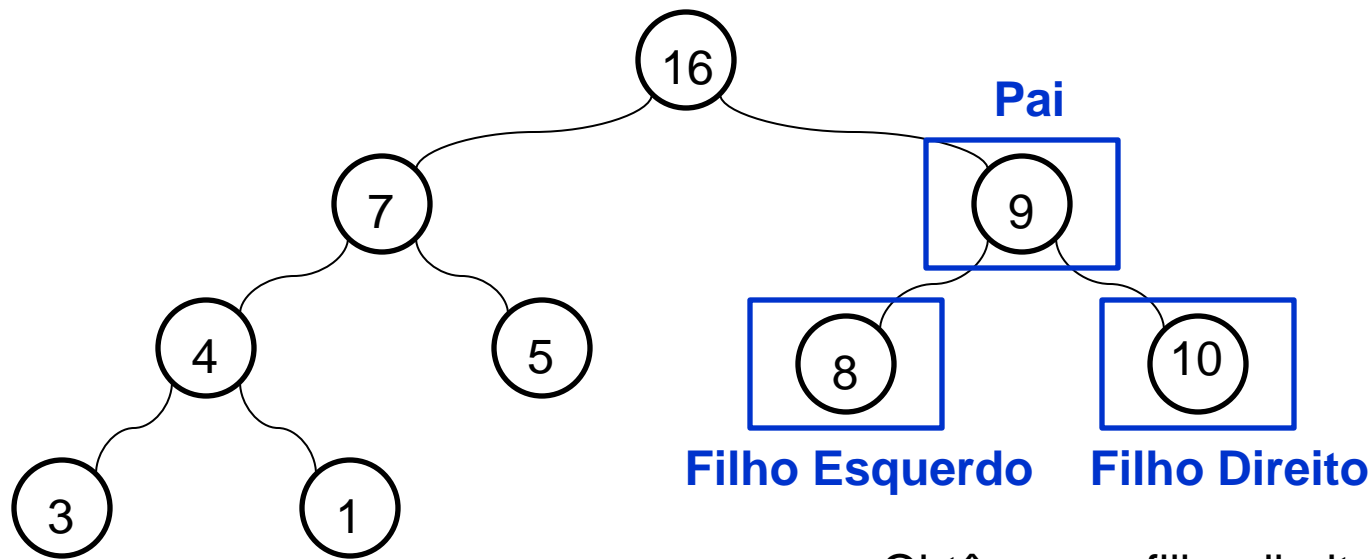
Observe que com a troca, propriedade estrutural de Heap foi violada exatamente na subárvore que promoveu a troca.

Deste modo, invoca-se **Peneirar** com pai igual onde ocorreu a troca, no caso, em **pai = filhoDireito = 2**

0	1	2	3	4	5	6	7	8
16	7	9	4	5	8	10	3	1

Peneirar

- Peneirar (Exemplo)

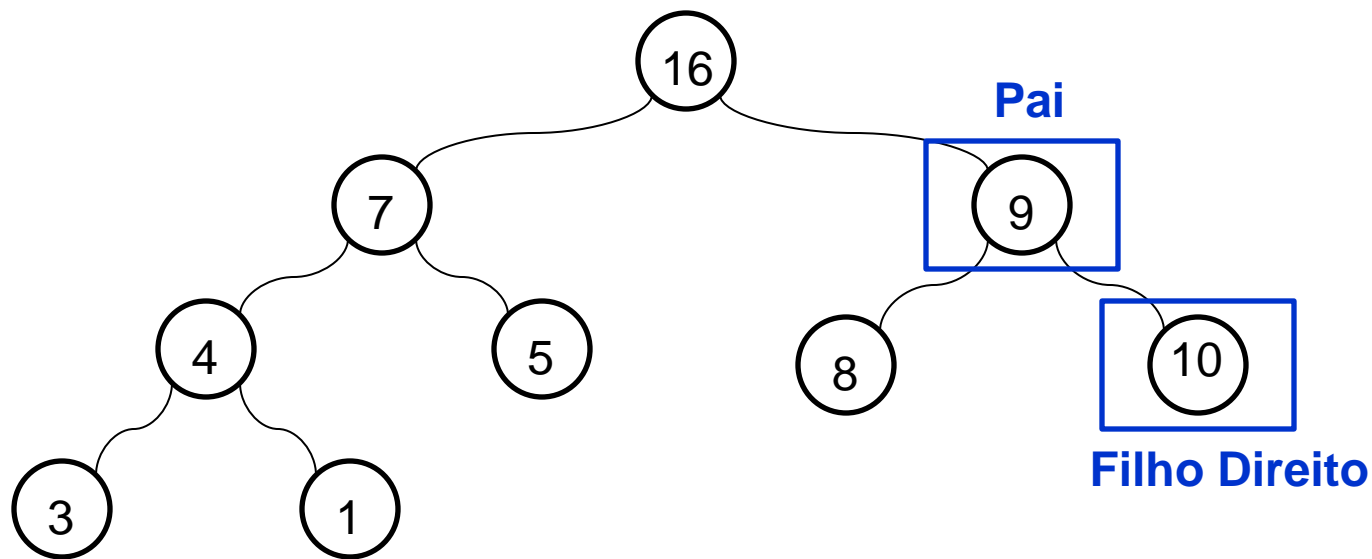


Obtêm-se o filho direito e esquerdo de
pai = 2

0	1	2	3	4	5	6	7	8
16	7	9	4	5	8	10	3	1

Peneirar

- Peneirar (Exemplo)

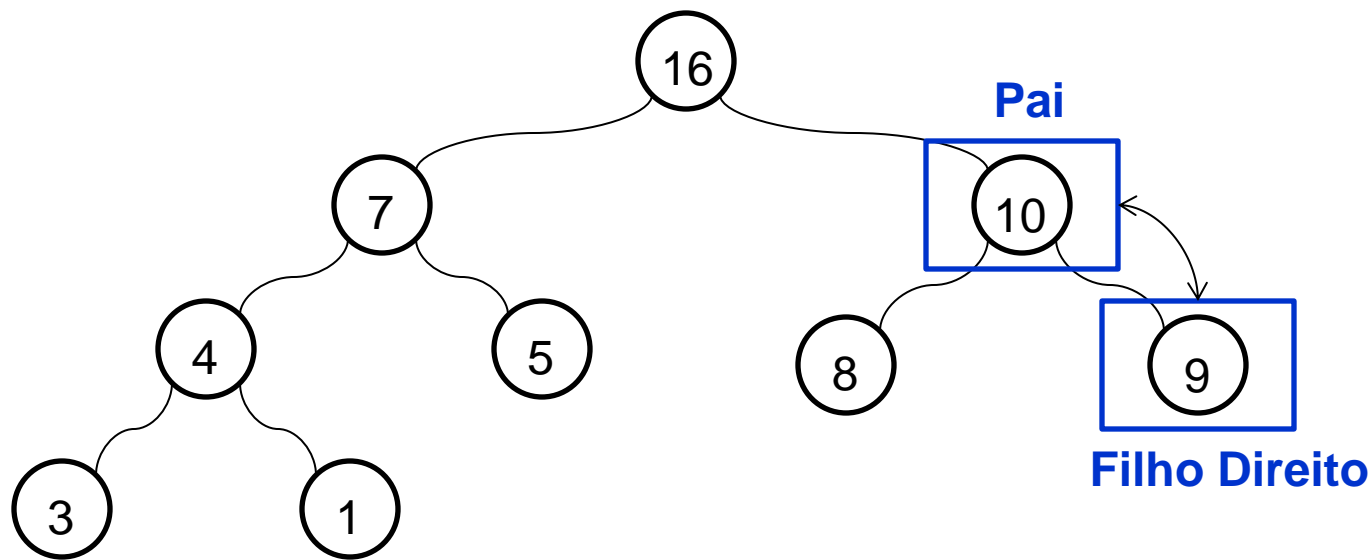


Determina-se qual dos filhos é maior.

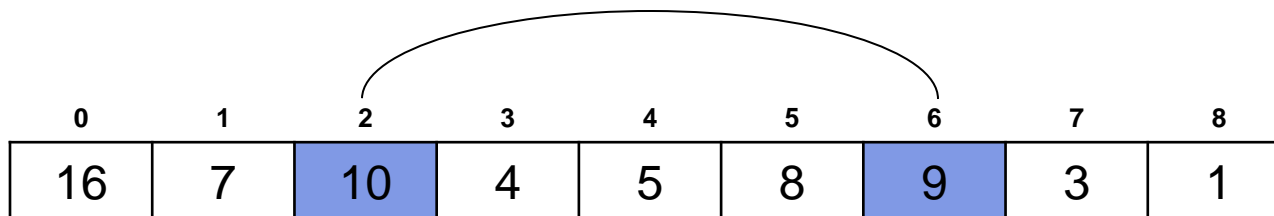
0	1	2	3	4	5	6	7	8
16	7	9	4	5	8	10	3	1

Peneirar

- Peneirar (Exemplo)

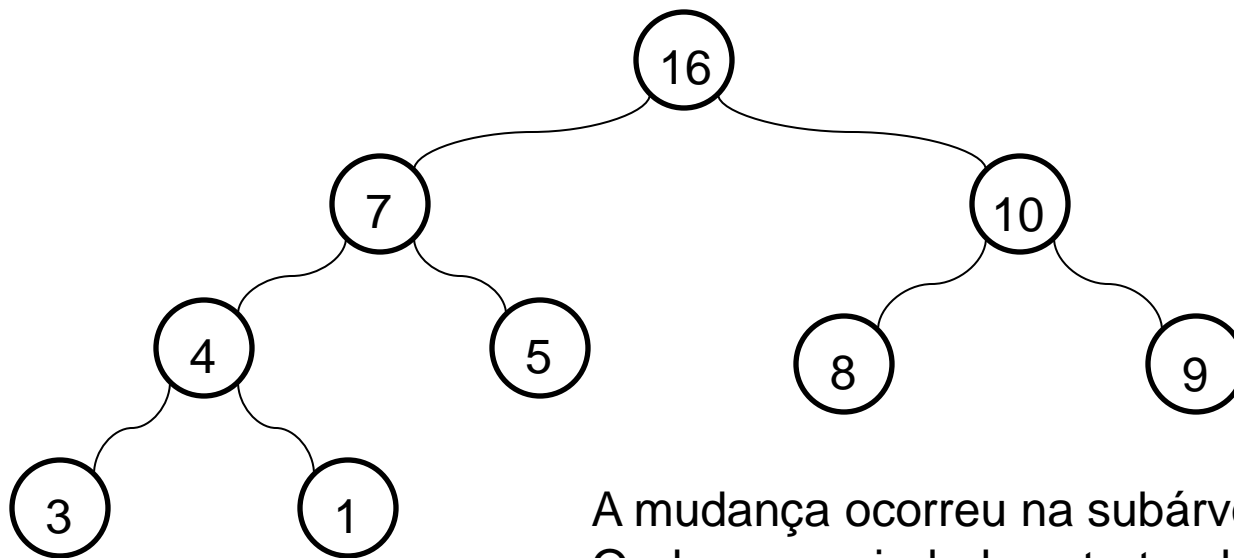


Determina-se qual dos filhos é maior.



Peneirar

- Peneirar (Exemplo)

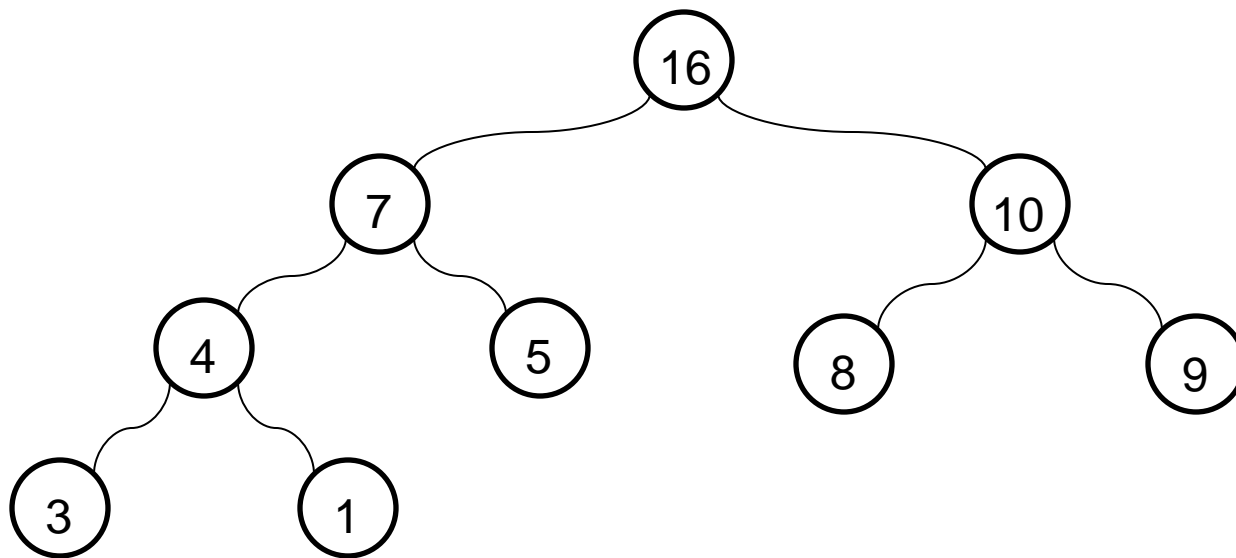


A mudança ocorreu na subárvore do **filhoDireito = 6**. Onde a propriedade estrutural poderia ser violada novamente, então invoca-se **Peneirar** para **pai = FilhoDireito = 6**, o que não resultará em nova chamada, pois o nó **6** é uma folha.

0	1	2	3	4	5	6	7	8
16	7	10	4	5	8	9	3	1

Peneirar

- Peneirar (Exemplo)

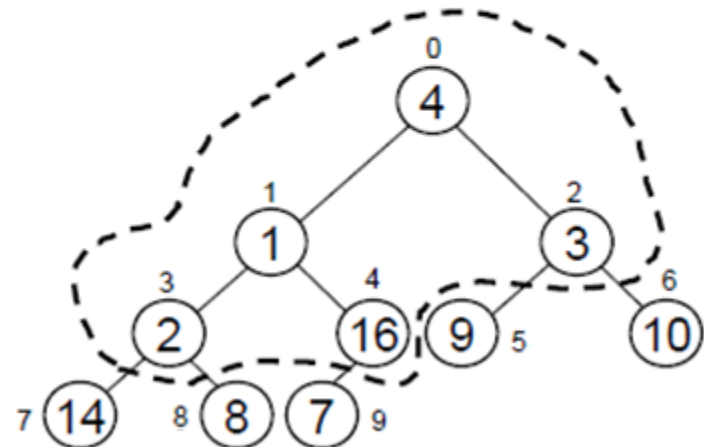
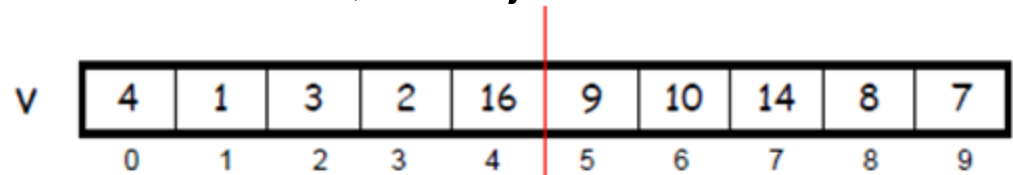


O que significa que o vetor resultante é um Heap!

0	1	2	3	4	5	6	7	8
16	7	10	4	5	8	9	3	1

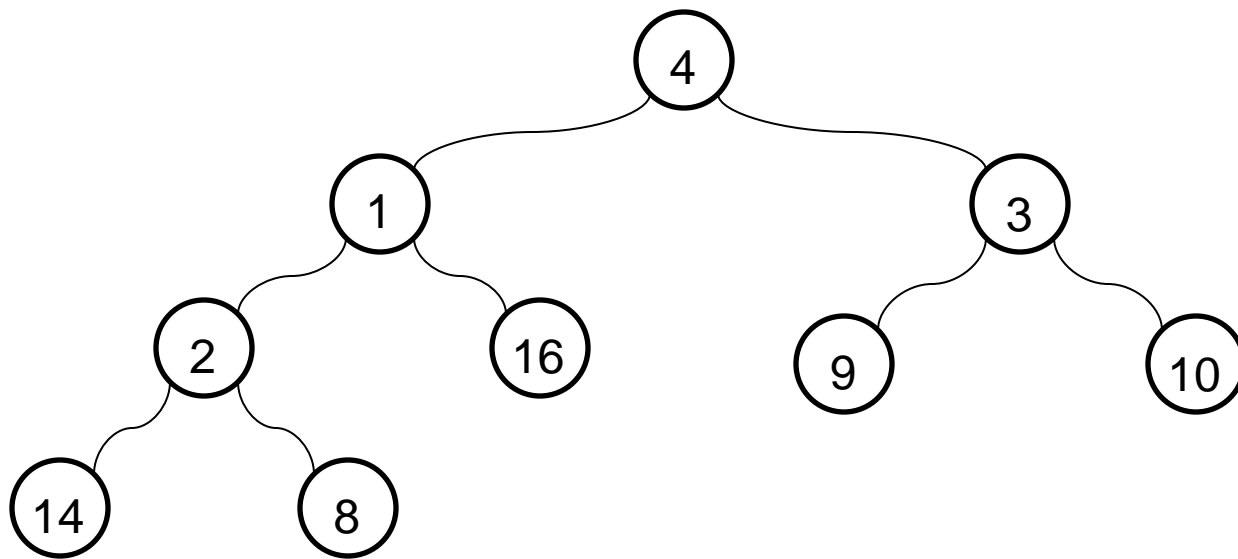
Construção de um Heap

- Construir um Heap a partir de um vetor qualquer:
 - O algoritmo Construir transforma um vetor qualquer em um heap.
 - Como os índices i , $\lfloor n/2 \rfloor \leq i < n$, são folhas, basta aplicar **Peneirar** entre as posições 0 e $\lfloor n/2 \rfloor - 1$, ou seja em todos os nós que são pais.



Construir

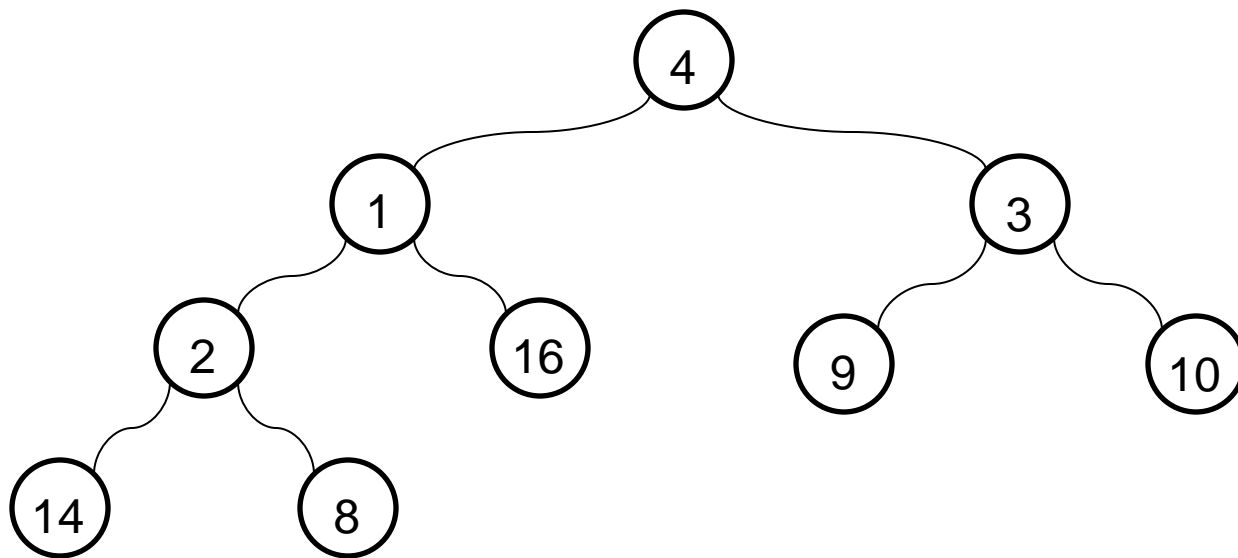
- Construir (Exemplo)



0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

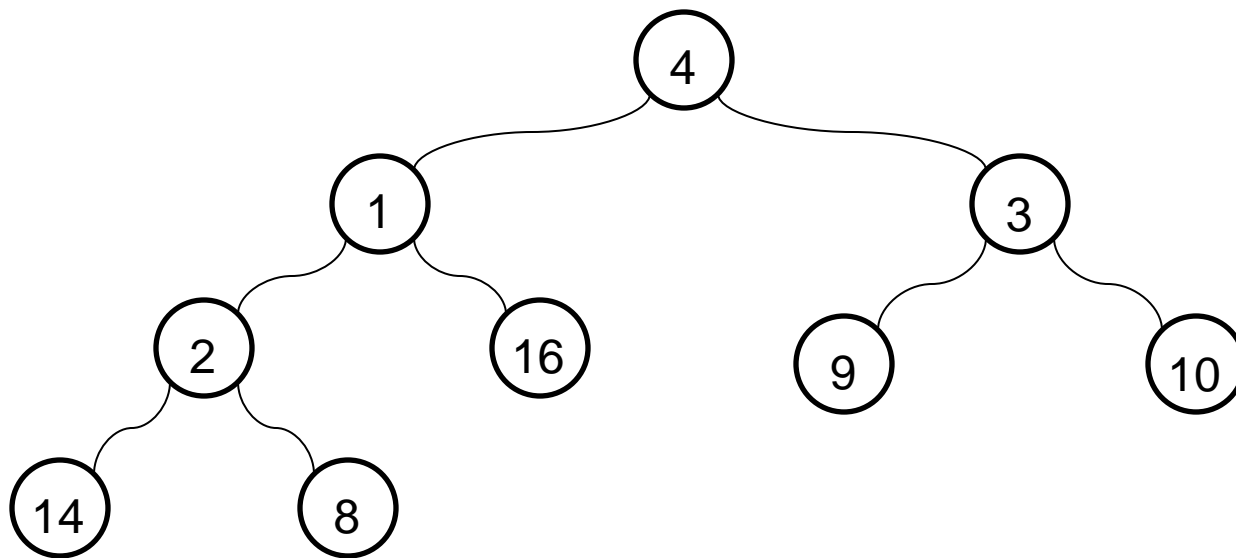


O que fazer?

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

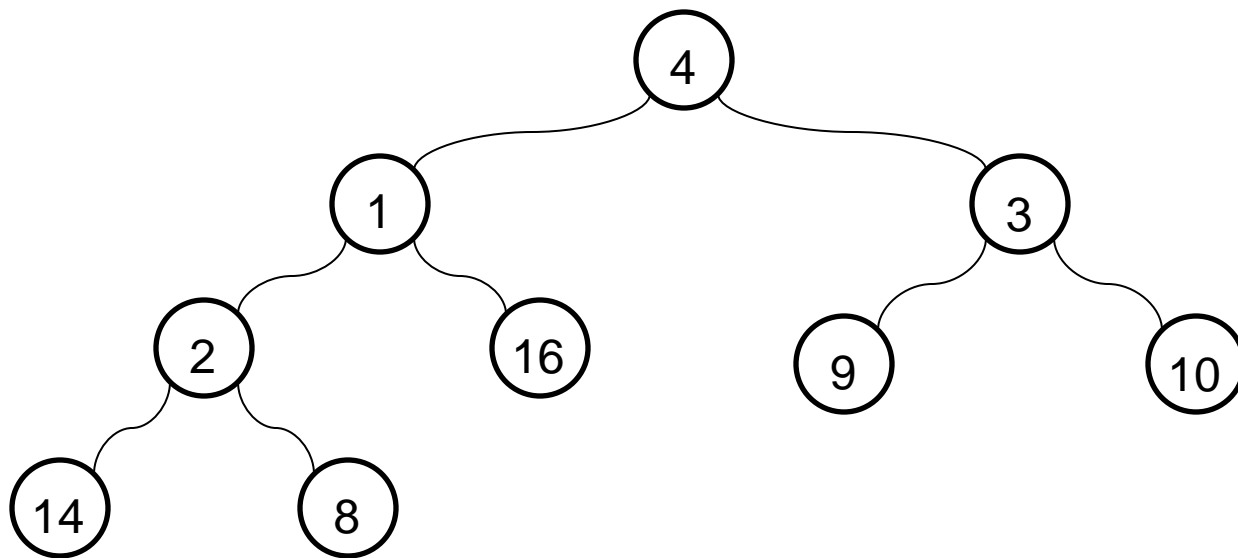


O que fazer?
Aplica-se Peneirar em todos
os nós pais.

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

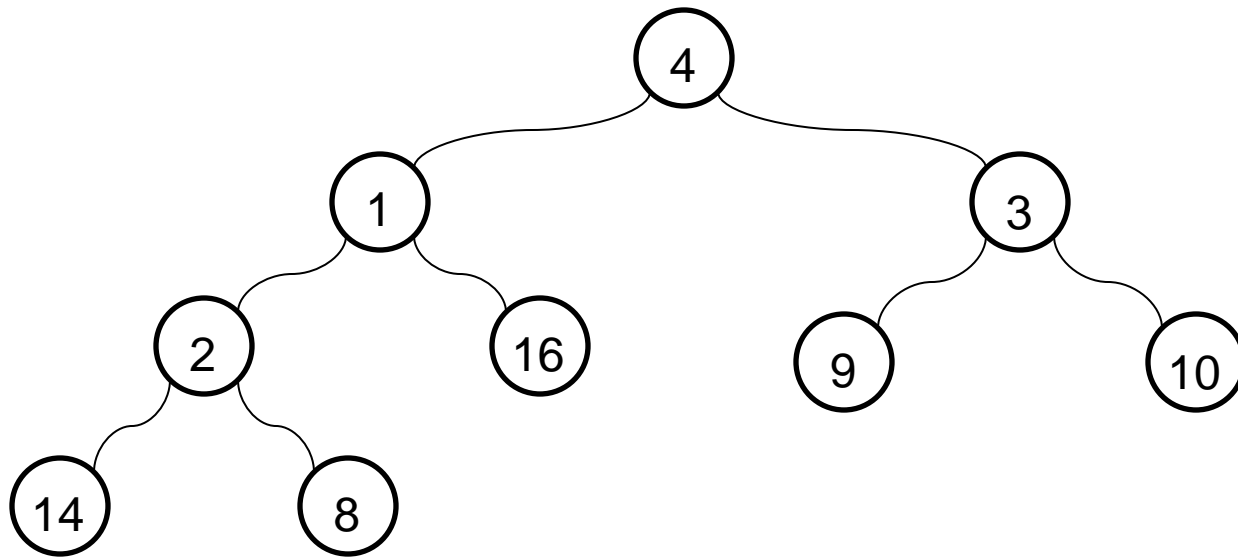


Começando onde ?

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

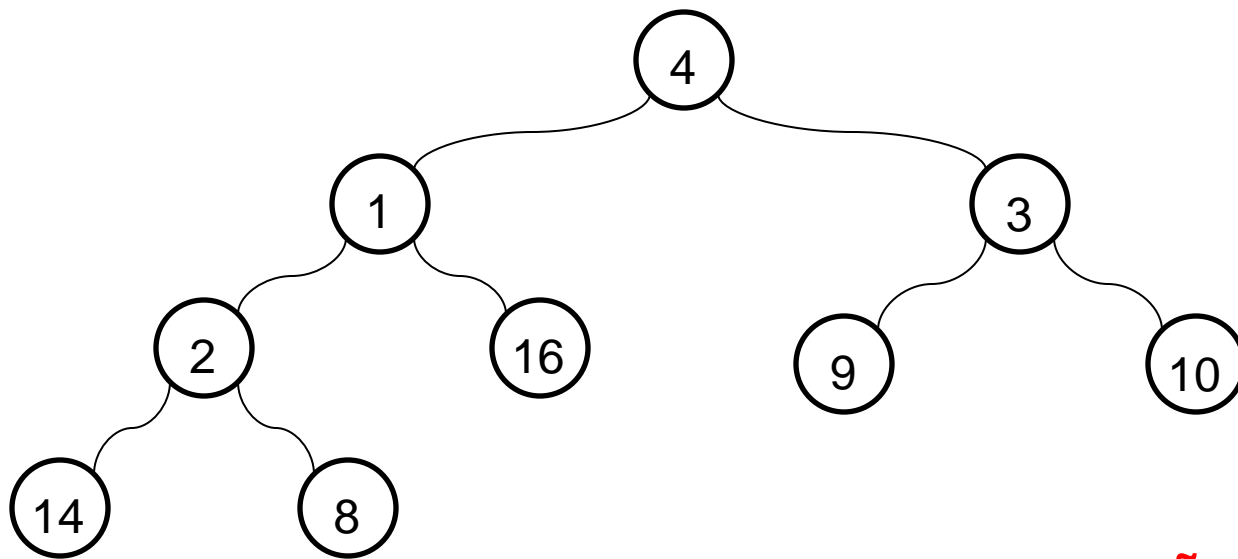


Começando onde ?
Na Raiz!

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

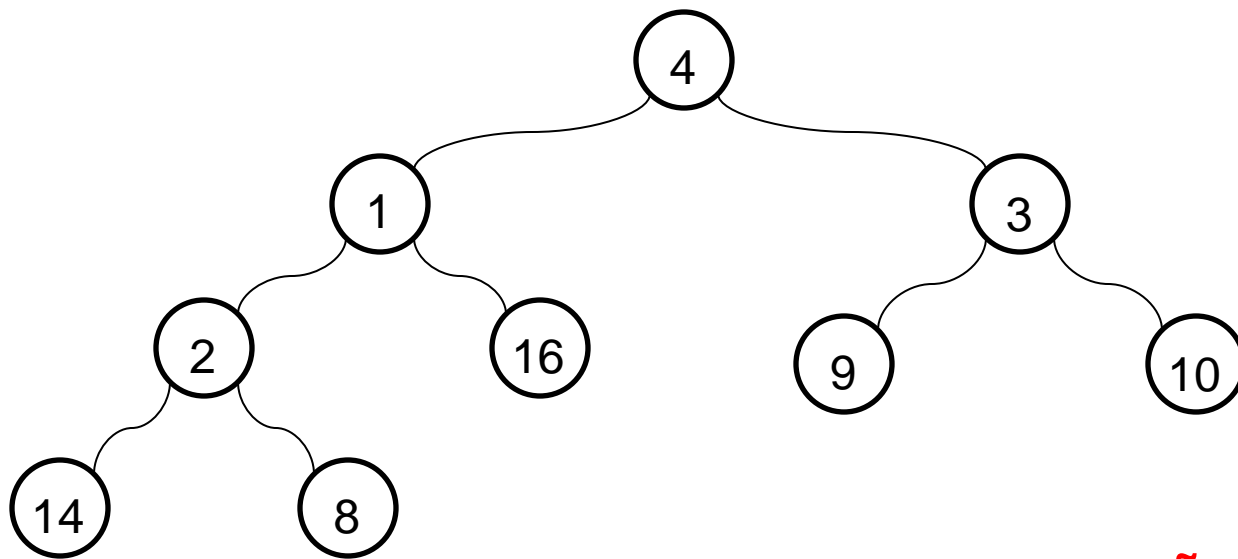


NÃO!!!

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

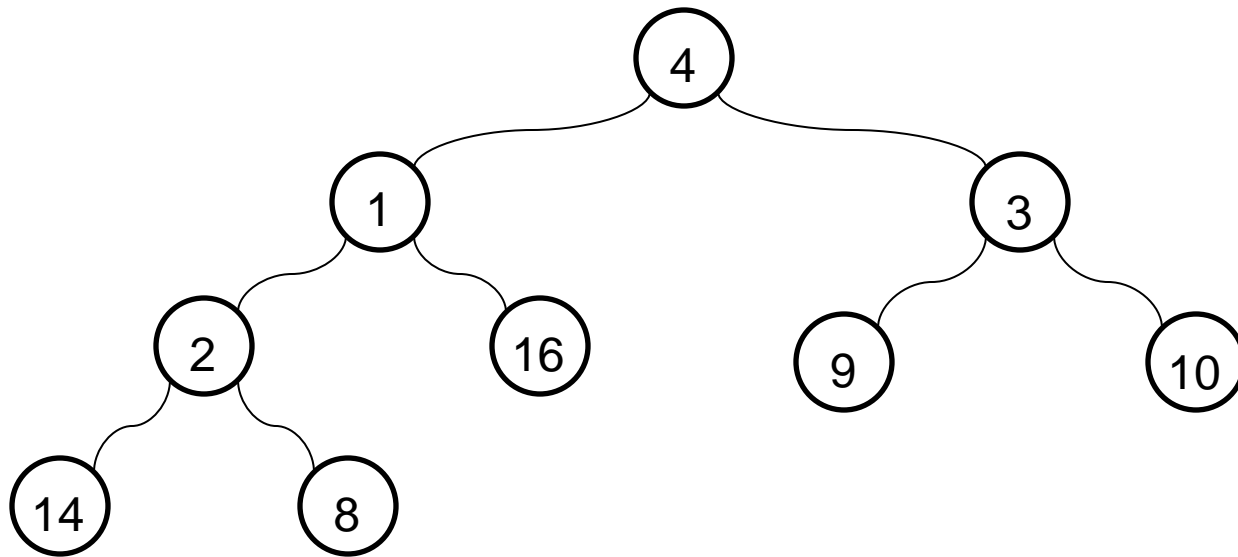


**NÃO!!!
POR QUE?**

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

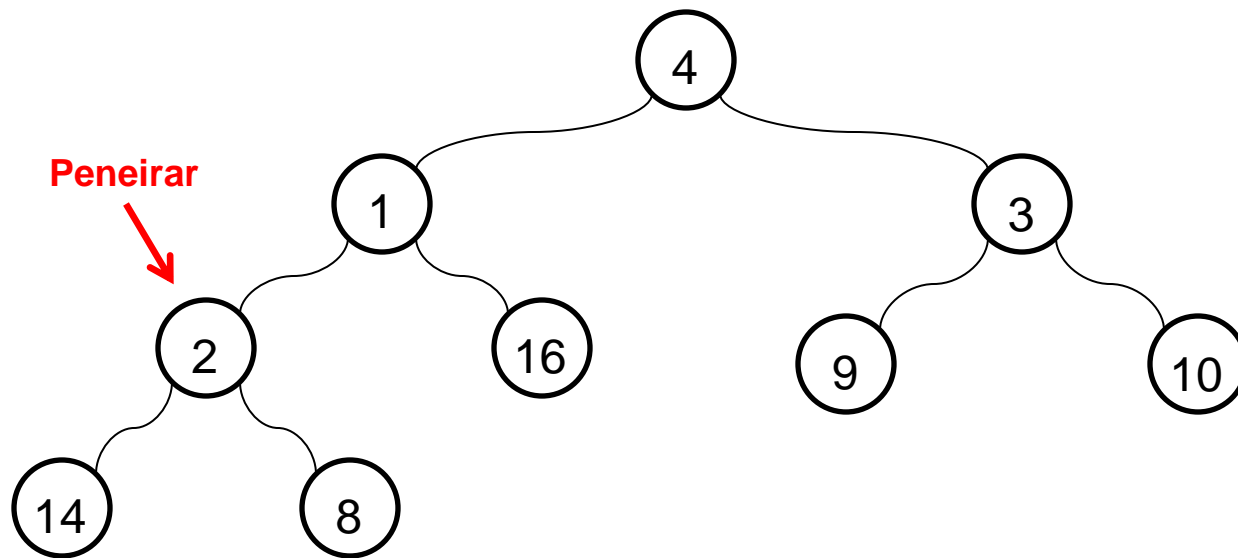


Parte-se do **Último Pai = 3**, até a raiz invocando-se a função **Peneirar**.

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

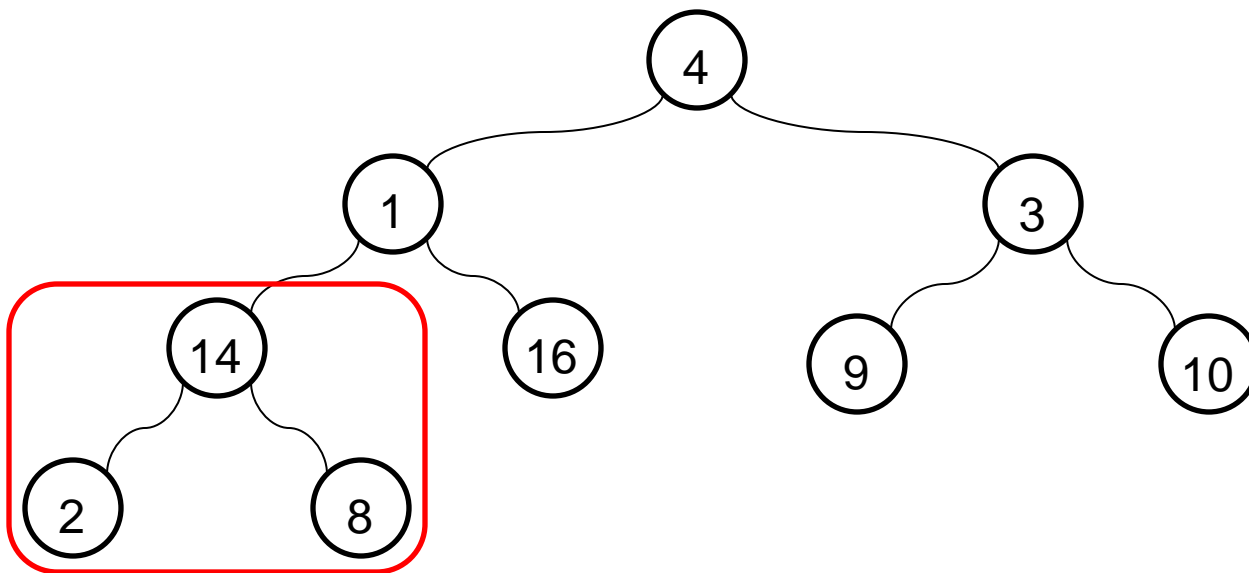


Invoca-se **Peneirar** para **pai = 3**.

0	1	2	3	4	5	6	7	8
4	1	3	2	16	9	10	14	8

Construir

- Construir (Exemplo)

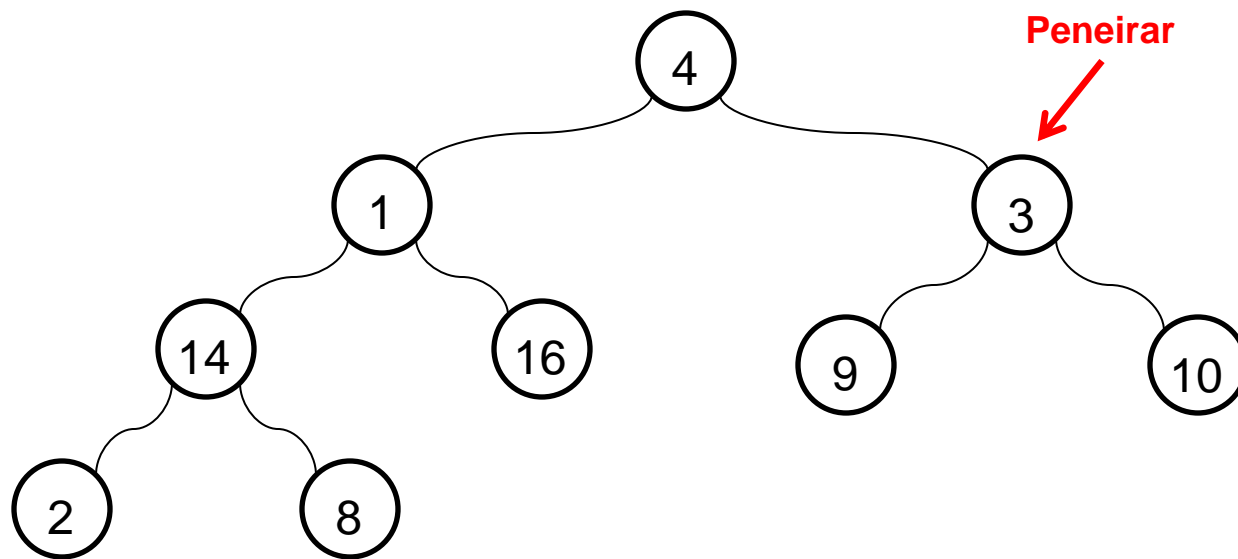


Invoca-se **Peneirar** para **pai = 3**, como resultado a subárvore será um heap.

0	1	2	3	4	5	6	7	8
4	1	3	14	16	9	10	2	8

Construir

- Construir (Exemplo)

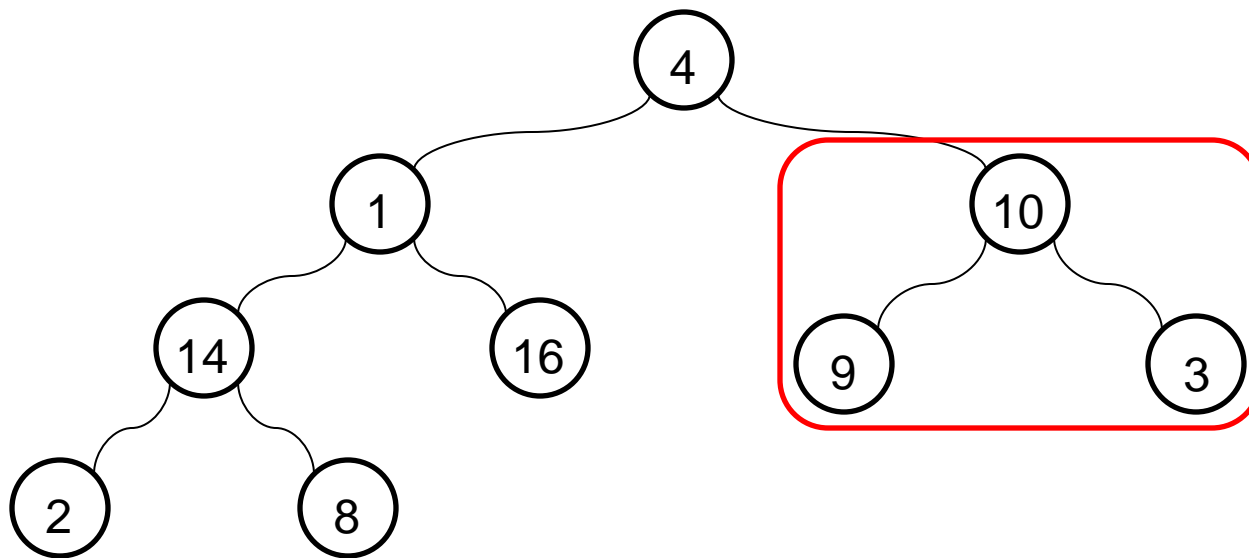


Invoca-se **Peneirar** para **pai = 2**.

0	1	2	3	4	5	6	7	8
4	1	3	14	16	9	10	2	8

Construir

- Construir (Exemplo)

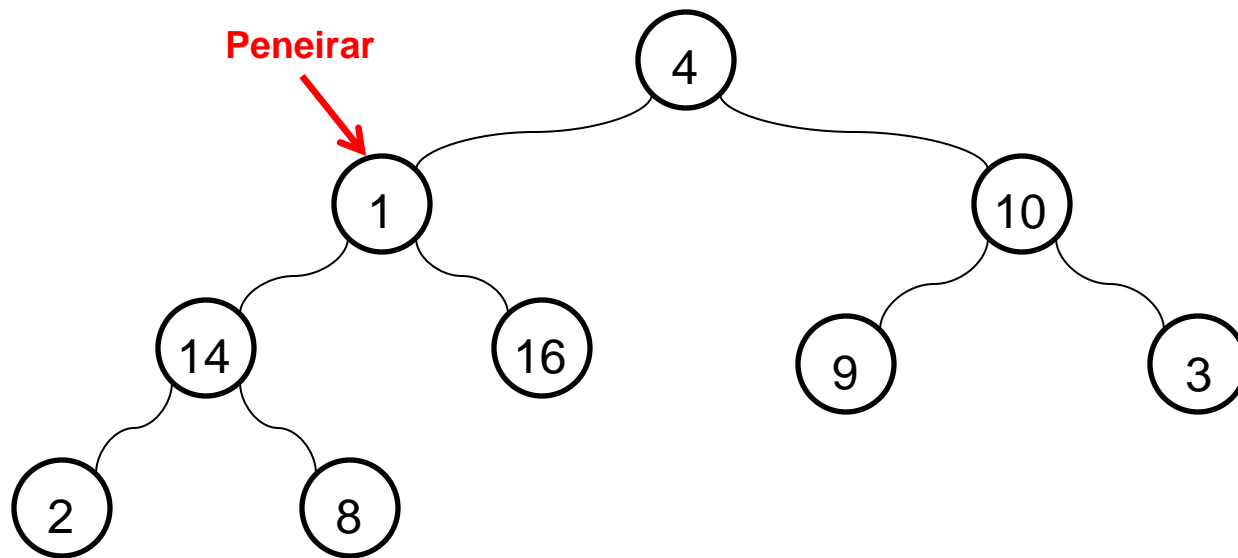


Invoca-se **Peneirar** para **pai = 2**, como resultado a subárvore será um heap.

0	1	2	3	4	5	6	7	8
4	1	10	14	16	9	3	2	8

Construir

- Construir (Exemplo)

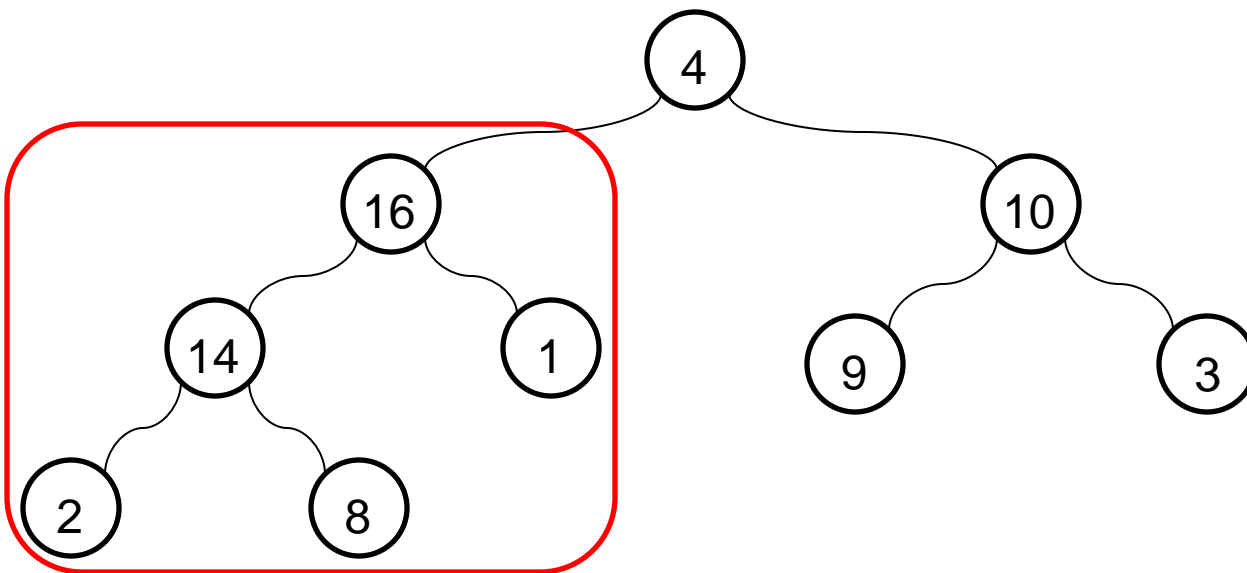


Invoca-se **Peneirar** para **pai = 1**.

0	1	2	3	4	5	6	7	8
4	1	10	14	16	9	3	2	8

Construir

- Construir (Exemplo)

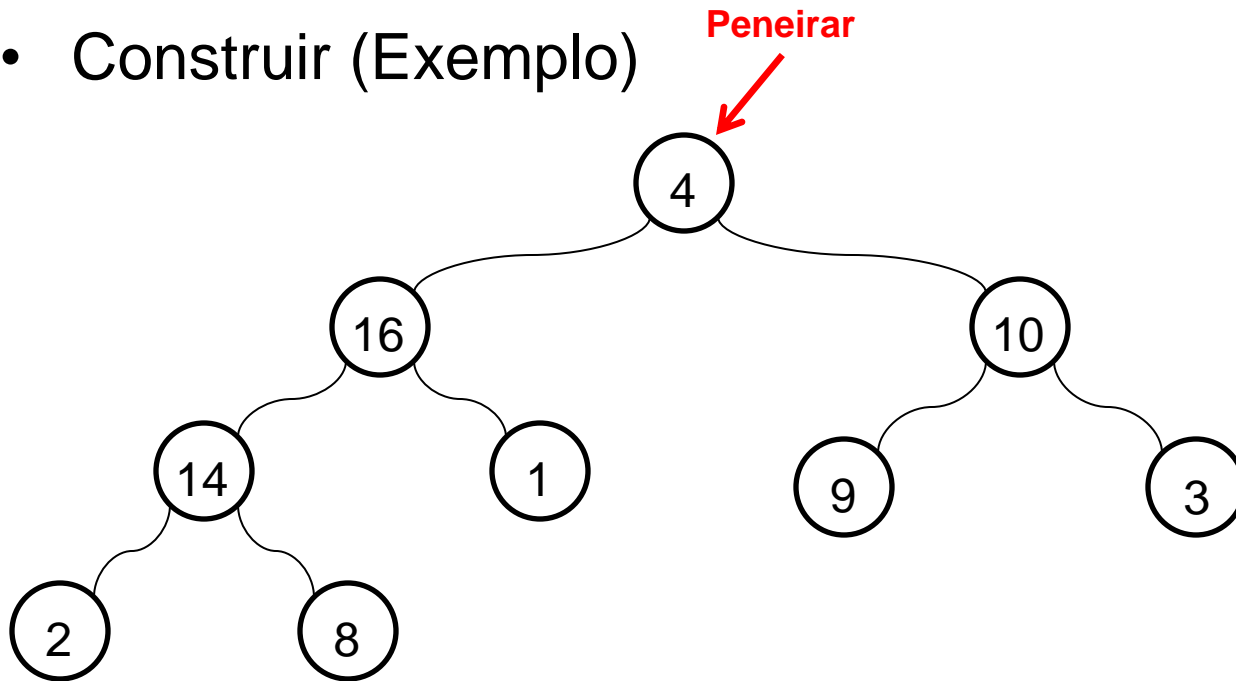


Invoca-se **Peneirar** para **pai = 1**, como resultado a subárvore será um heap.

0	1	2	3	4	5	6	7	8
4	16	10	14	1	9	3	2	8

Construir

- Construir (Exemplo)

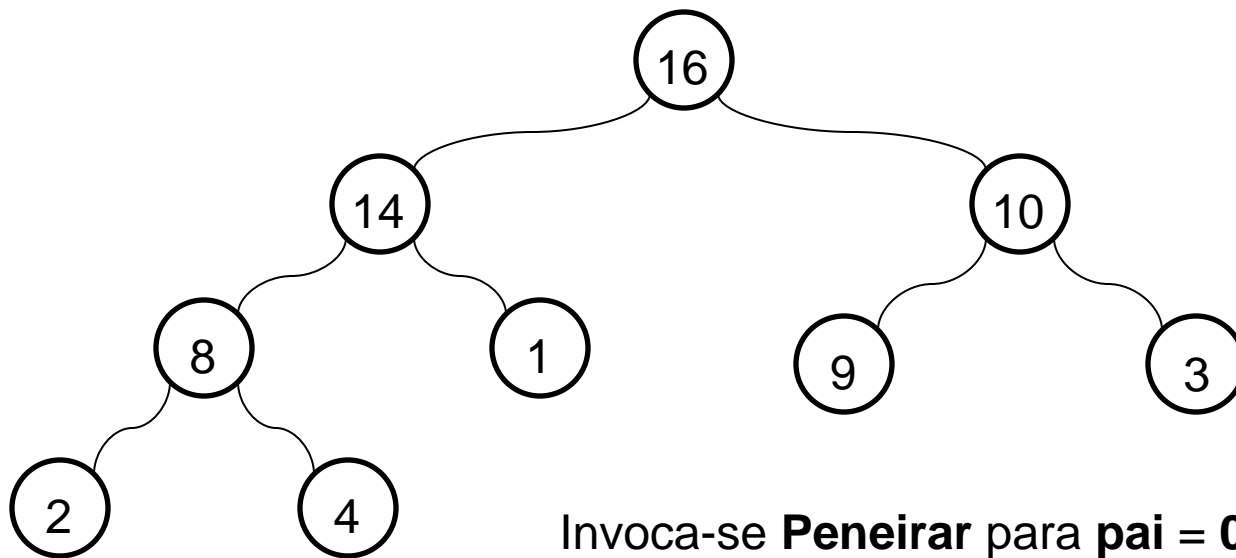


Invoca-se **Peneirar** para **pai = 0**.

0	1	2	3	4	5	6	7	8
4	16	10	14	1	9	3	2	8

Construir

- Construir (Exemplo)



Invoca-se **Peneirar** para **pai = 0**, como resultado a subárvore (no caso toda a árvore) será um heap.

** Observe que Peneirar será invocado recursivamente mais 2 vezes.*

0	1	2	3	4	5	6	7	8
16	14	10	8	1	9	3	2	4

Operações Básicas em um Heap

- Operações Principais:
 - Incluir item no Heap
 - Remover item do Heap (remover o máximo)
- Operações secundárias (apóiam as Operações Principais):
 - Filho Esquerda
 - Filho Direita
 - Pai
 - Último Pai
 - Peneirar
 - Construir (obtem um heap a partir de um vetor qualquer)

Inclusão em um Heap

- Inclusão:

7

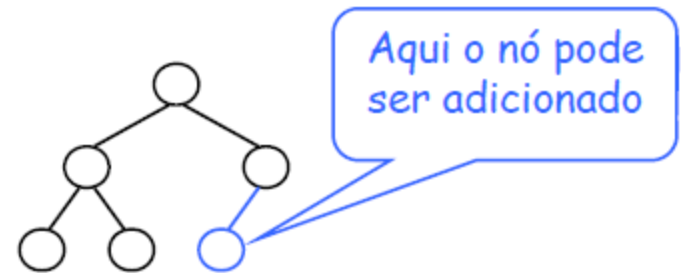
Uma árvore com um único nó já é automaticamente um Heap.

0	1	2	3	4	5	6	7	8
7								

Inclusão em um Heap

- Inclusão:

7



Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
7								

Inclusão em um Heap

- Inclusão:

7



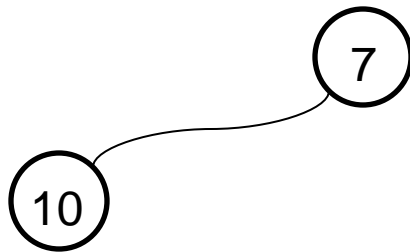
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
7								

Inclusão em um Heap

- Inclusão:



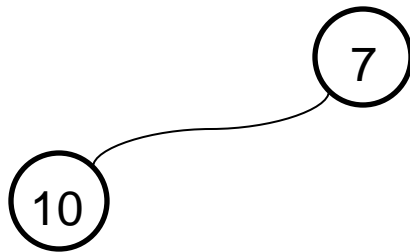
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
7	10							

Inclusão em um Heap

- Inclusão:



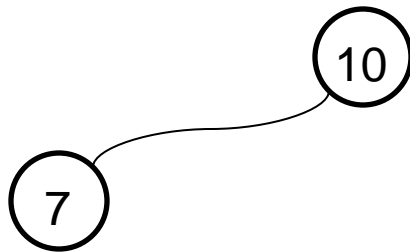
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
7	10							

Inclusão em um Heap

- Inclusão:



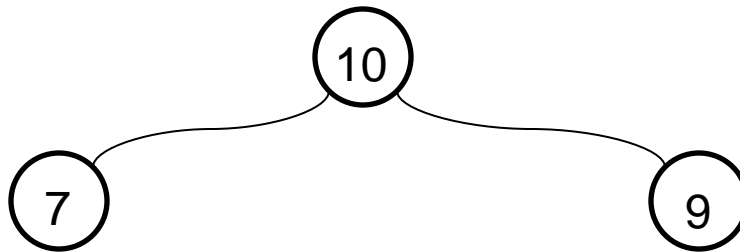
Procedimento para adição de novos nós a um heap:

- Após a chamada por **Construir**, obter-se-á um heap.

0	1	2	3	4	5	6	7	8
10	7							

Inclusão em um Heap

- Inclusão:



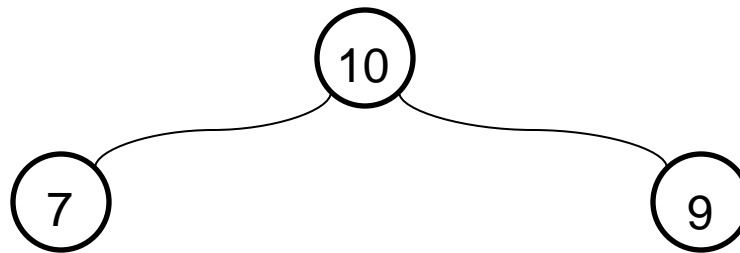
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
10	7	9						

Inclusão em um Heap

- Inclusão:



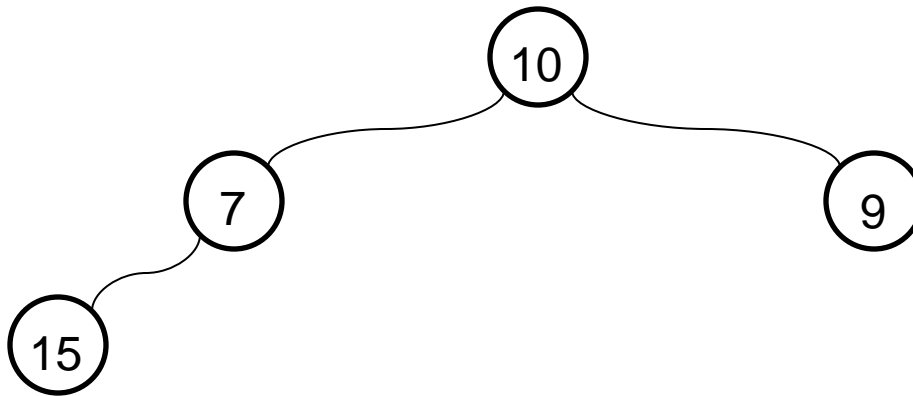
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
10	7	9						

Inclusão em um Heap

- Inclusão:



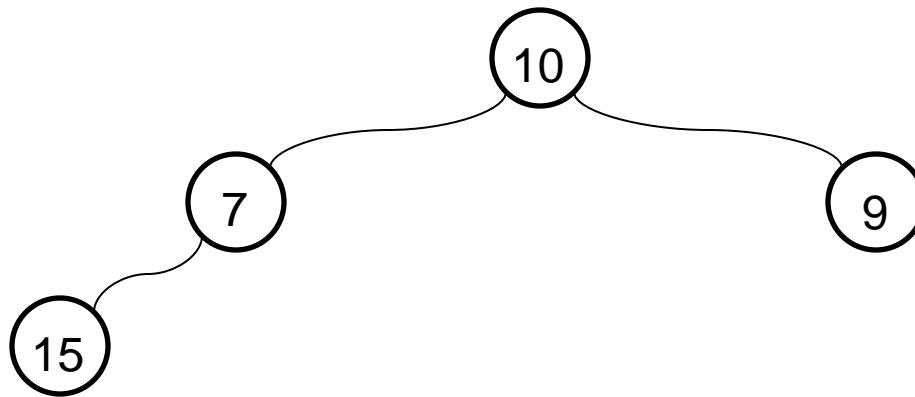
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
10	7	9	15					

Inclusão em um Heap

- Inclusão:



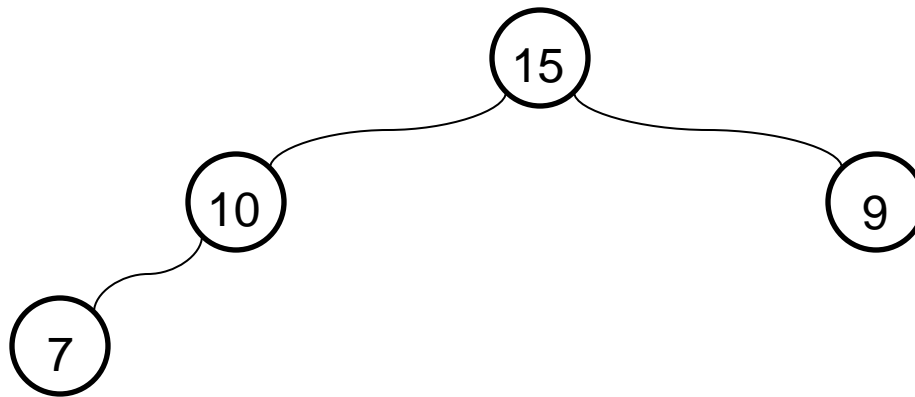
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
10	7	9	15					

Inclusão em um Heap

- Inclusão:



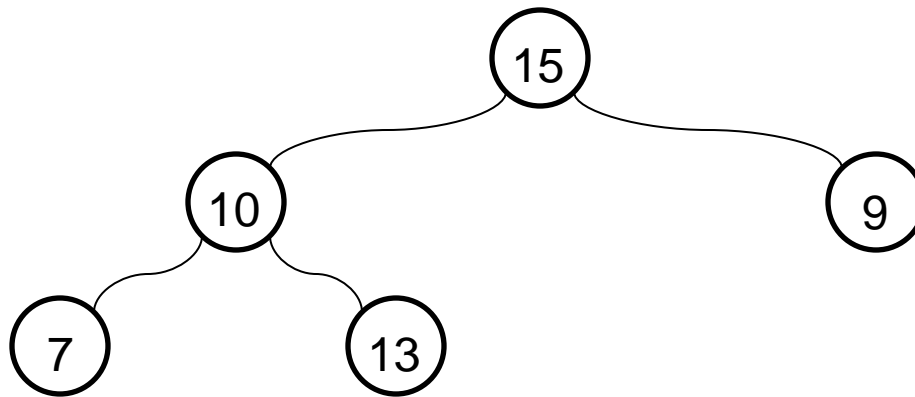
Procedimento para adição de novos nós a um heap:

- Após a chamada por **Construir**, obter-se-á um heap.

0	1	2	3	4	5	6	7	8
15	10	9	7					

Inclusão em um Heap

- Inclusão:



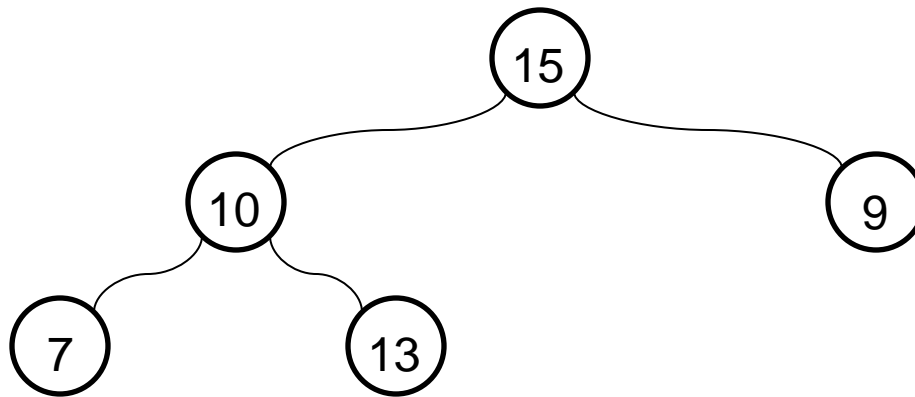
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
15	10	9	7	13				

Inclusão em um Heap

- Inclusão:



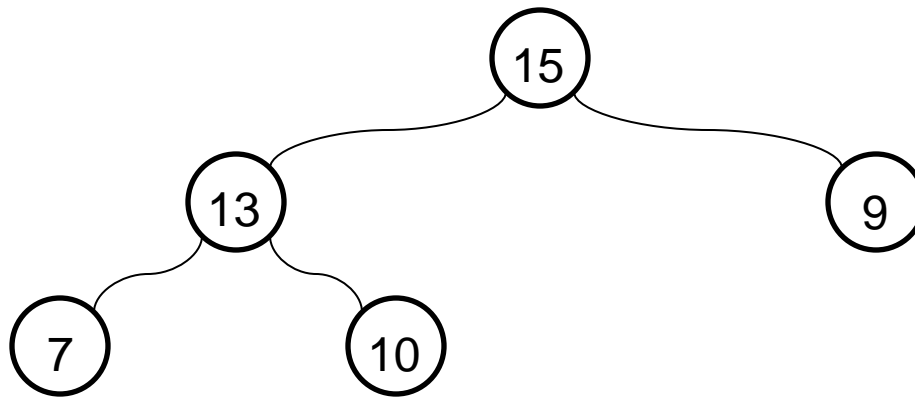
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
15	10	9	7	13				

Inclusão em um Heap

- Inclusão:



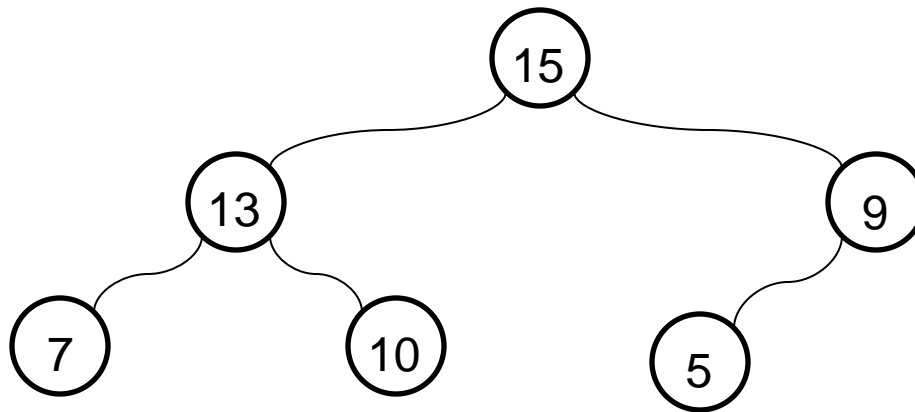
Procedimento para adição de novos nós a um heap:

- Após a chamada por **Construir**, obter-se-á um heap.

0	1	2	3	4	5	6	7	8
15	13	9	7	10				

Inclusão em um Heap

- Inclusão:



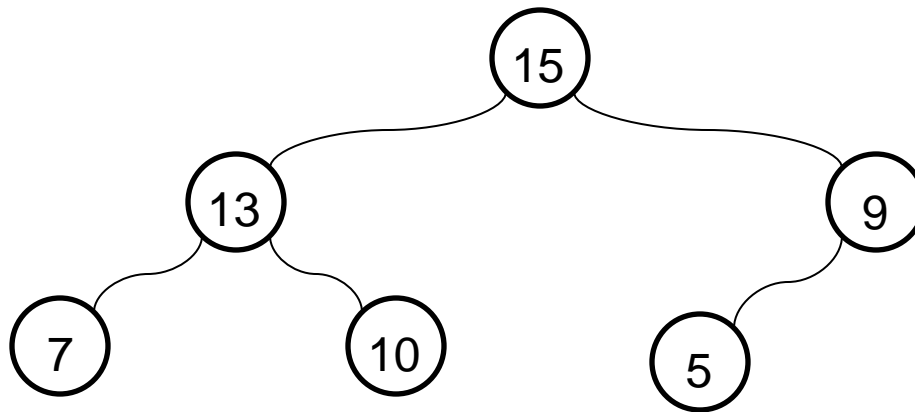
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5			

Inclusão em um Heap

- Inclusão:



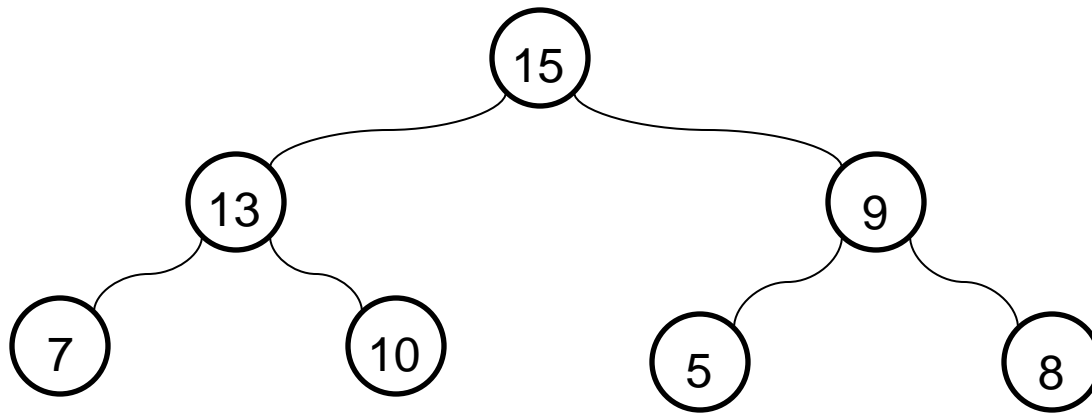
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5			

Inclusão em um Heap

- Inclusão:



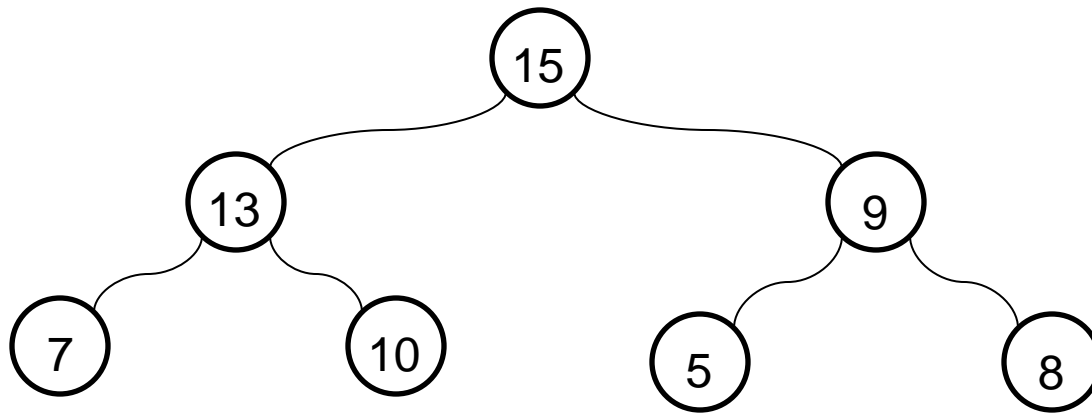
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5	8		

Inclusão em um Heap

- Inclusão:



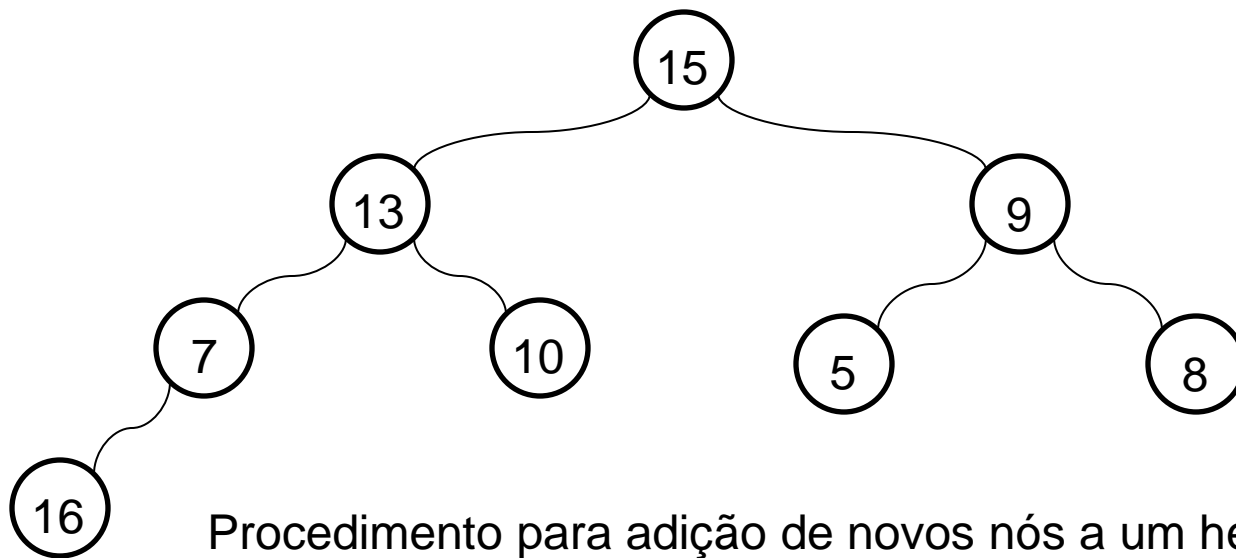
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5	8		

Inclusão em um Heap

- Inclusão:



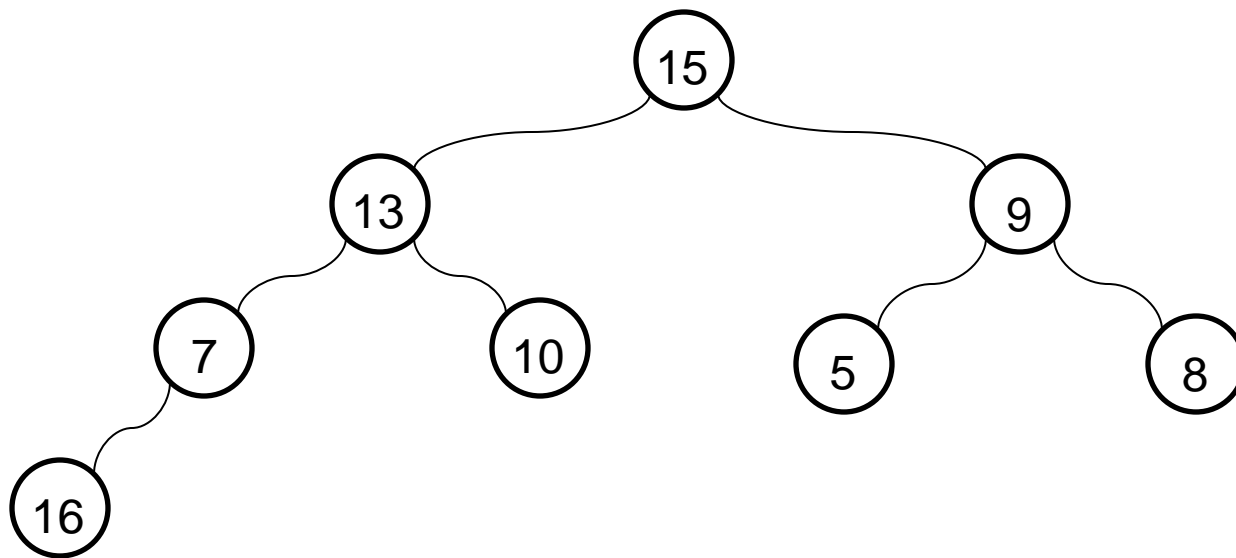
Procedimento para adição de novos nós a um heap:

- Se houver espaço no vetor, então
 - Deve ser folha no último nível, na primeira posição disponível mais à esquerda
 - Se este nível estiver cheio, comece um novo nível.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5	8	16	

Inclusão em um Heap

- Inclusão:



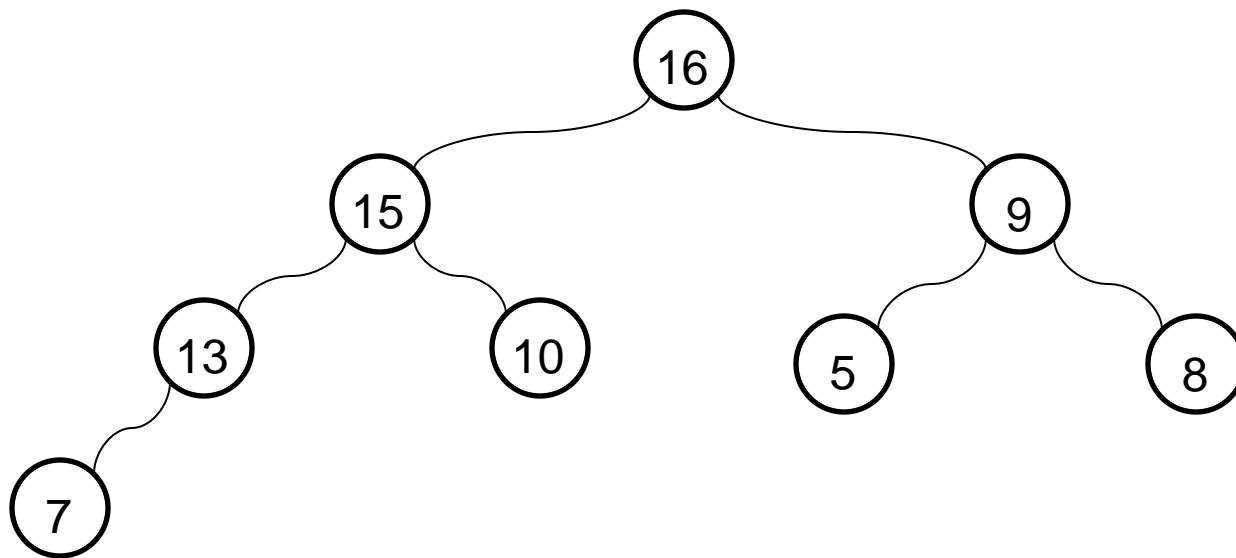
Procedimento para adição de novos nós a um heap:

- Se houve violação da estrutura de heap, então
 - Invoque a **Construir** para o heap.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5	8	16	

Inclusão em um Heap

- Inclusão:



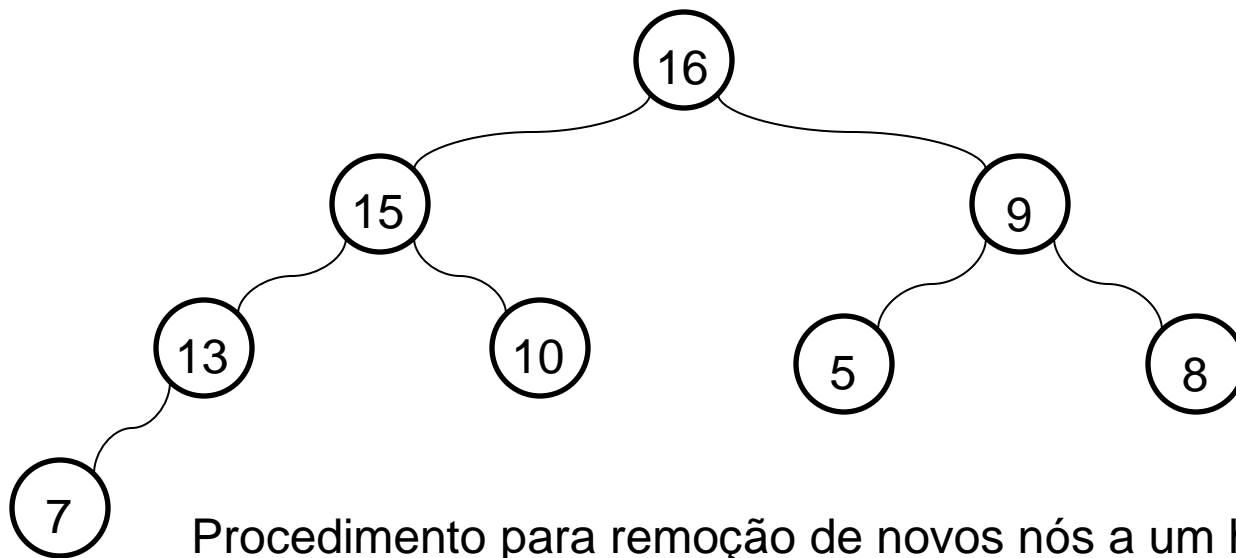
Procedimento para adição de novos nós a um heap:

- Após a chamada por **Construir**, obter-se-á um heap.

0	1	2	3	4	5	6	7	8
16	15	9	13	10	5	8	7	

Remover Item do Heap

- Remoção



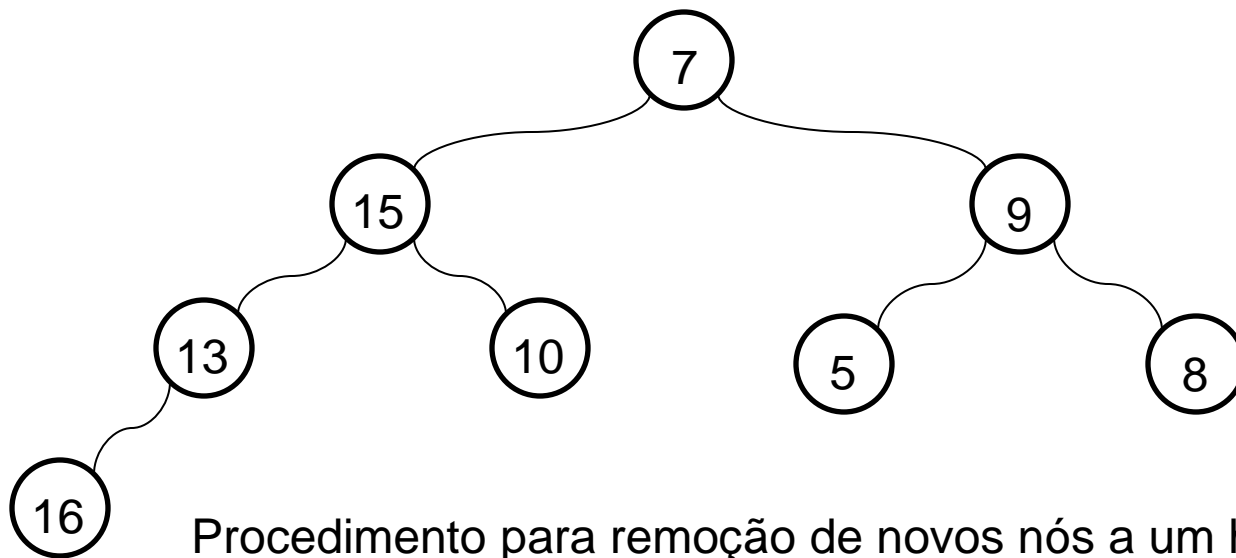
Procedimento para remoção de novos nós a um heap:

- Troca-se o elemento da raiz, índice **0**, com o último elemento do heap;
- Decrementa-se a quantidade;
- Invoca-se **Peneirar** na raiz do heap.

0	1	2	3	4	5	6	7	8
16	15	9	13	10	5	8	7	

Remover Item do Heap

- Remoção



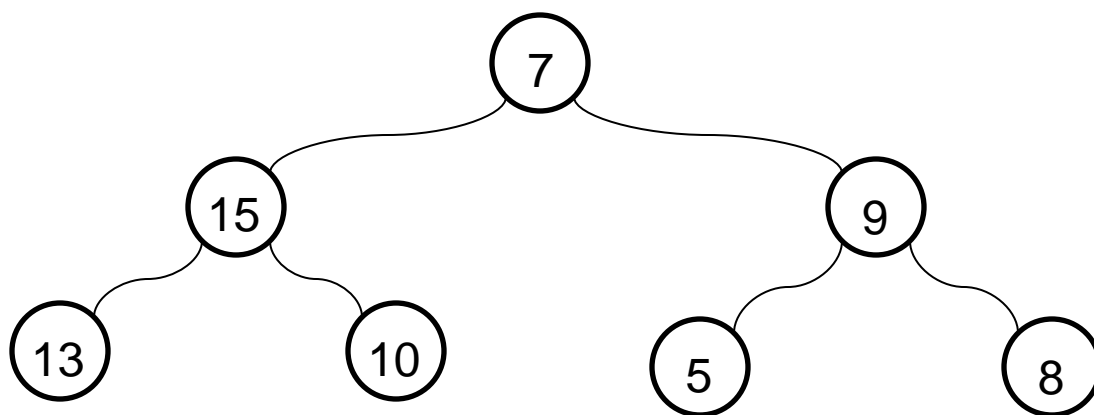
Procedimento para remoção de novos nós a um heap:

- Troca-se o elemento da raiz, índice **0**, com o último elemento do heap;
- Decrementa-se a quantidade;
- Invoca-se **Peneirar** na raiz do heap.

0	1	2	3	4	5	6	7	8
7	15	9	13	10	5	8	16	

Remover Item do Heap

- Remoção



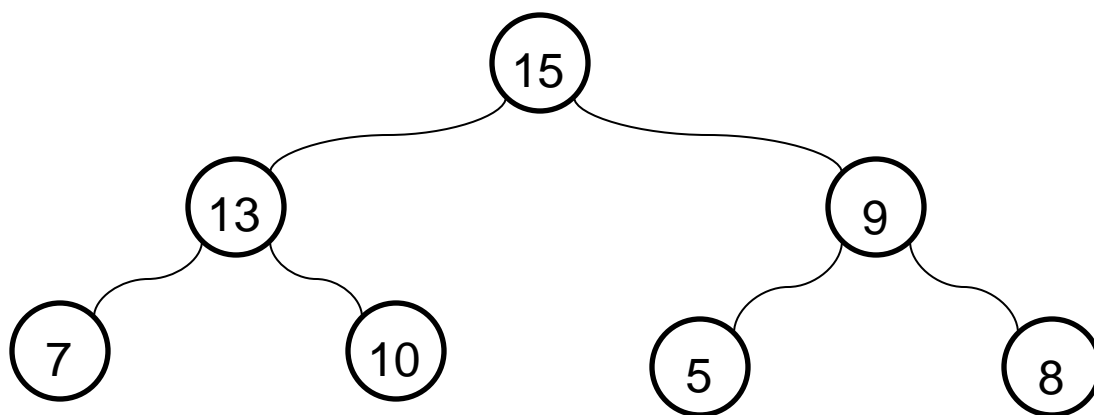
Procedimento para remoção de novos nós a um heap:

- Troca-se o elemento da raiz, índice **0**, com o último elemento do heap;
- Decrementa-se a quantidade;
- Invoca-se **Peneirar** na raiz do heap.

0	1	2	3	4	5	6	7	8
7	15	9	13	10	5	8		

Remover Item do Heap

- Remoção



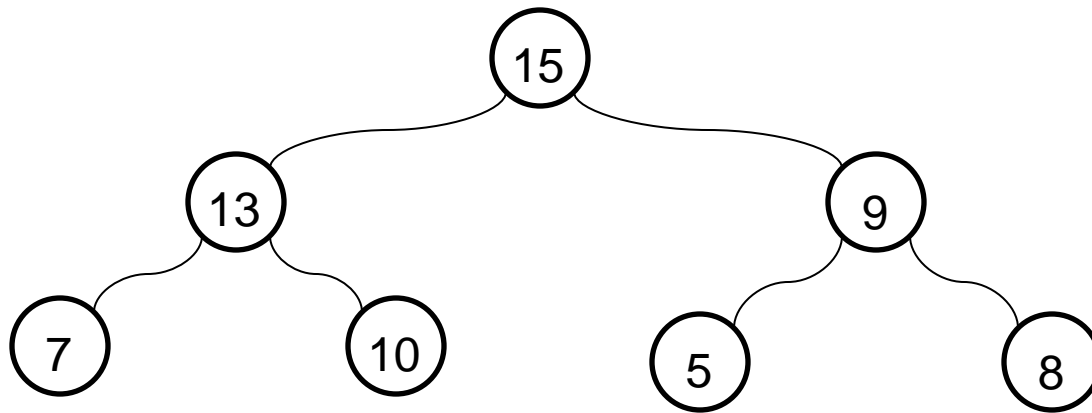
Procedimento para remoção de novos nós a um heap:

- Ao final, **Peneirar** garante que o vetor resultante é heap.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5	8		

Remover Item do Heap

- Remoção



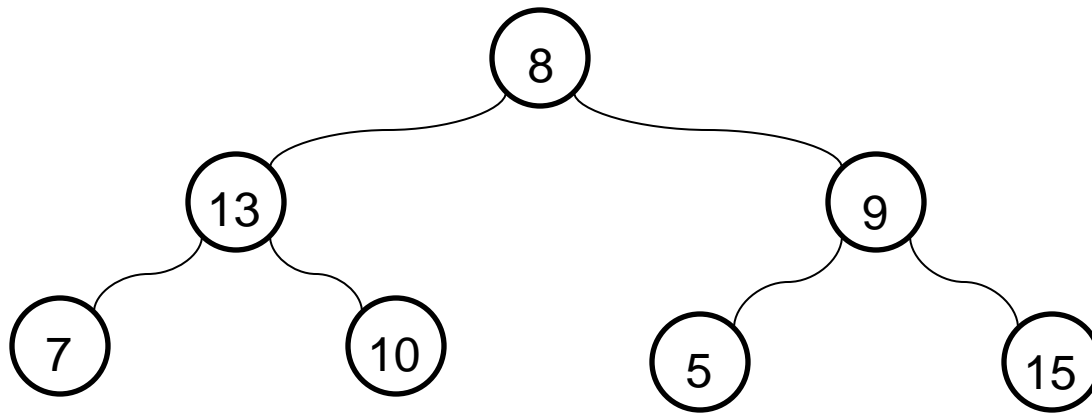
Procedimento para remoção de novos nós a um heap:

- Troca-se o elemento da raiz, índice **0**, com o último elemento do heap;
- Decrementa-se a quantidade;
- Invoca-se **Peneirar** na raiz do heap.

0	1	2	3	4	5	6	7	8
15	13	9	7	10	5	8		

Remover Item do Heap

- Remoção



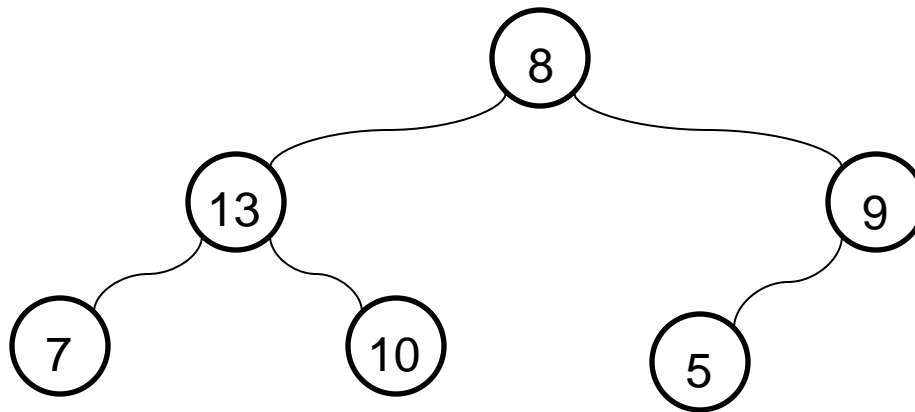
Procedimento para remoção de novos nós a um heap:

- Troca-se o elemento da raiz, índice **0**, com o último elemento do heap;
- Decrementa-se a quantidade;
- Invoca-se **Peneirar** na raiz do heap.

0	1	2	3	4	5	6	7	8
8	13	9	7	10	5	15		

Remover Item do Heap

- Remoção



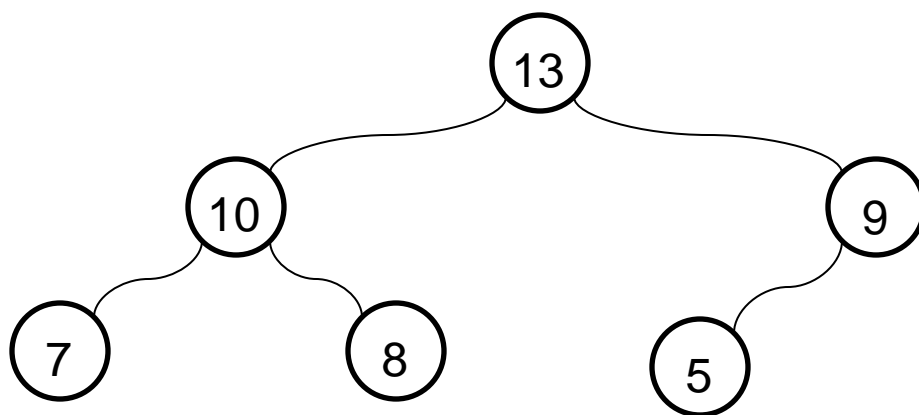
Procedimento para remoção de novos nós a um heap:

- Troca-se o elemento da raiz, índice **0**, com o último elemento do heap;
- Decrementa-se a quantidade;
- Invoca-se **Peneirar** na raiz do heap.

0	1	2	3	4	5	6	7	8
8	13	9	7	10	5			

Remover Item do Heap

- Remoção



Procedimento para remoção de novos nós a um heap:

- Ao final, **Peneirar** garante que o vetor resultante é heap.

0	1	2	3	4	5	6	7	8
13	10	9	7	8	5			

Heap

- Estruturas:

- struct tHeap

- struct tItem → Por simplificação do exemplo nossos itens serão inteiros.

- Funções:

- struct tHeap * criaHeap(void);

- void iniciaHeap(struct tHeap *);

- struct tItem * criaItem(void); → Por simplificação do exemplo nossos itens serão inteiros. Em casos mais complexos teríamos estruturas nesses valores.

- int lerItem(void);



Heap

Por simplificação do exemplo nossos itens não serão devolvidos. Em casos mais complexos teríamos o retorno estruturas.

- Funções Principais:
 - void inserirHeap(struct tHeap *);
 - void inserir(struct tHeap *);
 - void removerMaxHeap(struct tHeap *);
- Funções Secundárias:
 - int filhoDir(int);
 - int filhoEsq(int);
 - int pai(int);
 - int ultimoPai(struct tHeap);
 - void construirHeap(struct tHeap *);
 - void peneirar(struct tHeap *, int);

Heap

- Implementação da Estrutura tHeap

```
1. struct tHeap
2. {
3.     int itens[TAMANHO];
4.     int quantidade;
5. };
```

→ Por simplificação do exemplo nossos itens serão inteiros.

- Criação e Inicialização do Heap

```
1. struct tHeap * criaHeap(void) {
2.     struct tHeap * heap = (struct tHeap *) malloc(sizeof(struct tHeap));
3.     if(heap != NULL) {
4.         iniciaHeap(heap);
5.     } else {
6.         printf("\nErro na alocação do heap!\n");
7.         exit(1);
8.     }
9.     return heap;
10. }
```

```
1. void iniciaHeap(struct tHeap *h)
2. {
3.     h->quantidade = 0;
4. }
```

Heap

- Leitura do Item:

```
1. int lerItem(void) {  
2.     int it;  
3.     printf("Informe o dado: ");  
4.     scanf("%d", &it);  
5.     return it;  
6. }
```

- Função de Apoio:

```
1. int filhoDir(int pai)  
2. {  
3.     return 2*pai+2;  
4. }
```

```
1. int filhoEsq(int pai)  
2. {  
3.     return 2*pai+1;  
4. }
```

```
1. int pai(int filho)  
2. {  
3.     return (int)(filho-1)/2;  
4. }
```

```
1. int ultimoPai(struct tHeap h)  
2. {  
3.     return (h.quantidade/2)-1;  
4. }
```

Heap

- Função Peneirar (Sift)

```
1. void peneirar ( struct tHeap *heap, int pai )    {
2.         int fEsq = filhoEsq ( pai ), fDir = filhoDir ( pai ), maior, aux;
3.         if ( fEsq < heap->quantidade && heap->itens[fEsq] > heap->itens[pai] )    {
4.             maior = fEsq;
5.         }
6.         else    {
7.             maior = pai;
8.         }
9.
10.        if ( fDir < heap->quantidade && heap->itens[fDir] > heap->itens[maior] )    {
11.            maior = fDir;
12.        }
13.
14.        if ( maior != pai )    {
15.            aux = heap->itens[pai];
16.            heap->itens[pai] = heap->itens[maior];
17.            heap->itens[maior] = aux;
18.            peneirar(heap, maior);
19.        }
20. }
```


Heap

- Função Peneirar (Sift)

```
1. void peneirar ( struct tHeap *heap, int pai )    {
2.     int fEsq = filhoEsq ( pai ), fDir = filhoDir ( pai ), maior, aux;
3.     if ( fEsq < heap->quantidade && heap->itens[fEsq] > heap->itens[pai] )    {
4.         maior = fEsq;
5.     }
6.     else    {
7.         maior = pai;
8.     }
9.
10.    if ( fDir < heap->quantidade && heap->itens[fDir] > heap->itens[maior] )    {
11.        maior = fDir;
12.    }
13.
14.    if ( maior != pai )    {
15.        aux = heap->itens[pai];
16.        heap->itens[pai] = heap->itens[maior];
17.        heap->itens[maior] = aux;
18.        peneirar(heap, maior);
19.    }
20. }
```

Obtém o **filho Esquerdo**
e **Filho Direito** do pai.



Heap

- Função Peneirar (Sift)

```
1. void peneirar ( struct tHeap *heap, int pai )    {
2.     int fEsq = filhoEsq ( pai ), fDir = filhoDir ( pai ), maior, aux;
3.     if ( fEsq < heap->quantidade && heap->itens[fEsq] > heap->itens[pai] )    {
4.         maior = fEsq;
5.     }
6.     else {
7.         maior = pai;
8.     }
9.
10.    if ( fDir < heap->quantidade && heap->itens[fDir] > heap->itens[maior] )    {
11.        maior = fDir;
12.    }
13.
14.    if ( maior != pai )    {
15.        aux = heap->itens[pai];
16.        heap->itens[pai] = heap->itens[maior];
17.        heap->itens[maior] = aux;
18.        peneirar(heap, maior);
19.    }
20. }
```

Se o **pai** tiver **Filho Esquerdo** e o filho for maior que o **pai**, **maior** recebe o **Filho Esquerdo**.

Heap

- Função Peneirar (Sift)

```
1. void peneirar ( struct tHeap *heap, int pai )    {
2.     int fEsq = filhoEsq ( pai ), fDir = filhoDir ( pai ), maior, aux;
3.     if ( fEsq < heap->quantidade && heap->itens[fEsq] > heap->itens[pai] )    {
4.         maior = fEsq;
5.     }
6.     else {
7.         maior = pai; → Senão o pai é o maior.
8.     }
9.
10.    if ( fDir < heap->quantidade && heap->itens[fDir] > heap->itens[maior] )    {
11.        maior = fDir;
12.    }
13.
14.    if ( maior != pai )    {
15.        aux = heap->itens[pai];
16.        heap->itens[pai] = heap->itens[maior];
17.        heap->itens[maior] = aux;
18.        peneirar(heap, maior);
19.    }
20. }
```

Heap

- Função Peneirar (Sift)

```
1. void peneirar ( struct tHeap *heap, int pai )    {
2.         int fEsq = filhoEsq ( pai ), fDir = filhoDir ( pai ), maior, aux;
3.         if ( fEsq < heap->quantidade && heap->itens[fEsq] > heap->itens[pai] )    {
4.             maior = fEsq;
5.         }
6.         else {
7.             maior = pai;
8.         }
9.
10.         if ( fDir < heap->quantidade && heap->itens[fDir] > heap->itens[maior] )    {
11.             maior = fDir;
12.         }
13.
14.         if ( maior != pai )    {
15.             aux = heap->itens[pai];
16.             heap->itens[pai] = heap->itens[maior];
17.             heap->itens[maior] = aux;
18.             peneirar(heap, maior);
19.         }
20. }
```

Se o **pai** tiver **Filho Direito** e o filho for maior que o **maior** encontrado, então **maior** recebe o **Filho Direito**. (o Filho Direito é o maior de todos)

Heap

- Função Peneirar (Sift)

```
1. void peneirar ( struct tHeap *heap, int pai )    {
2.         int fEsq = filhoEsq ( pai ), fDir = filhoDir ( pai ), maior, aux;
3.         if ( fEsq < heap->quantidade && heap->itens[fEsq] > heap->itens[pai] )    {
4.             maior = fEsq;
5.         }
6.         else {
7.             maior = pai;
8.         }
9.
10.        if ( fDir < heap->quantidade && heap->itens[fDir] > heap->itens[maior] )    {
11.            maior = fDir;
12.        }
13.
14.        if ( maior != pai ) {
15.            aux = heap->itens[pai];
16.            heap->itens[pai] = heap->itens[maior];
17.            heap->itens[maior] = aux;
18.            peneirar(heap, maior);
19.        }
20. }
```

Caso o **maior** encontrado não seja o próprio **pai** (então há violação), troque o **pai** pelo **maior** filho encontrado e invoque **peneirar** recursivamente na subárvore onde houve a troca.

Heap

- Função Constroi Heap

```
1. void construirHeap(struct tHeap *heap)
2. {
3.     int i;
4.     for ( i = ultimoPai ( *heap ); i >= 0; i--)
5.     {
6.         peneirar ( heap, i );
7.     }
8. }
```

Heap

- Função Constroi Heap

```
1. void construirHeap(struct tHeap *heap)
2. {
3.     int i;
4.     for ( i = ultimoPai ( *heap ); i >= 0; i--)
5.     {
6.         peneirar ( heap, i );
7.     }
8. }
```

Para cada nó a partir do **Último Pai**, subindo até a **raiz** invoque o **peneirar**.

Heap

- Função Principal: Inserir

```
1. void inserirHeap(struct tHeap *heap) {
2.     int novo;
3.     int novoInd = heap->quantidade;
4.     if(heap->quantidade != TAMANHO) {
5.         novo = lerItem();
6.         heap->itens[novoInd] = novo;
7.         heap->quantidade++;
8.         if(heap->quantidade != 1) {
9.             if(heap->itens[pai(novoInd)] < heap->itens[novoInd]) {
10.                 construirHeap(heap);
11.             }
12.         }
13.     } else {
14.         printf("\nHeap Cheio!\n");
15.         system("pause");
16.     }
17. }
```


Heap

- Função Principal: Inserir

```
1. void inserirHeap(struct tHeap *heap) {
2.     int novo;
3.     int novoInd = heap->quantidade;
4.     if(heap->quantidade != TAMANHO) {
5.         novo = lerItem();
6.         heap->itens[novoInd] = novo;
7.         heap->quantidade++;
8.         if(heap->quantidade != 1) {
9.             if(heap->itens[pai(novoInd)] < heap->itens[novoInd]) {
10.                 construirHeap(heap);
11.             }
12.         }
13.     } else {
14.         printf("\nHeap Cheio!\n");
15.         system("pause");
16.     }
17. }
```

Se há espaço no Heap

Heap

- Função Principal: Inserir

```
1. void inserirHeap(struct tHeap *heap) {  
2.     int novo;  
3.     int novoInd = heap->quantidade;  
4.     if(heap->quantidade != TAMANHO) {  
5.         novo = lerItem();  
6.         heap->itens[novoInd] = novo;  
7.         heap->quantidade++;  
8.         if(heap->quantidade != 1) {  
9.             if(heap->itens[pai(novoInd)] < heap->itens[novoInd]) {  
10.                 construirHeap(heap);  
11.             }  
12.         }  
13.     } else {  
14.         printf("\nHeap Cheio!\n");  
15.         system("pause");  
16.     }  
17. }
```

Lê um novo item, e coloque na última posição mais a esquerda e incremente a quantidade

Heap

- Função Principal: Inserir

```
1. void inserirHeap(struct tHeap *heap) {
2.     int novo;
3.     int novoInd = heap->quantidade;
4.     if(heap->quantidade != TAMANHO) {
5.         novo = lerItem();
6.         heap->itens[novoInd] = novo;
7.         heap->quantidade++;
8.         if(heap->quantidade != 1) {
9.             if(heap->itens[pai(novoInd)] < heap->itens[novoInd]) {
10.                 construirHeap(heap);
11.             }
12.         }
13.     } else {
14.         printf("\nHeap Cheio!\n");
15.         system("pause");
16.     }
17. }
```

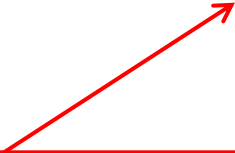
Se for o primeiro item,
então já é heap.

Heap

- Função Principal: Inserir

```
1. void inserirHeap(struct tHeap *heap) {
2.     int novo;
3.     int novoInd = heap->quantidade;
4.     if(heap->quantidade != TAMANHO) {
5.         novo = lerItem();
6.         heap->itens[novoInd] = novo;
7.         heap->quantidade++;
8.         if(heap->quantidade != 1) {
9.             if(heap->itens[pai(novoInd)] < heap->itens[novoInd]) {
10.                 construirHeap(heap);
11.             }
12.         }
13.     } else {
14.         printf("\nHeap Cheio!\n");
15.         system("pause");
16.     }
17. }
```

Se **não** for o primeiro item, então verifique se houve violação da estrutura, se sim, então invoque **constroi** para recompor o **heap**.



Heap

- Função Principal: Remover

```
1. void removerMaxHeap(struct tHeap *heap) {
2.     int aux;
3.     if(heap->quantidade > 0) {
4.         aux = heap->itens[0];
5.         heap->itens[0] = heap->itens[heap->quantidade-1];
6.         heap->itens[heap->quantidade-1] = aux;
7.         heap->quantidade--;
8.         construirHeap(heap);
9.         printf("\nItem Maximo Removido: %d\n", aux);
10.    }
11.    else {
12.        printf("\nHeap Vazio!!!\n");
13.    }
14.    system("pause");
15. }
```

Heap

- Função Principal: Remover


```
1. void removerMaxHeap(struct tHeap *heap) {  
2.     int aux;  
3.     if(heap->quantidade > 0) {  
4.         aux = heap->itens[0];  
5.         heap->itens[0] = heap->itens[heap->quantidade-1];  
6.         heap->itens[heap->quantidade-1] = aux;  
7.         heap->quantidade--;  
8.         construirHeap(heap);  
9.         printf("\nItem Maximo Removido: %d\n", aux);  
10.    }  
11.    else {  
12.        printf("\nHeap Vazio!!!\n");  
13.    }  
14.    system("pause");  
15. }
```

Se houver itens no Heap

Heap

- Função Principal: Remove

```
1. void removerMaxHeap(struct tHeap *heap) {
2.     int aux;
3.     if(heap->quantidade > 0) {
4.         aux = heap->itens[0];
5.         heap->itens[0] = heap->itens[heap->quantidade-1];
6.         heap->itens[heap->quantidade-1] = aux;
7.         heap->quantidade--;
8.         construirHeap(heap);
9.         printf("\nItem Maximo Removido: %d\n", aux);
10.    }
11.    else {
12.        printf("\nHeap Vazio!!!\n");
13.    }
14.    system("pause");
15. }
```




Troca a **raiz** com o **último item** do heap e decrementa a quantidade.

Heap

- Função Principal: Remover

```
1. void removerMaxHeap(struct tHeap *heap) {  
2.     int aux;  
3.     if(heap->quantidade > 0) {  
4.         aux = heap->itens[0];  
5.         heap->itens[0] = heap->itens[heap->quantidade-1];  
6.         heap->itens[heap->quantidade-1] = aux;  
7.         heap->quantidade--;  
8.         construirHeap(heap);  
9.         printf("\nItem Maximo Removido: %d\n", aux);  
10.    }  
11.    else {  
12.        printf("\nHeap Vazio!!!\n");  
13.    }  
14.    system("pause");  
15. }
```



Invoca **constroi** para recompôr as propriedades de heap.