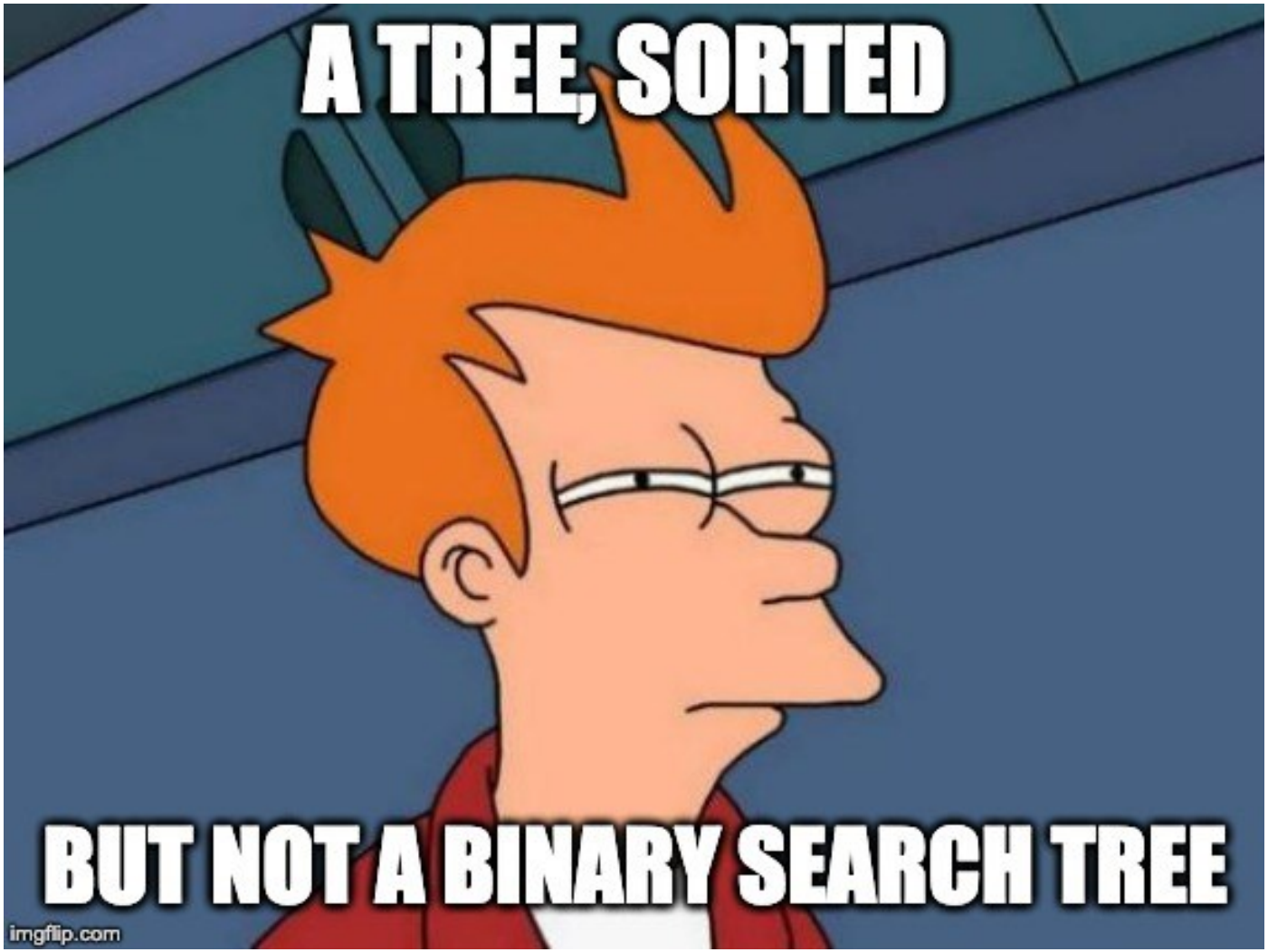
Filas de prioridade e heap



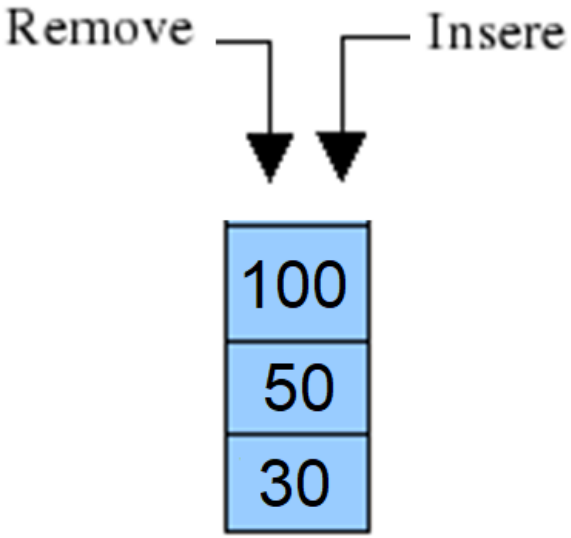
Fila de prioridade

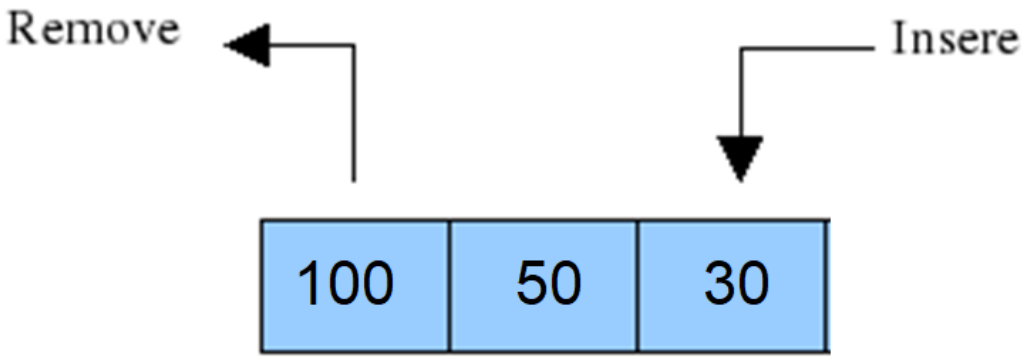
- **Definição:** Uma fila de prioridade é uma estrutura de dados formada por duas operações básicas:
  - Inserir um novo elemento.
  - Remover o elemento que tem **maior prioridade** (maior chave).

- Quando usar?
  - Em aplicações que necessitamos **remover da fila** o elemento de **maior prioridade** (ex: pronto socorro).

Fila de prioridade

1. Uma **pilha** se comporta como uma **fila de prioridades** quando: o elemento de maior chave é o último inserido.
2. Uma **fila** se comporta como uma **fila de prioridades** quando: o elemento com maior chave é sempre removido primeiro.





Fila de prioridade

Para ilustrar, considere uma lista de tarefas (mediada por um campo chave).

Logo,

- A cada momento, realiza-se a tarefa de maior prioridade.
- Selecione a tarefa de maior prioridade da lista, e então remova-a da lista.
- Prioridades das tarefas **podem mudar !!!**
- Novas tarefas podem chegar (inserção), e necessitam ser acomodadas de acordo com seus graus de prioridade.

Fila de prioridade

Operações usuais:

- Inserir novo elemento.
- Remover elemento de maior prioridade.
- Seleção do elemento de maior prioridade.
- Alteração da prioridade de um dado elemento.

Fila de prioridade

- Há diferentes propostas de TADs para representar uma fila de prioridade:

  1. Lista não-ordenada.
  2. Lista ordenada (segundo prioridade dada pelo campo chave).
  3. Heap.

1) Fila de prioridade: lista não-ordenada

Operações

- **Inserção:** elementos (registros) podem ser introduzidos em uma lista em qualquer ordem.
- **Remoção:** percorrer a lista sequencialmente em busca do elemento de maior prioridade.
- **Alteração:** não implica em mudança na estrutura, mas exige buscar o elemento a ser alterado.
- **Seleção:** idem à Remoção.

1) Fila de prioridade: lista não-ordenada

Complexidade (lista com n elementos)

- **Inserção:** O(1)
- **Seleção:** O(n)
- **Remoção:** O(n)
- **Alteração:** O(n)
- **Construção:** O(n)

2) Fila de prioridade: lista ordenada

Operações

- **Inserção:** necessita percorrer a lista para encontrar a posição exata de inserção (**para manter a ordenação válida**!).
- **Remoção/Seleção:** imediata, visto que o elemento de maior prioridade será o **primeiro da lista**.
- **Alteração:** similar à inserção.

2) Fila de prioridade: lista ordenada

Complexidade (lista com n elementos)

- **Inserção:** O(n)
- **Seleção:** O(1)
- **Remoção:** O(1)
- **Alteração:** O(n)
- **Construção:** O(n log n) (complexidade da ordenação).

3) Fila de prioridade: heap

Fila de prioridades via heap

- **Definição (Heap)**: É uma (i) **Árvore Binária Quase Completa,** e também uma árvore de (ii) **Prioridade**.
- **Em termos de TAD**: é representada por uma **lista linear (array)** composta de elementos contendo suas chaves.
  - As chaves representam a prioridade!!!
- **Vantagem:** É mais eficiente nas diferentes operações, em especial a alteração, cujas alternativas anteriores é O(n).

Iniciando Faculdade de TI

5 anos depois

Árvore binária quase completa

- **Definição:** Uma **Árvore Binária** é dita **Quase Completa (ABQC)** quando:
  1. Todos os seus níveis estão cheios, exceto o último.
  2. Os nós do último nível estão o mais à esquerda possível.

Equivalentemente para 1), tem-se também:

1. Se a altura da árvore binária é *d*, cada nó-folha deverá estar ou no nível *d* ou no nível *d-1*.
2. As sub-árvores vazias estão apenas no último ou penúltimo nível.

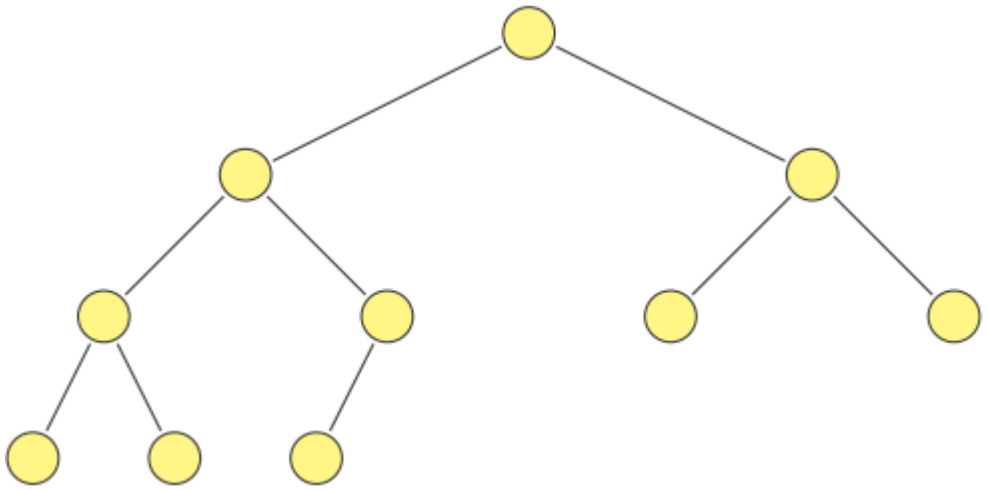
Observação: No contexto de heap, alguns livros estabelecem o conceito de árvore completa como se fosse nossa definição de quase completa.

Árvore binária quase completa - propriedades

- **Teorema:** Se uma ABQC tem N nós, então sua altura h (número total de níveis) é dada pela fórmula:

$$h(N) = \lceil \log_2(N+1) \rceil$$

Na fórmula acima, o operador: $\lceil x \rceil$ representa o menor número inteiro maior ou igual a x (é o "arredondamento para cima").

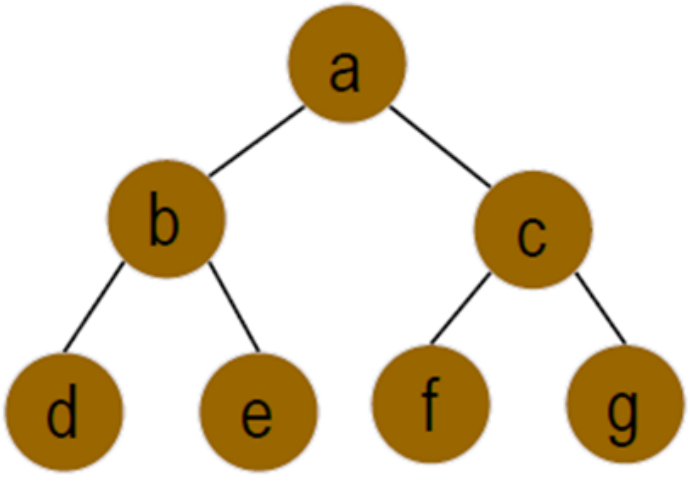


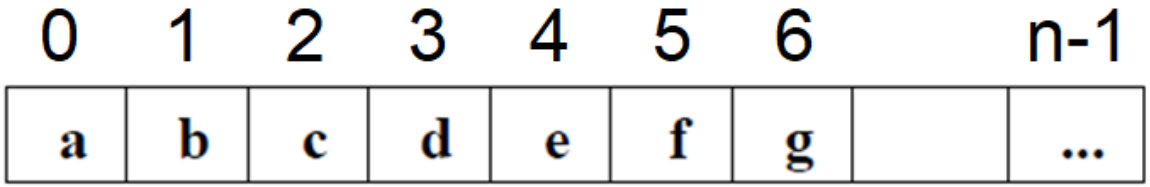
$N = 10$ nós, então:

$$h(10) = \lceil \log_2(11+1) \rceil$$
$$= \lceil 3{,}4594 \rceil = 4$$

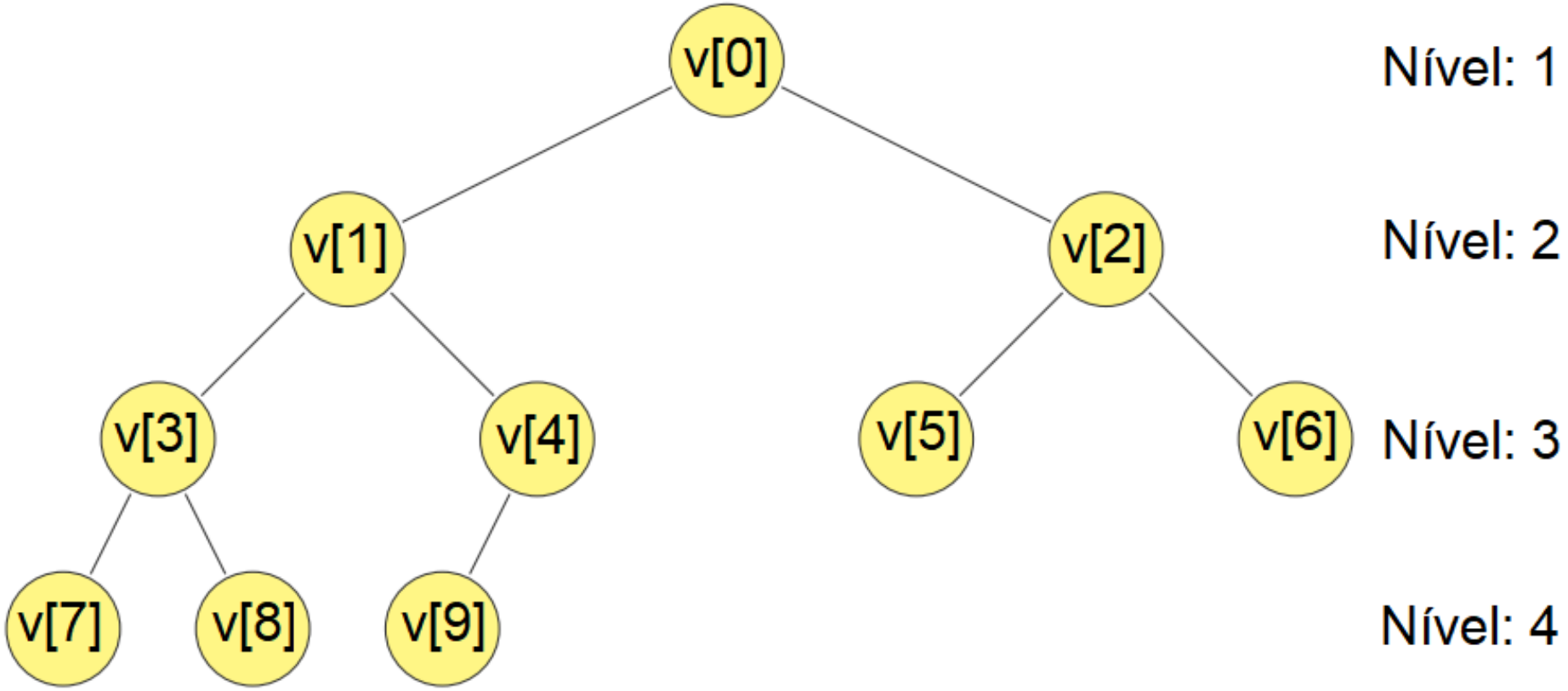
TAD: Árvore binária quase completa

- **Objetivo:** Implementar uma **Árvore Binária Quase Completa** via **alocação estática e sequencial!**
- **Ideia central:** Associar os nós de cada nível com os elementos de um **array**, aproveitando da melhor forma possível a estrutura "fechada" e "padronizada" desse tipo de árvore.





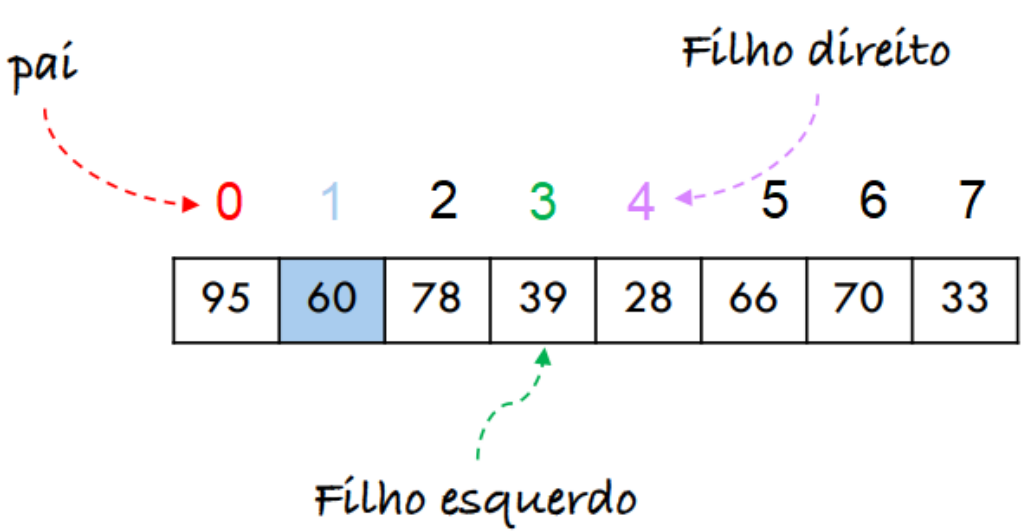
TAD: Árvore binária quase completa

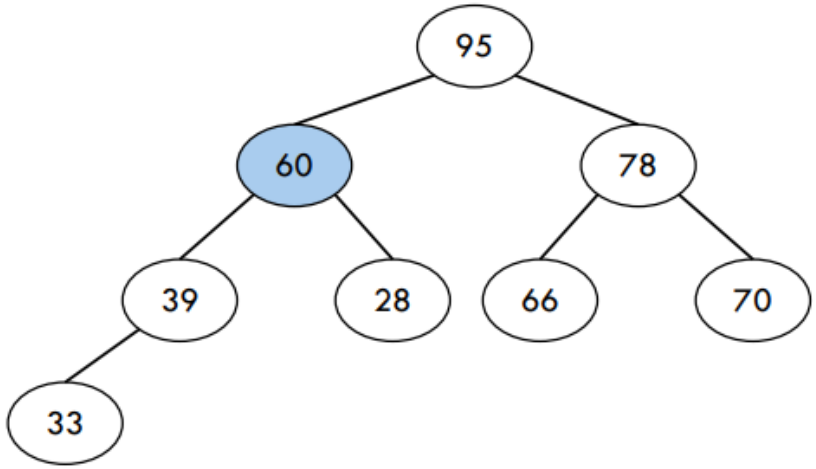


- **Critério lógico:** Se um nó está na posição $\boldsymbol{i}$, seus filhos estarão nas posições $\boldsymbol{2i + 1}$ e $(\boldsymbol{2i + 2})$.

TAD: Árvore binária quase completa

- Em resumo:
  - Filho esquerdo de v[i]: v[2*i + 1]
  - Filho direito de v[i]: v[2*i + 2]
  - Dado um nó alocado em v[i], seu pai será o elemento v[(i-1)/2] (divisão inteira!)





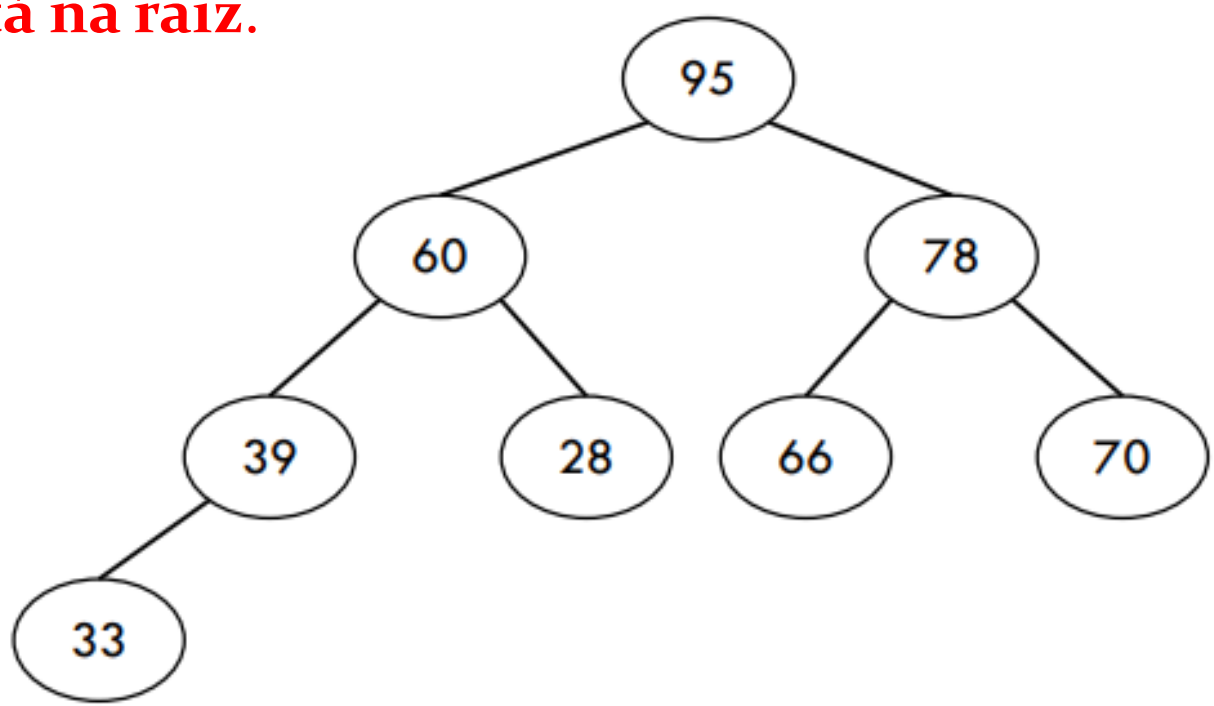
TAD: Árvore binária quase completa

- **Vantagens do uso desse tipo de TAD (array):**
  - Utiliza um critério matemático simples para estabelecer conexão entre os nós e seus filhos.
  - Alocação de fácil gerenciamento, pois usa array.
  - Reduz a possibilidade de espaços vagos no array, pois a árvore é organizada de forma a preservar seus nós folhas ao mais a esquerda e nos últimos níveis possíveis.

Árvore binária de prioridade

- **Definição:** Uma **Árvore Binária** é dita de **Prioridade** quando:
  - O valor de cada nó tem maior ou igual prioridade que de seus filhos ➔ Filhos **são menores ou iguais ao pai**.

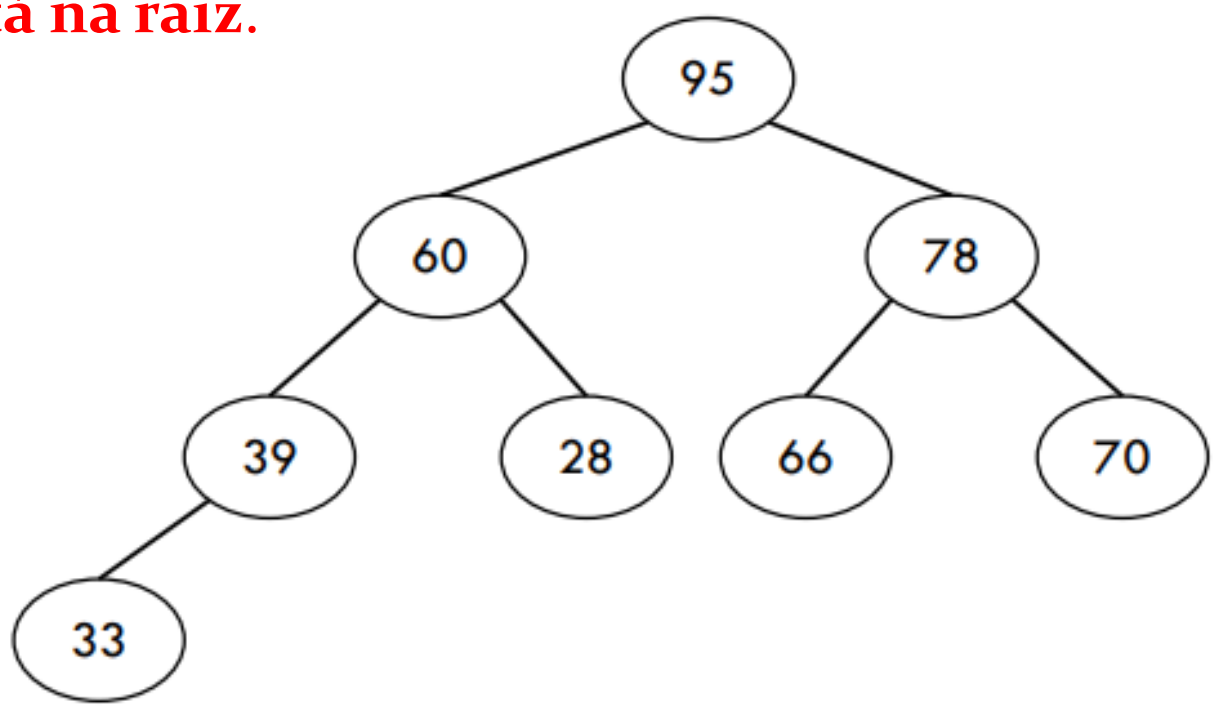
Consequência: **o elemento de maior prioridade da árvore está na raiz.**



Árvore bin

- **Definição:** Uma **Árvore**
  - O valor de cada nó
    seus filhos ➔ Filhos

Consequência: **o elem**
está na raiz.



Árvore bin

- **Definição:** Uma **Árvore**
  - O valor de cada nó
    seus filhos ➔ Filhos

Consequência: **o elem**

Árvore binária de prioridade - finalidade

- Usadas para **implementar filas de prioridade.**
  - Permite que elementos sejam adicionados com uma prioridade associada.
  - Permite que elementos com a maior prioridade sejam removidos primeiro.
- São efetivas para **inserção** e **remoção** de elementos, tornando-as úteis para algoritmos de manipulação de fila de prioridade.

Voltando para a definição de heap ...

Fila de prioridade: heap

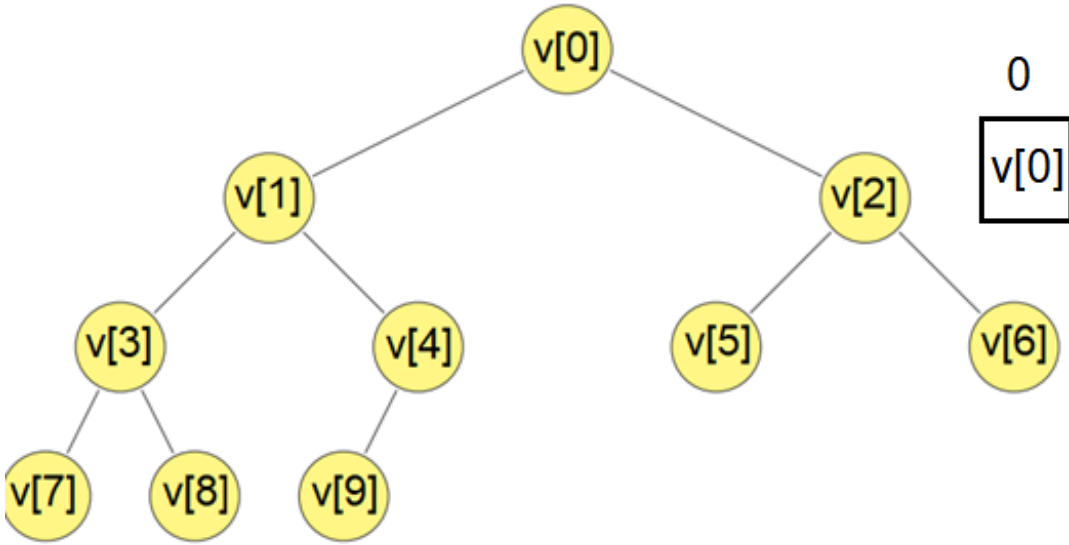
- Definição (**Heap**): É uma (i) **Árvore Binária Quase Completa,** e também uma árvore de (ii) **Prioridade**.

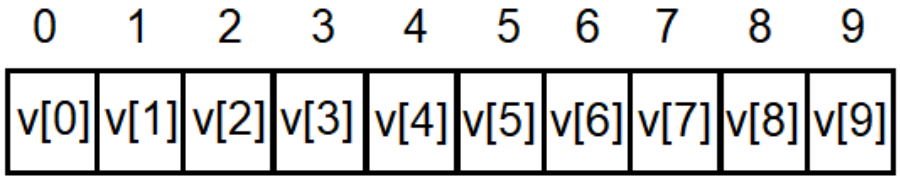
Heap de máximo (ou Max-Heap):

- **TAD**: Lista (**array**), que é interpretada a partir de um heap (árvore quase completa, e de prioridade máxima).
- Em um Heap de máximo:
  - Queremos o **elemento de maior prioridade!!!**
  - **Raiz detém o elemento de maior prioridade**.
  - Os filhos **são menores ou iguais ao seu pai**.

Heap: TAD e operações

- **TAD**: Array de elementos
  - Deve satisfazer ser uma árvore quase completa.
  - Os filhos são menores ou iguais ao seu pai.
  - Elemento de maior prioridade está na raiz.





- **Critério lógico:** se um nó está na posição $\boldsymbol{i}$, seus filhos estarão nas posições $\boldsymbol{2i+1}$ e $\boldsymbol{(2i+2)}$.

Heap: TAD e operações

```
//---------------------------
//Tipo elemento
typedef struct {
    int chave;
    //char nome[100]; //demais atributos
} Tipo_elem;

//Fila de prioridade
typedef struct {
    Tipo_elem *A;
    int n;          //elementos preenchidos
    int tam;        //tamanho máximo de A
} Fila_pri;
//---------------------------
typedef Fila_pri *fp;
```

Heap: TAD e operações

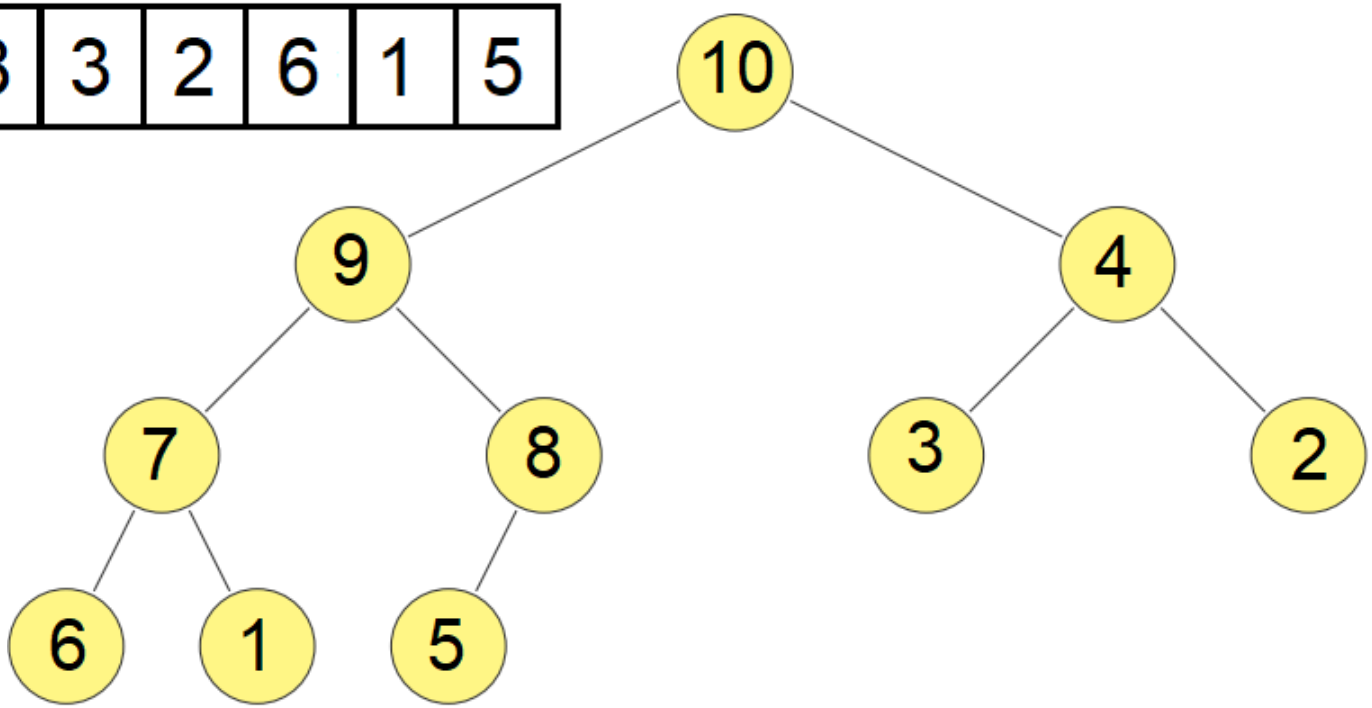
```c
//Permuta dois elementos (por referência, via ponteiros)
void Permuta (Tipo_elem *a, Tipo_elem *b) {
    Tipo_elem aux = *a;
    *a = *b;
    *b = aux;
}
```

```c
//Funções auxiliares
#define Pai(i) ((i-1)/2)
#define F_esq(i) (2*i+1) //Filho esquerdo de i
#define F_dir(i) (2*i+2) //Filho direito de i
```

Montagem de árvore de heap máximo

- Nós da árvore gerados sequencialmente, **da raiz para os níveis mais baixos**, **da esquerda para a direita**.
- Filhos **são sempre menores ou iguais ao seus pais.**

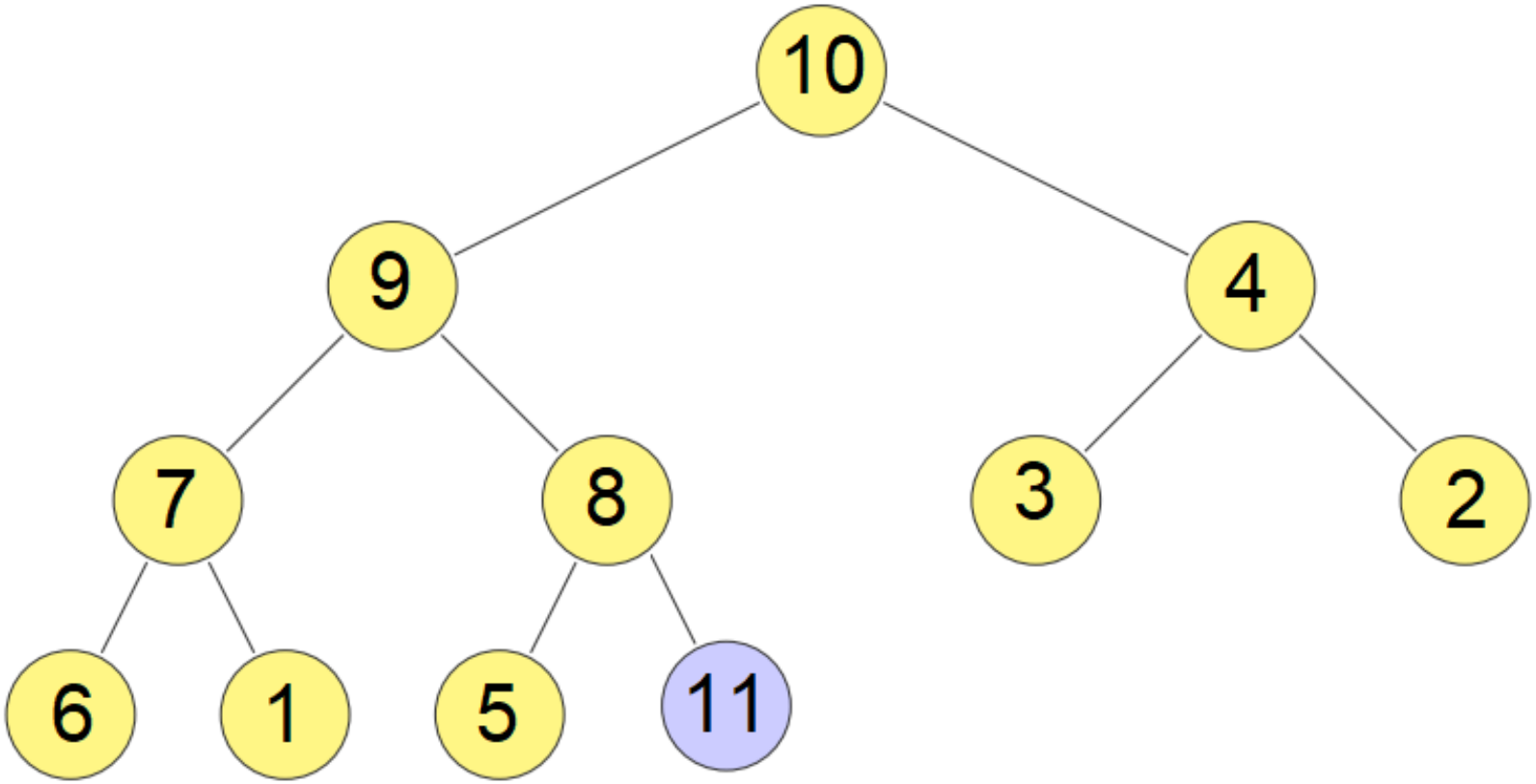
0	1	2	3	4	5	6	7	8	9
10	9	4	7	8	3	2	6	1	5



Operação de inserção

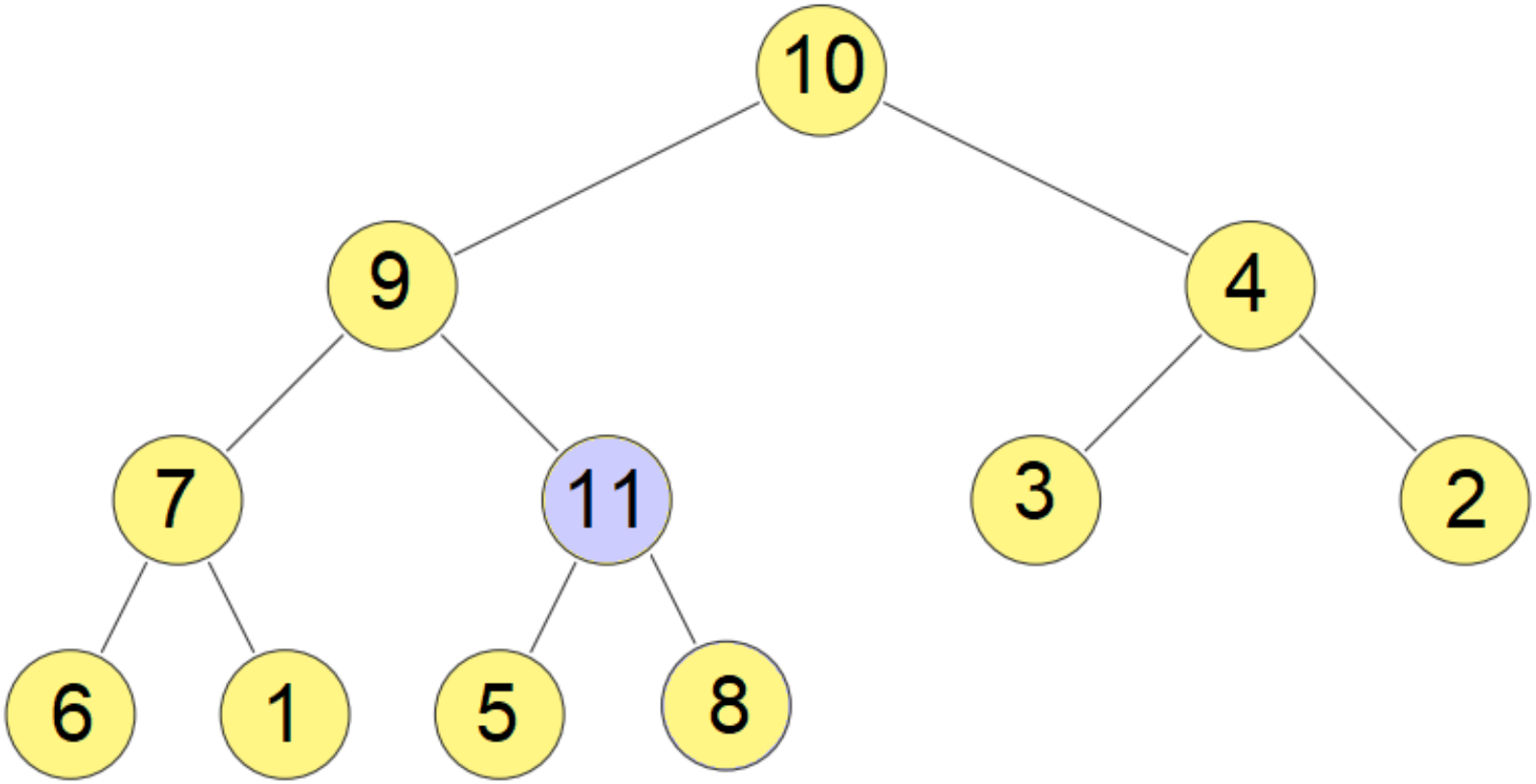
Inserção no heap

- Basta ir subindo no heap, permutando com o pai se necessário.
- Na heap: o nó = 11 foi para a posição v[10] do array



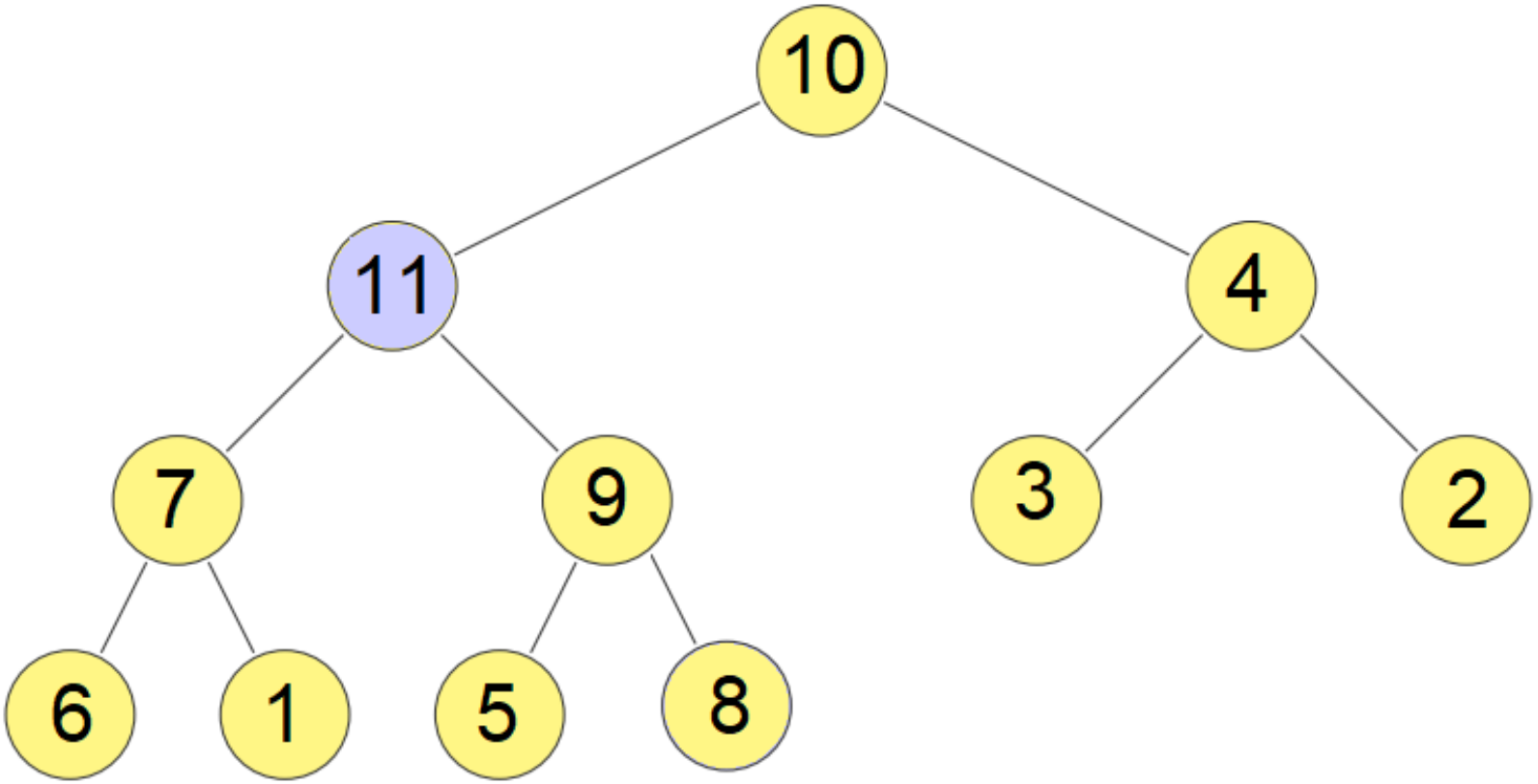
Inserção no heap

- Basta ir subindo no heap, permutando com o pai se necessário.
- Nó 11 deve ser trocado com o nó 9.



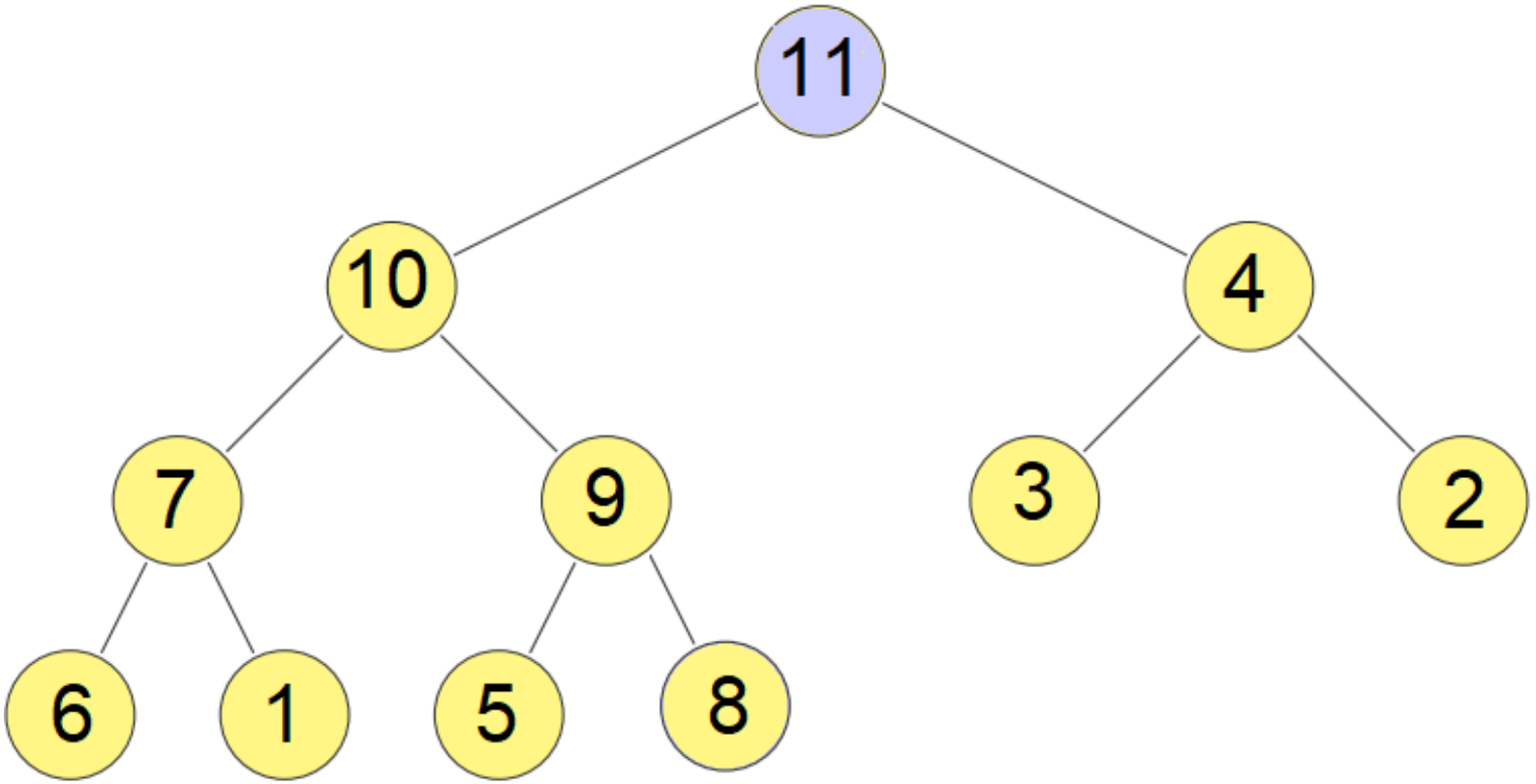
Inserção no heap

- Basta ir subindo no heap, permutando com o pai se necessário.
- Nó 11 deve ser trocado com o nó 10.



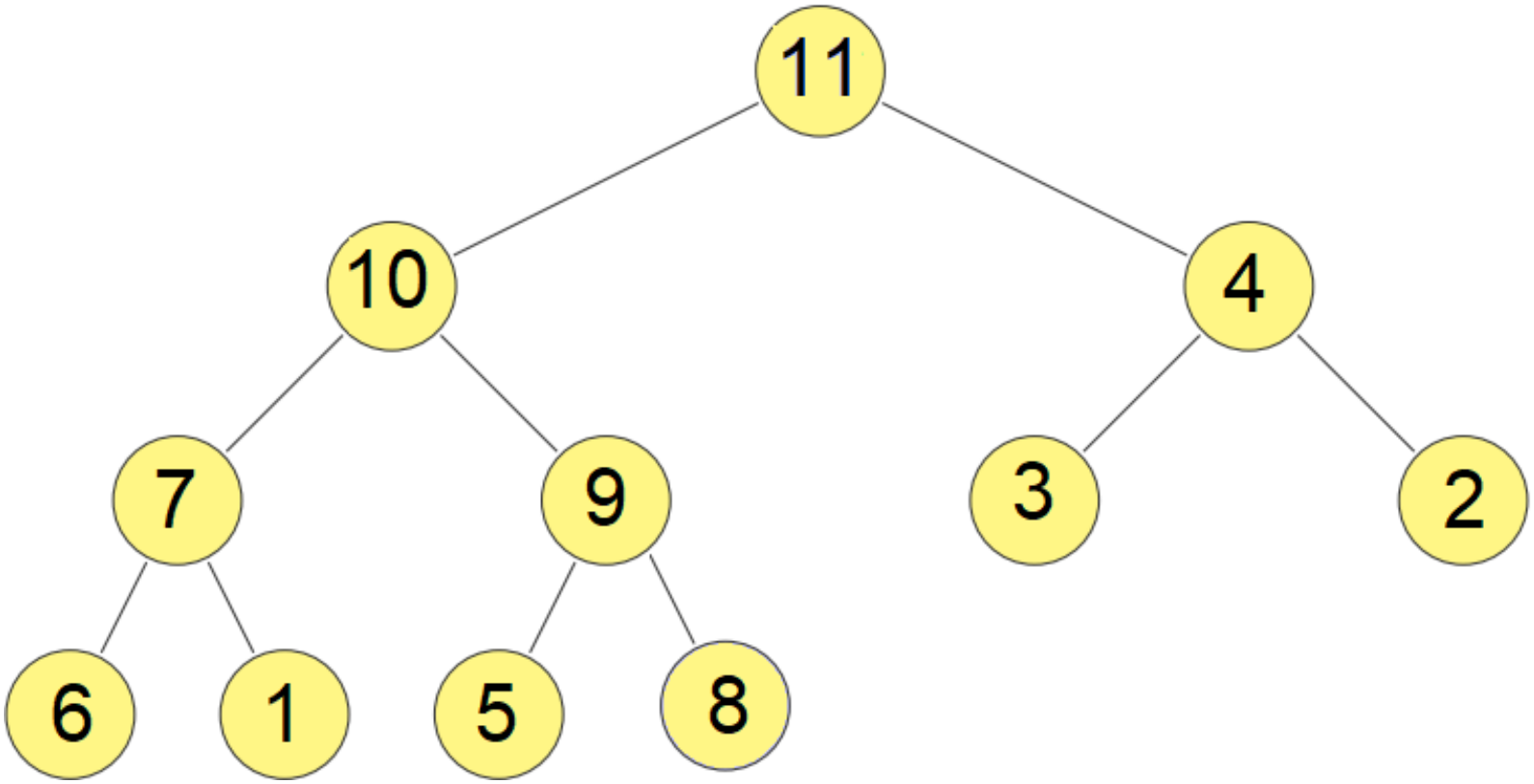
Inserção no heap

- Basta ir subindo no heap, permutando com o pai se necessário.
- Chegou na raiz, então para.



Inserção no heap

- **Critério de parada:** subimos até que o nó permutado pare de violar a propriedade de heap (de prioridade).



Heap: inserção

```
//Insere um novo elemento na fp
void Insere (fp filapri, Tipo_elem elem) {
    filapri->A[filapri->n] = elem;
    filapri->n++;
    Sobe_no_heap (filapri, filapri->n-1);
}

//Avalia se é necessário permutar o índice corrente k e seu pai.
void Sobe_no_heap (fp filapri, int k) {
    if (k > 0 && filapri->A[Pai(k)].chave < filapri->A[k].chave){
        Permuta (&filapri->A[k], &filapri->A[Pai(k)]);
        Sobe_no_heap (filapri, Pai(k)); //Sobe um nível
    }
}
```

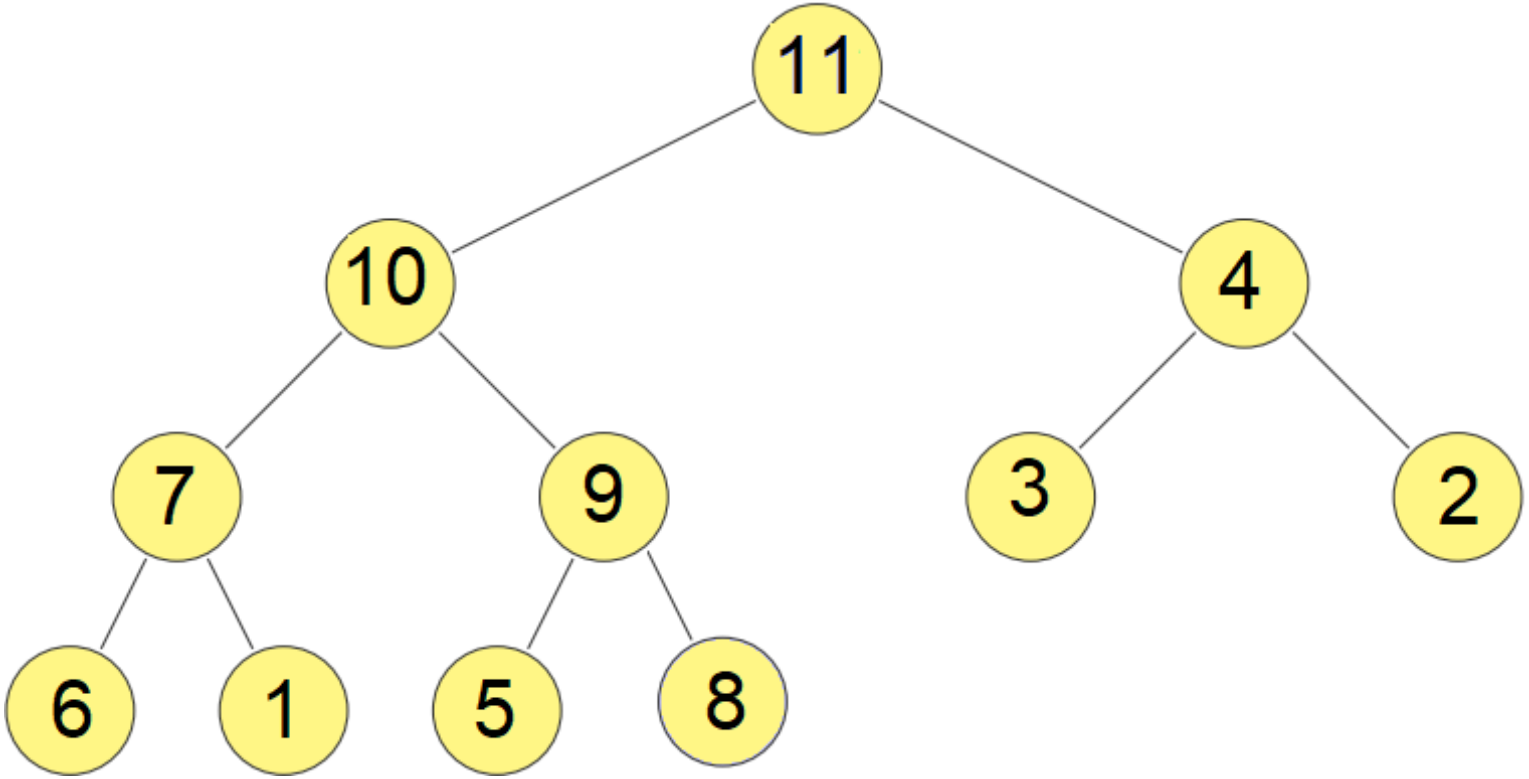
Tempo do insere:

No máximo subimos até a raiz ➔ O(log n)

Operação de remoção

Remoção (extração do maior) no heap

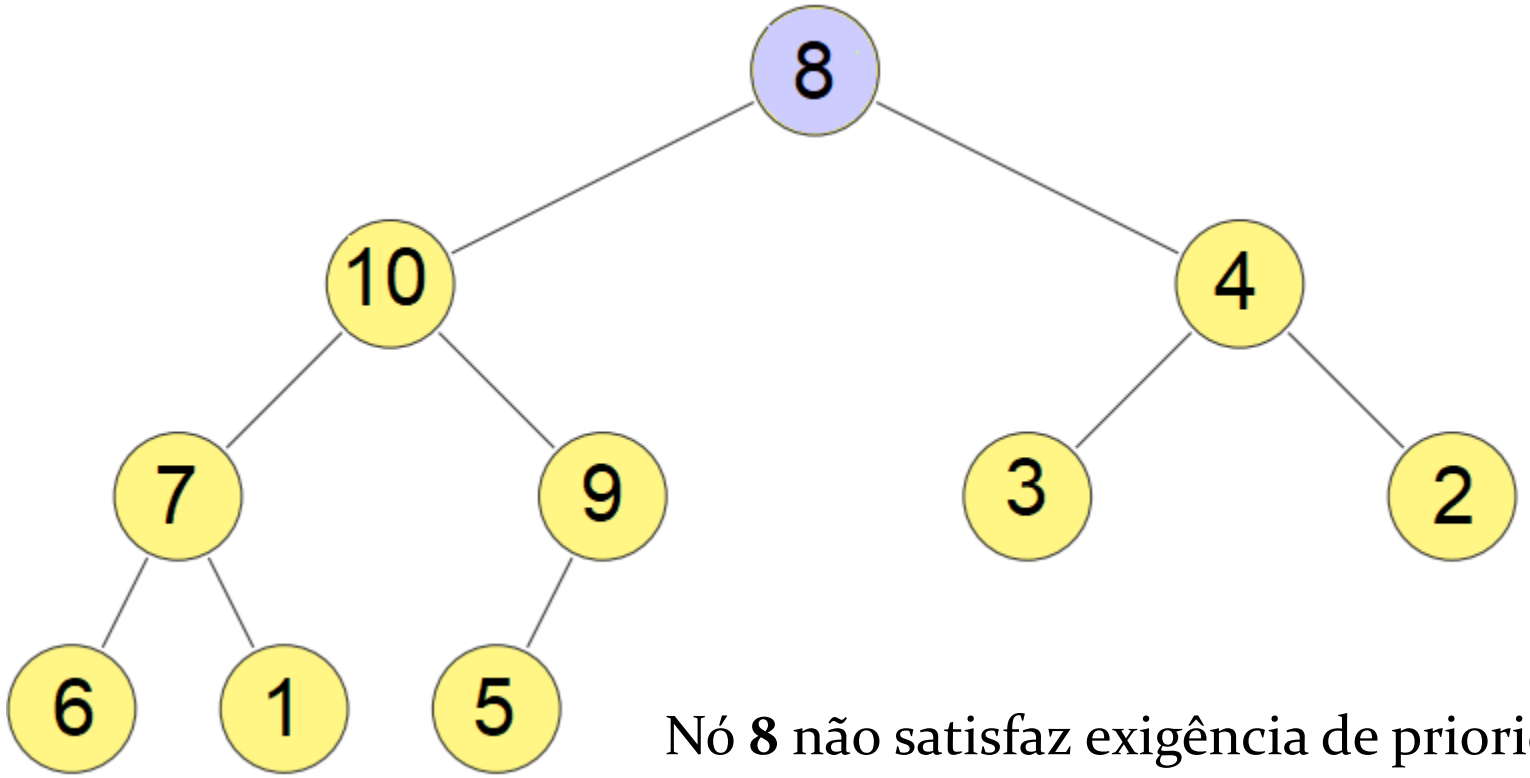
- Permutamos a raiz com o último elemento do heap, e então removemos o antigo nó raiz.
- Descemos no heap fazendo os ajustes necessários:
  - Permutamos o pai com o maior dos dois filhos.



Remoção (extração do maior) no heap

- Descemos na heap fazendo os ajustes necessários

  -Permutamos o pai com o maior dos dois filhos.

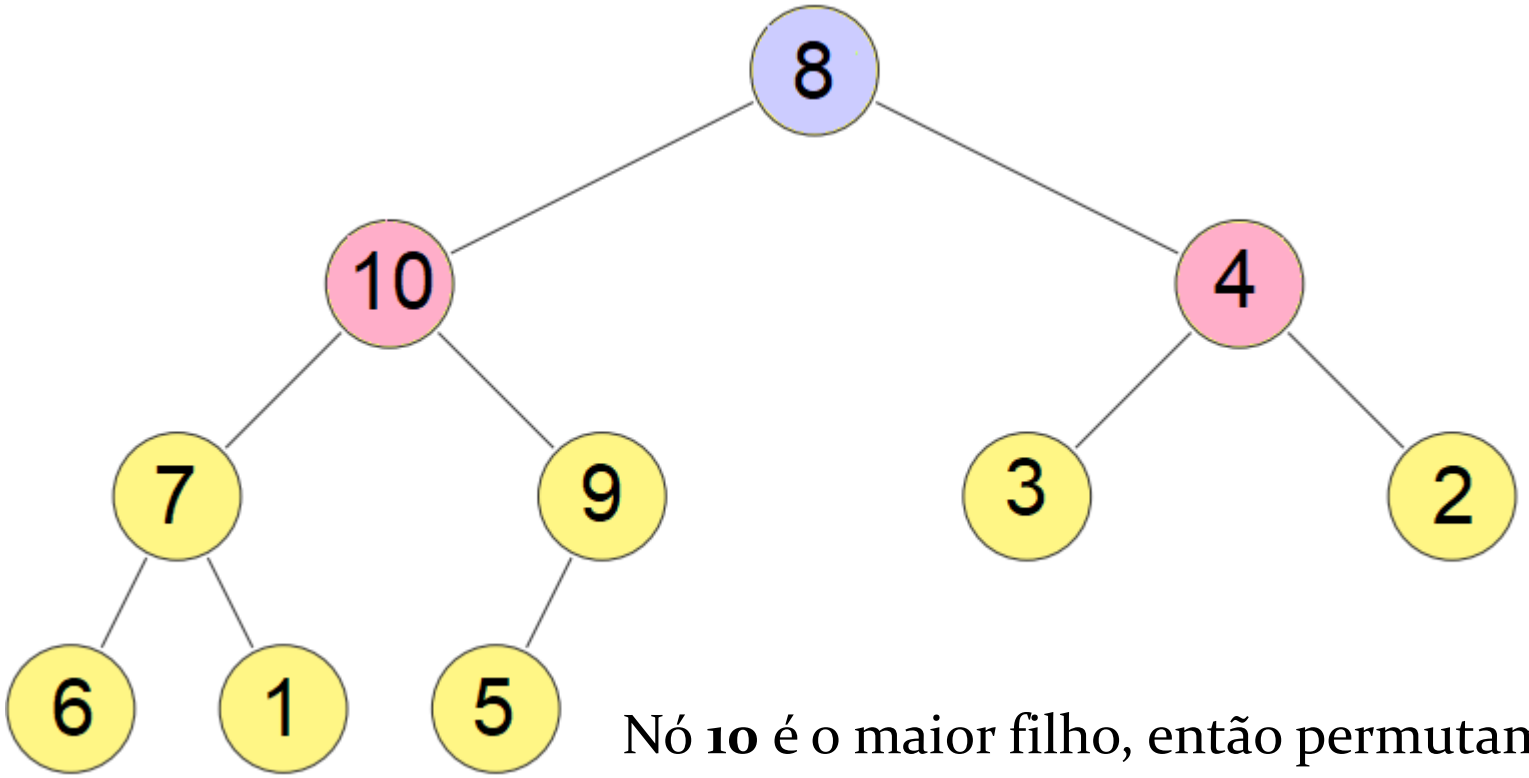


Nó **8** não satisfaz exigência de prioridade!

Remoção (extração do maior) no heap

- Descemos na heap fazendo os ajustes necessários

  -Permutamos o pai com o maior dos dois filhos.

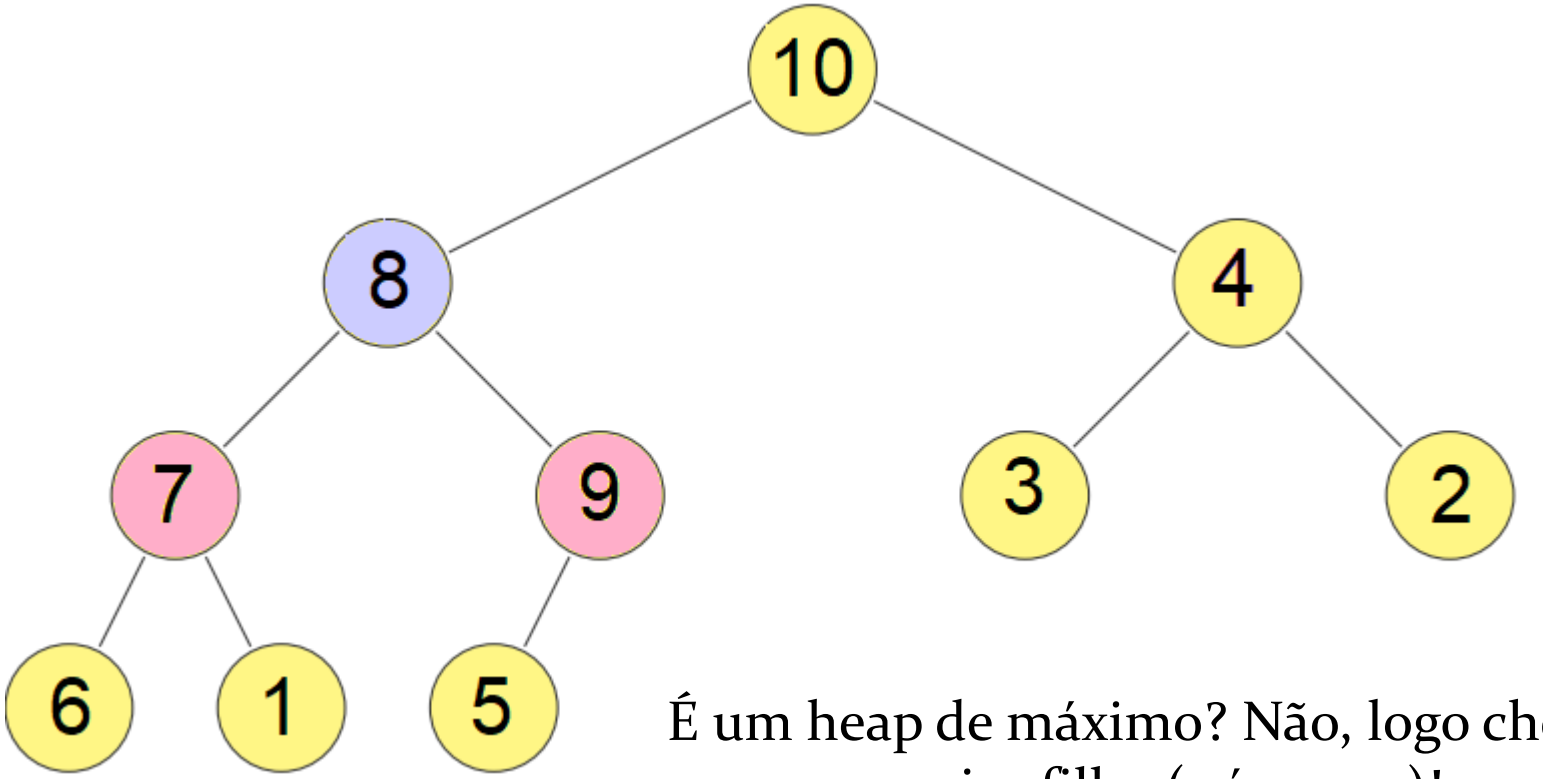


Nó **10** é o maior filho, então permutamos ele

Remoção (extração do maior) no heap

- Descemos na heap fazendo os ajustes necessários

  -Permutamos o pai com o maior dos dois filhos.

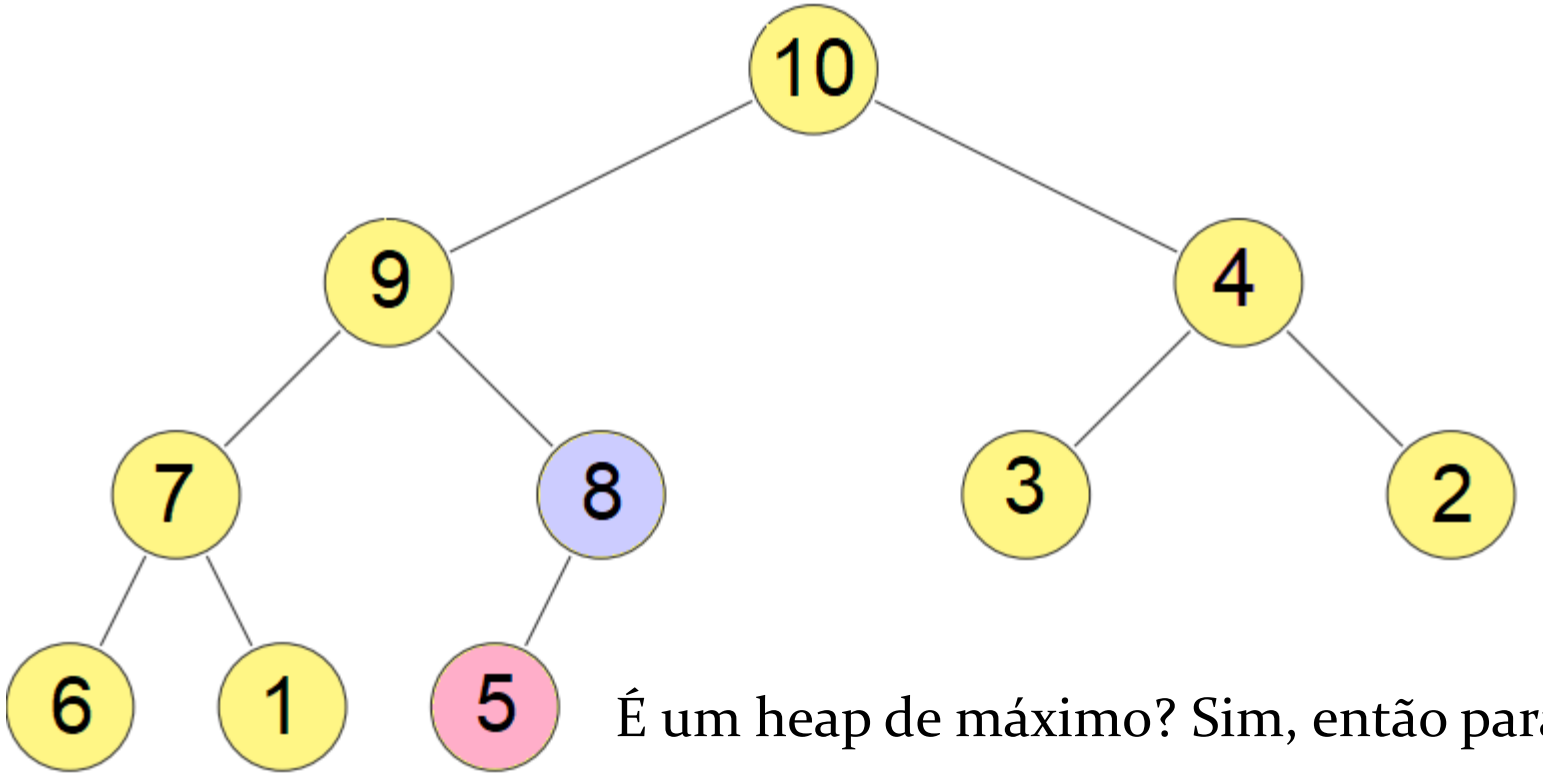


É um heap de máximo? Não, logo checa o maior filho (nós rosas)!

Remoção (extração do maior) no heap

- Permutamos a raiz com o último elemento da heap.
- Descemos na heap fazendo os ajustes necessários

  -Permutamos o pai com o maior dos dois filhos.

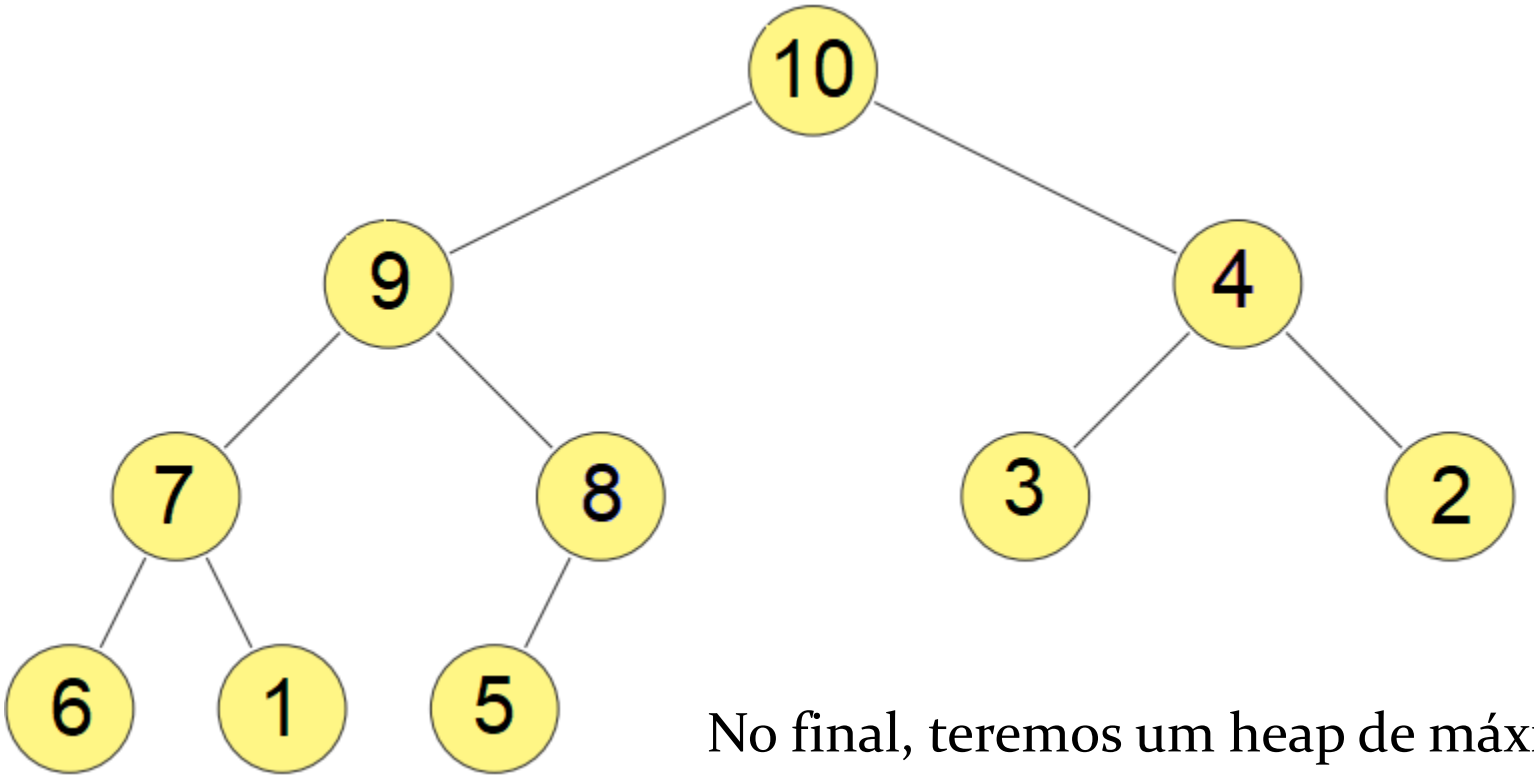


É um heap de máximo? Sim, então paramos!

Remoção (extração do maior) no heap

- Permutamos a raiz com o último elemento da heap.
- Descemos na heap fazendo os ajustes necessários

  -Permutamos o pai com o maior dos dois filhos.



No final, teremos um heap de máximo!

Heap: remoção

```
//Remove o elemento de maior prioridade
Tipo_elem Remove_maior (fp filapri) {
    Tipo_elem aux = filapri->A[0]; //Copia maior
    //Permuta maior com último (pq é mais fácil remover o último)
    Permuta (&(filapri->A[0]), &(filapri->A[filapri->n-1]));
    filapri->n--;

    Desce_no_heap (filapri, 0);
    return aux;
}
```

Heap: remoção

```
void Desce_no_heap (fp filapri, int k) {
    int maior_filho;
    if (F_esq(k) < filapri->n) {
        maior_filho = F_esq(k); //palpite
        if ( F_dir(k) < filapri->n &&
             filapri->A[F_esq(k)].chave < filapri->A[F_dir(k)].chave )
            maior_filho = F_dir(k);
        if (filapri->A[k].chave < filapri->A[maior_filho].chave) {
            Permuta (&filapri->A[k], &filapri->A[maior_filho]);
            Desce_no_heap (filapri, maior_filho);
        }
    }
}
```

Tempo da remoção: O(log n)

Alteração de prioridade de um elemento

- Se a prioridade de algum elemento:

1. Aumentar, precisamos subir no heap.
   - Cai no caso da operação de inserção.

2. Diminuir, precisamos descer no heap.
   - Cai no caso da operação de remoção.

Heap: alteração de prioridade

```
//Altera a prioridade de um elemento (k)
void Altera_prioridade (fp filapri, int k, int valor){
    if (filapri->A[k].chave < valor) {
        filapri->A[k].chave = valor;
        Sobe_no_heap (filapri, k);
    }
    else {
        filapri->A[k].chave = valor;
        Desce_no_heap (filapri, k);
    }
}
```

Tempo da alteração: O(log n)

Construção de uma lista de prioridade via heap

Construção de uma lista de prioridades via heap

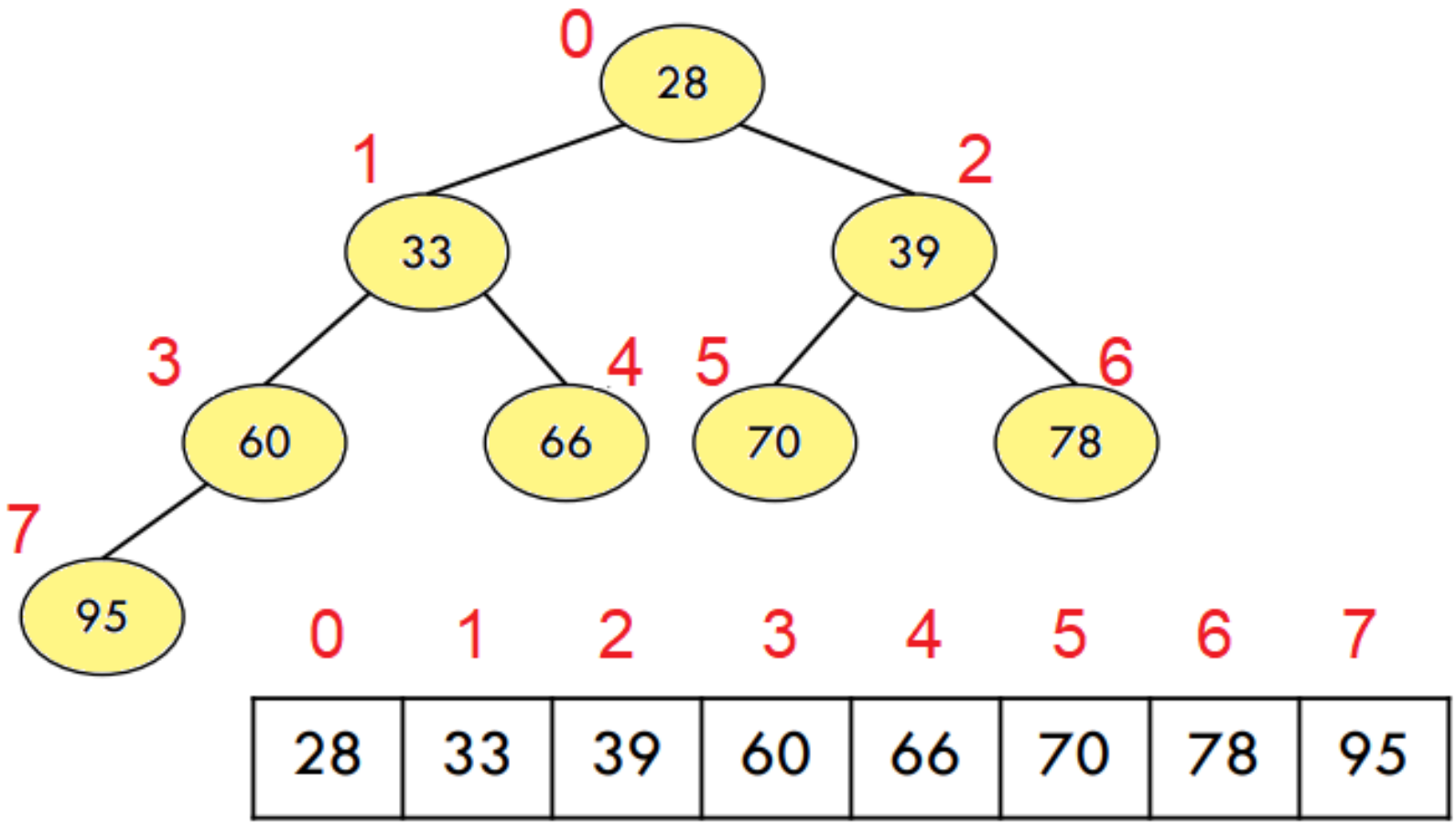
- Seja L uma lista de elementos para qual desejamos construir uma heap H. Há duas estratégias de construção:

1. Inserir os elementos de L, um a um, em uma heap vazia.

2. Considerar que a lista L é uma heap, e ir corrigindo as prioridades conforme necessário.
   - Nesse caso, assume-se que as **prioridades das folhas já estão corretas**, pois elas não têm filhos.
   - Em seguida, deve-se ir ajustando as prioridades dos nós internos, realizando descidas quando necessário.

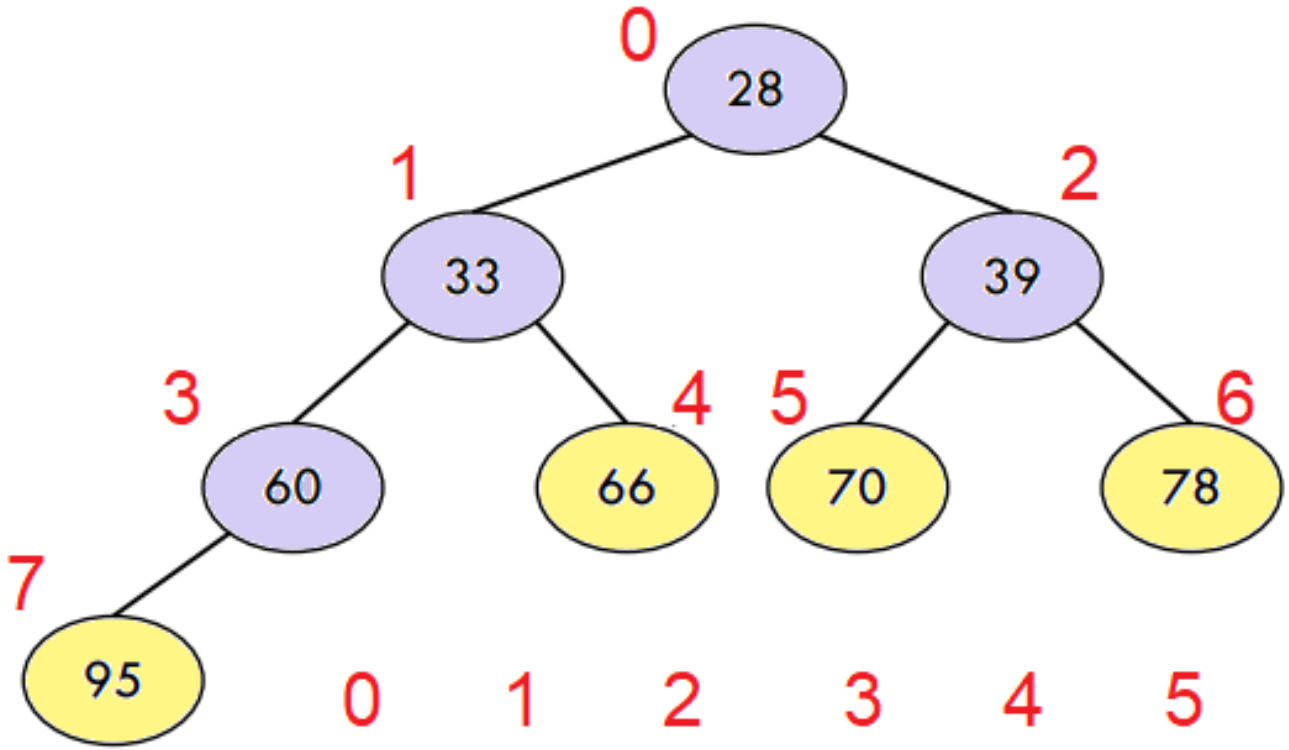
Construção de uma lista de prioridades via heap

- **Exemplo:** construir uma heap de máximo a partir da seguinte lista: {28, 33, 39, 60, 66, 70, 78, 95}.



Construção de uma lista de prioridades via heap

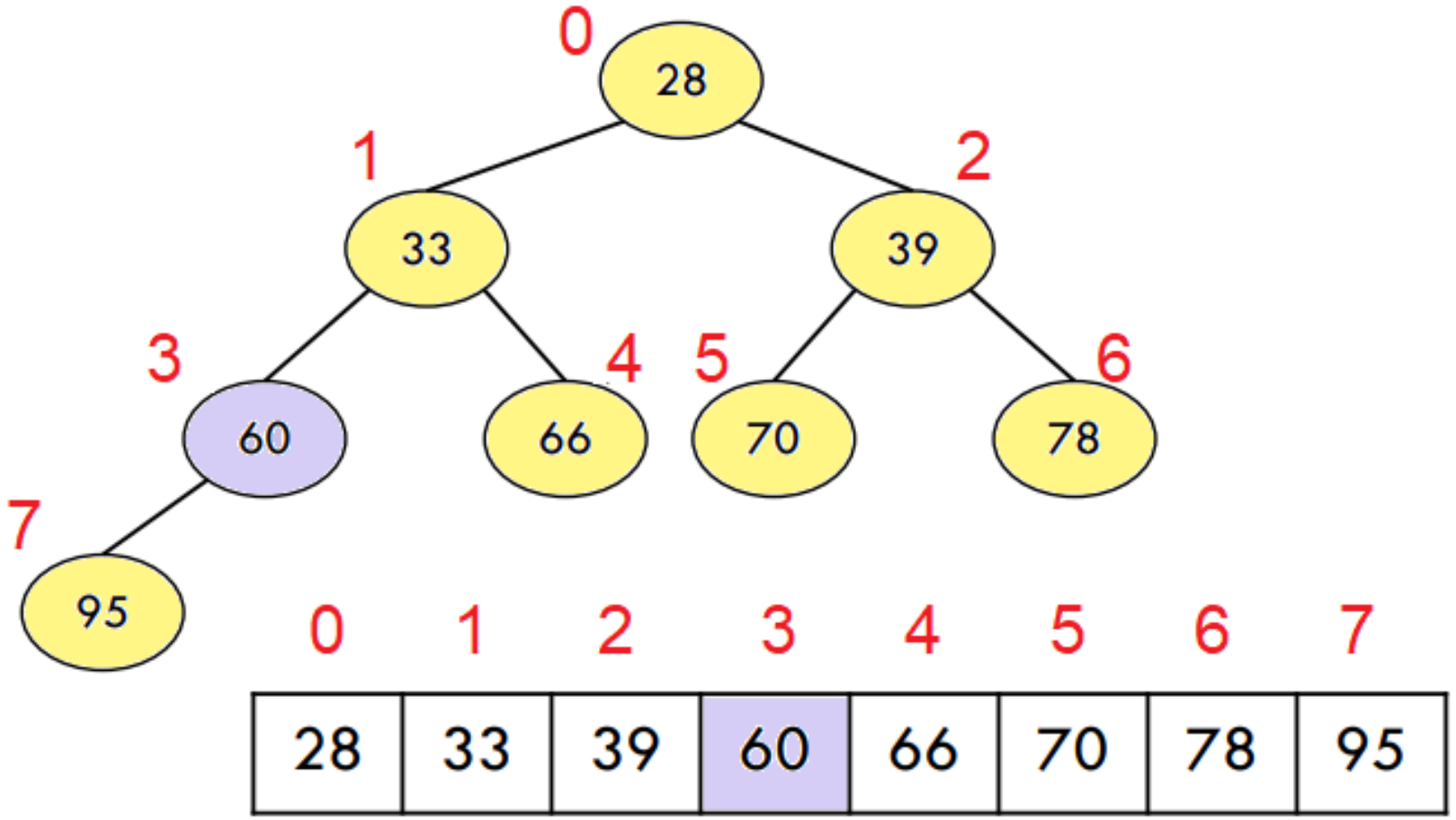
- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



0	1	2	3	4	5	6	7
28	33	39	60	66	70	78	95

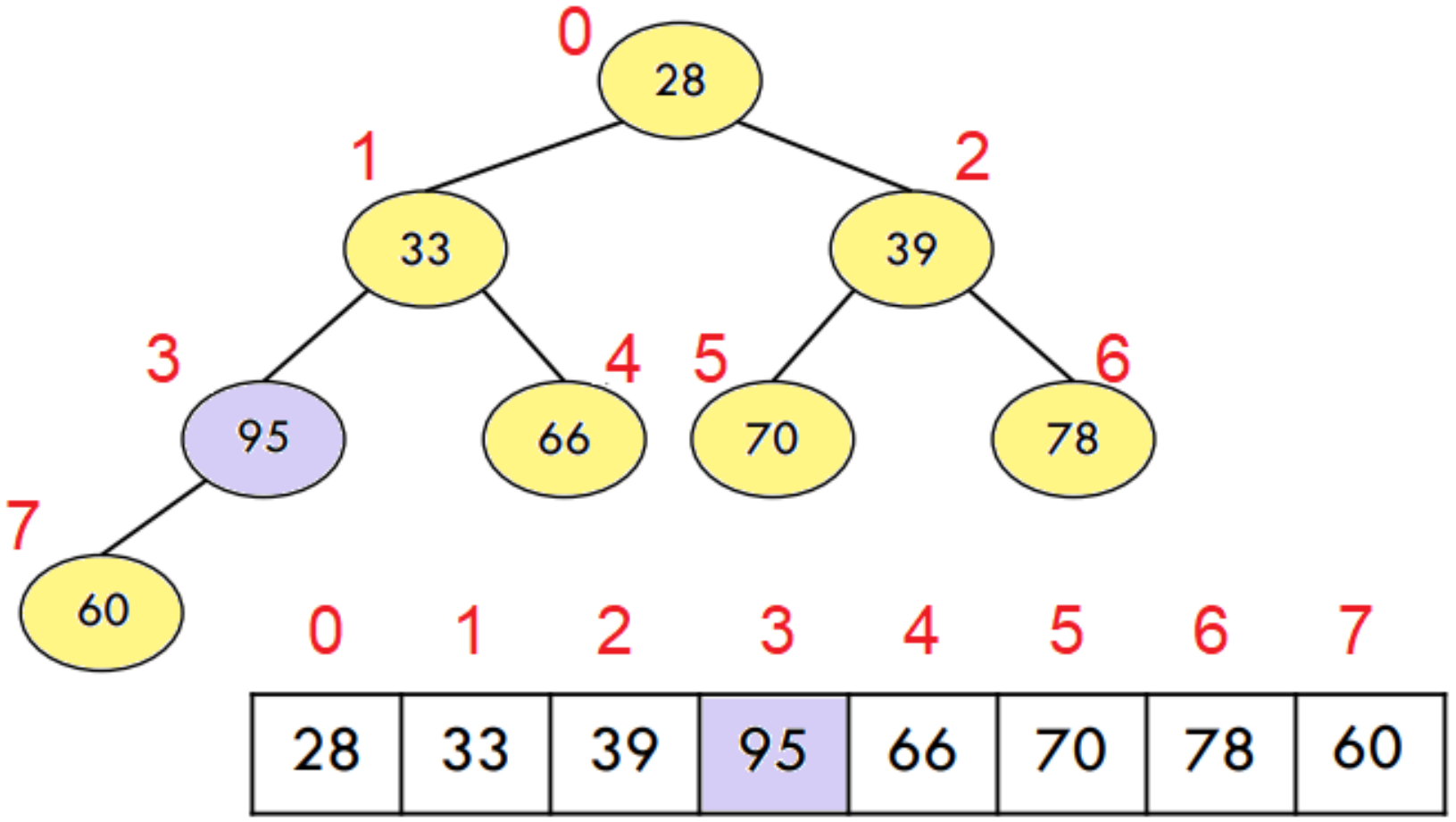
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



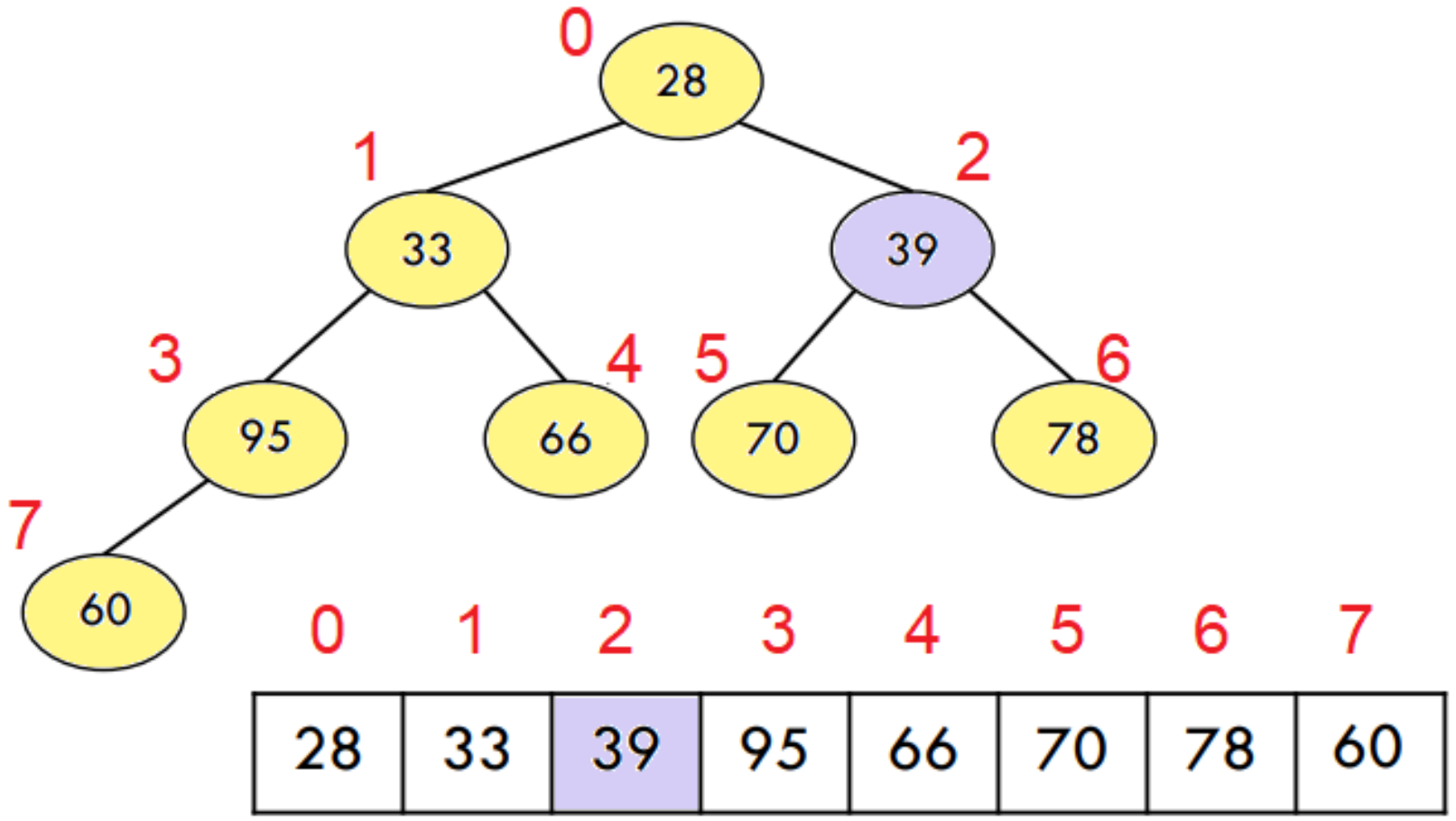
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



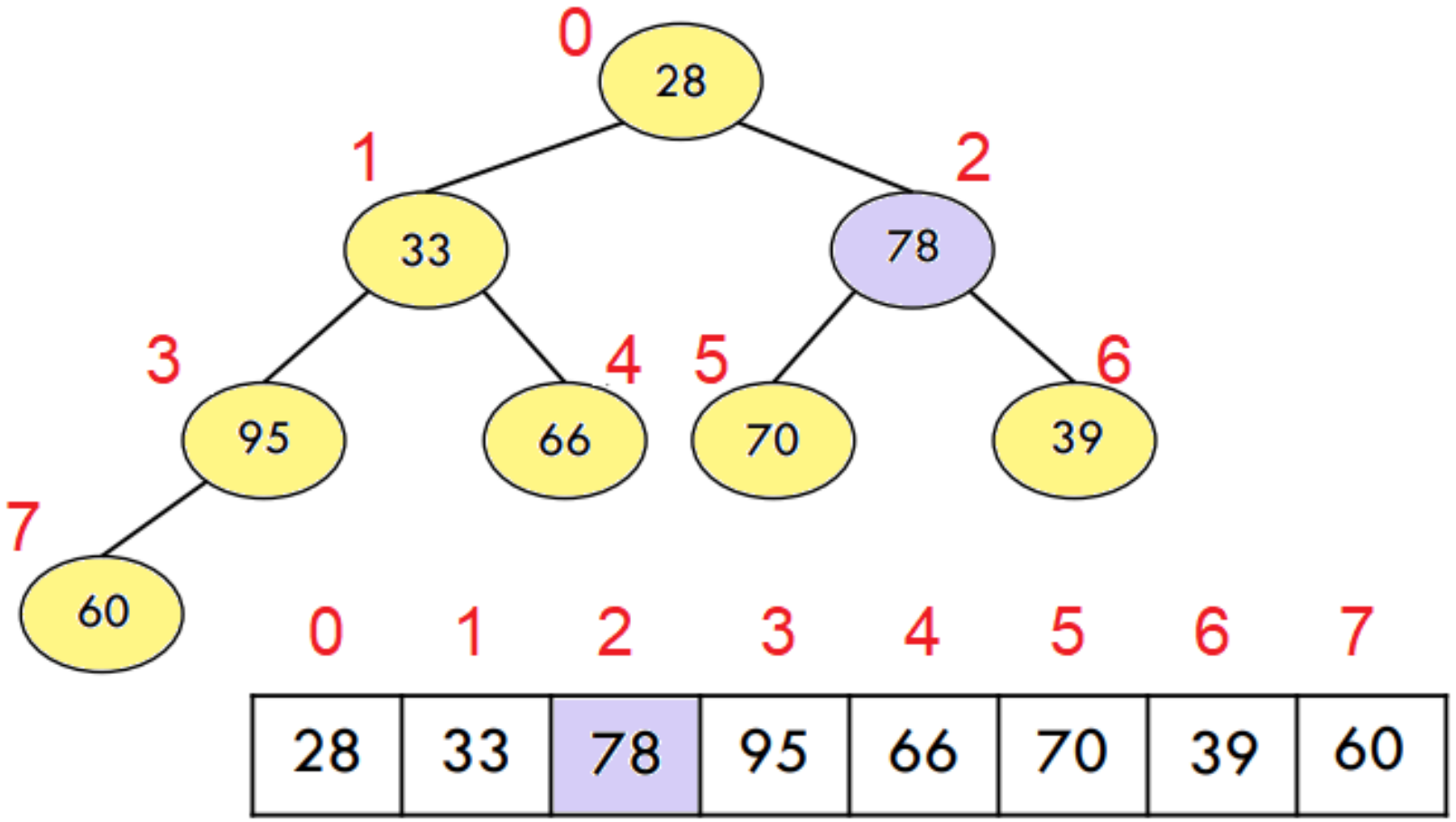
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



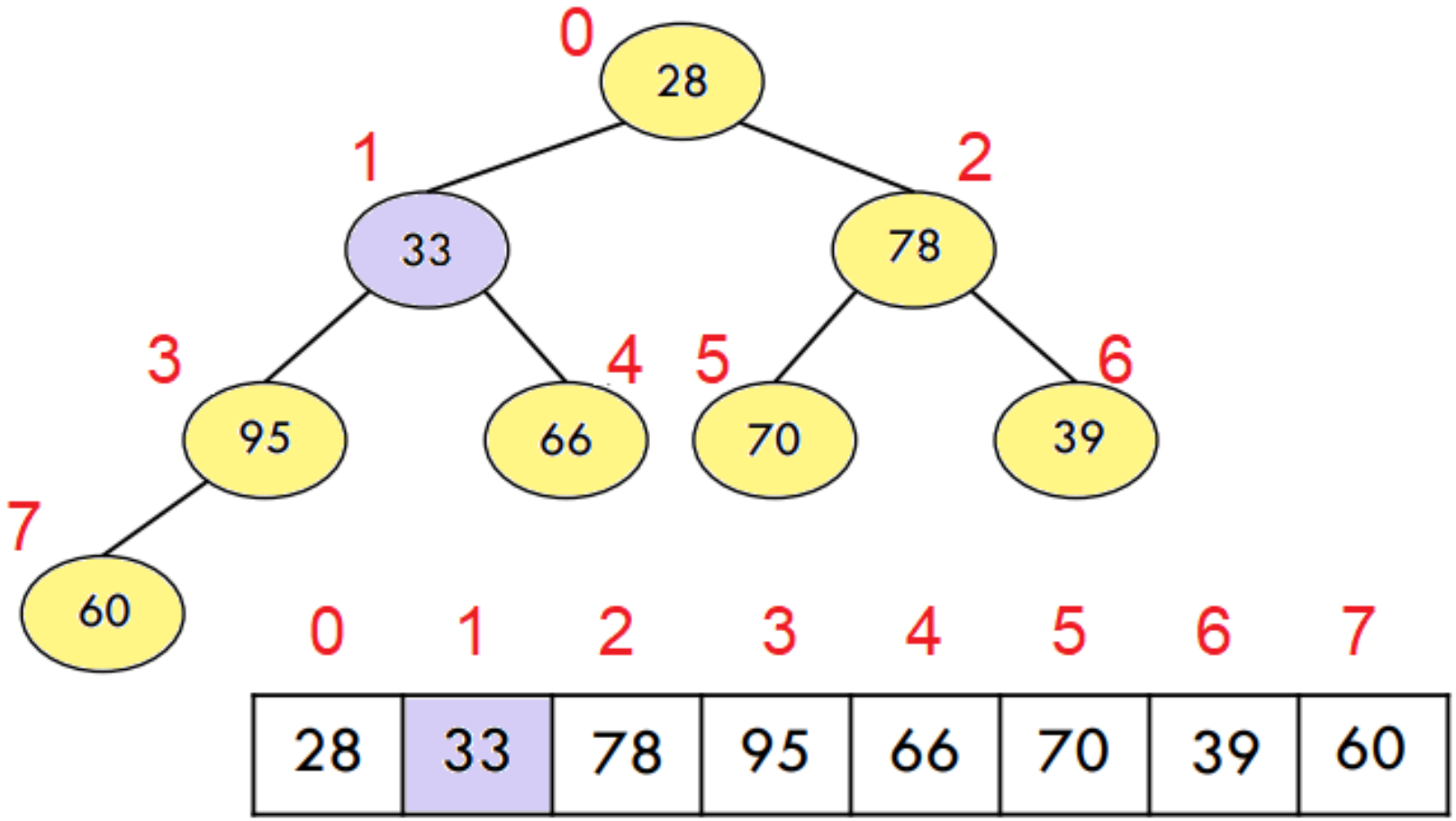
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



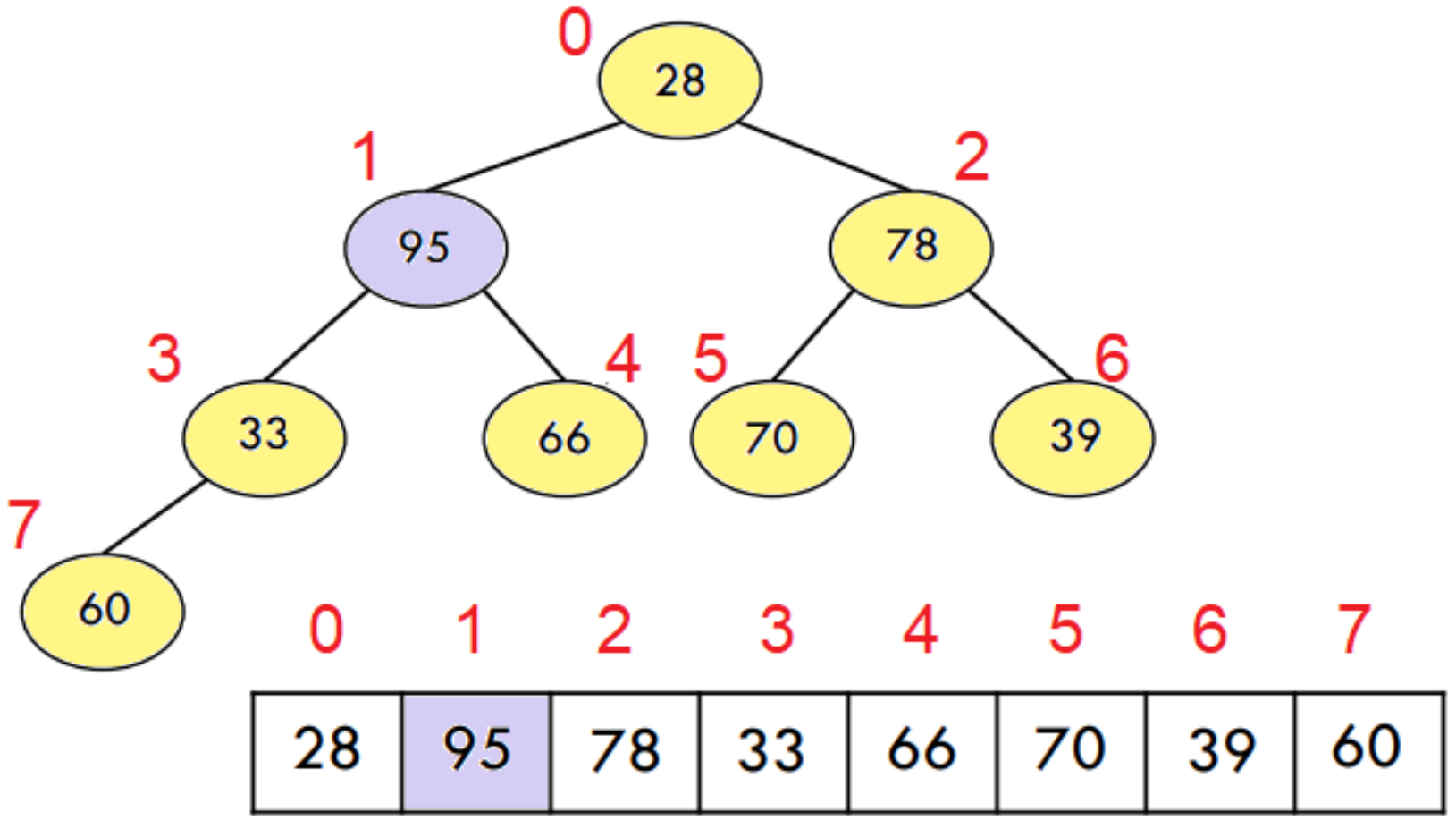
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



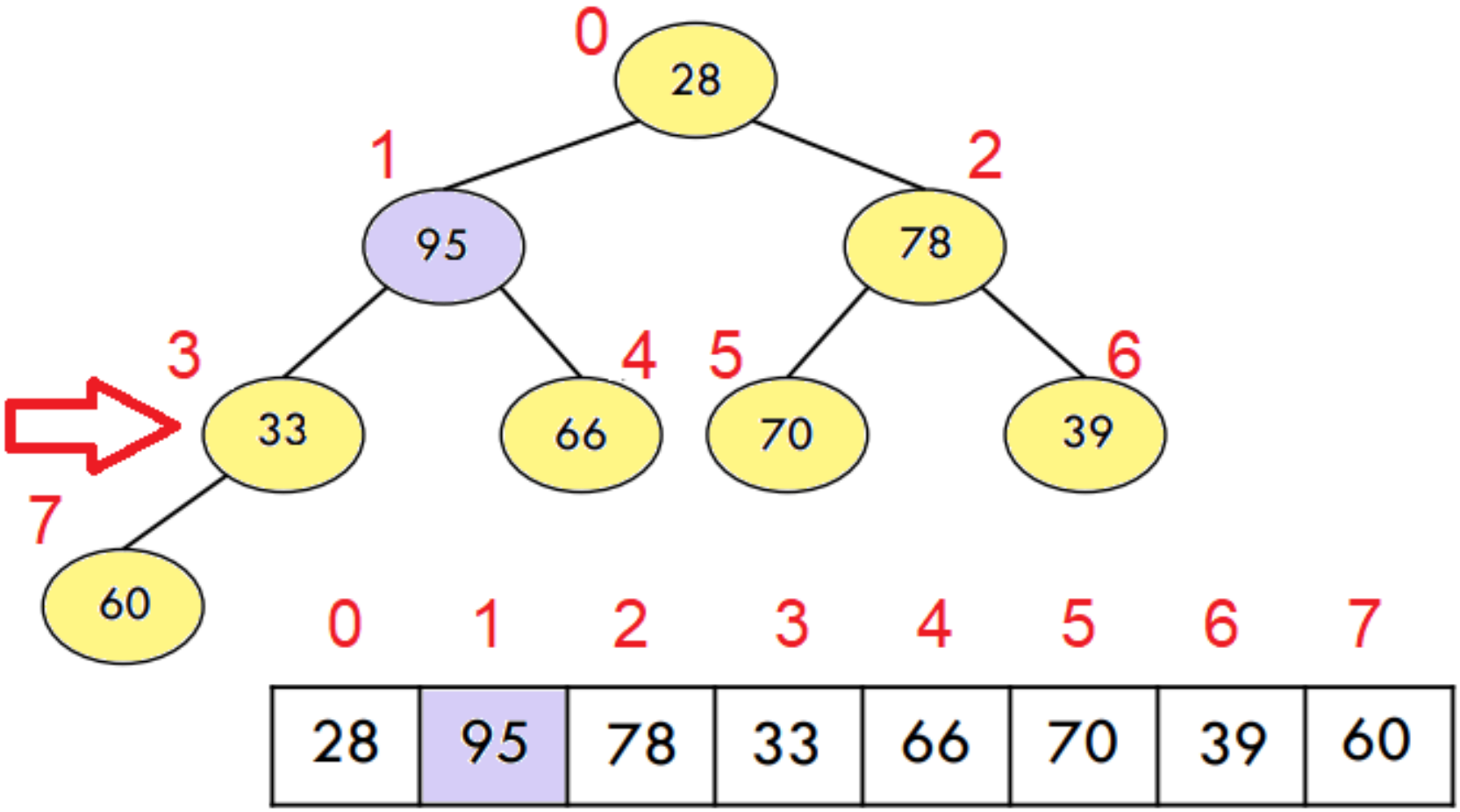
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



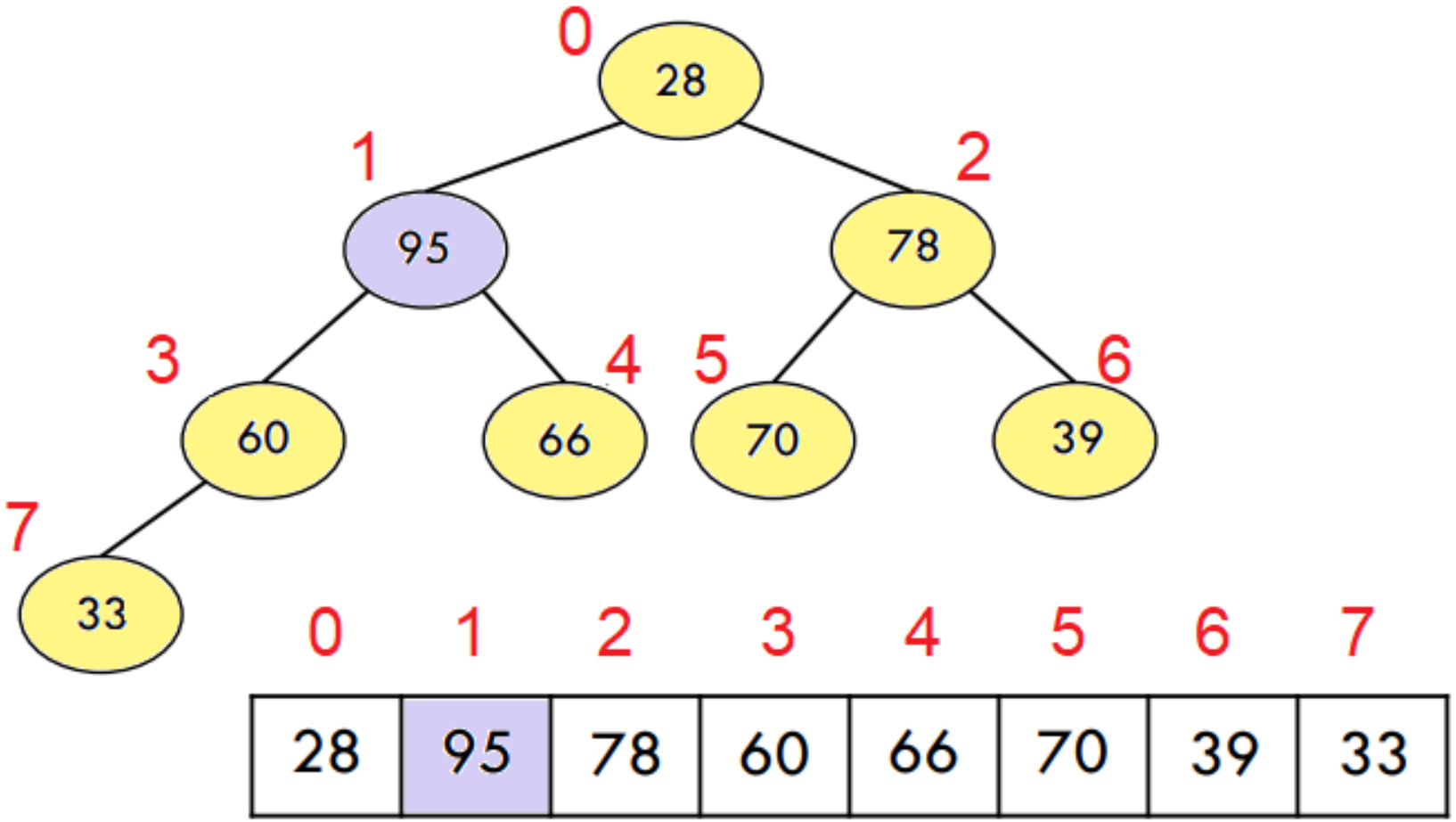
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando **descidas quando necessário**.



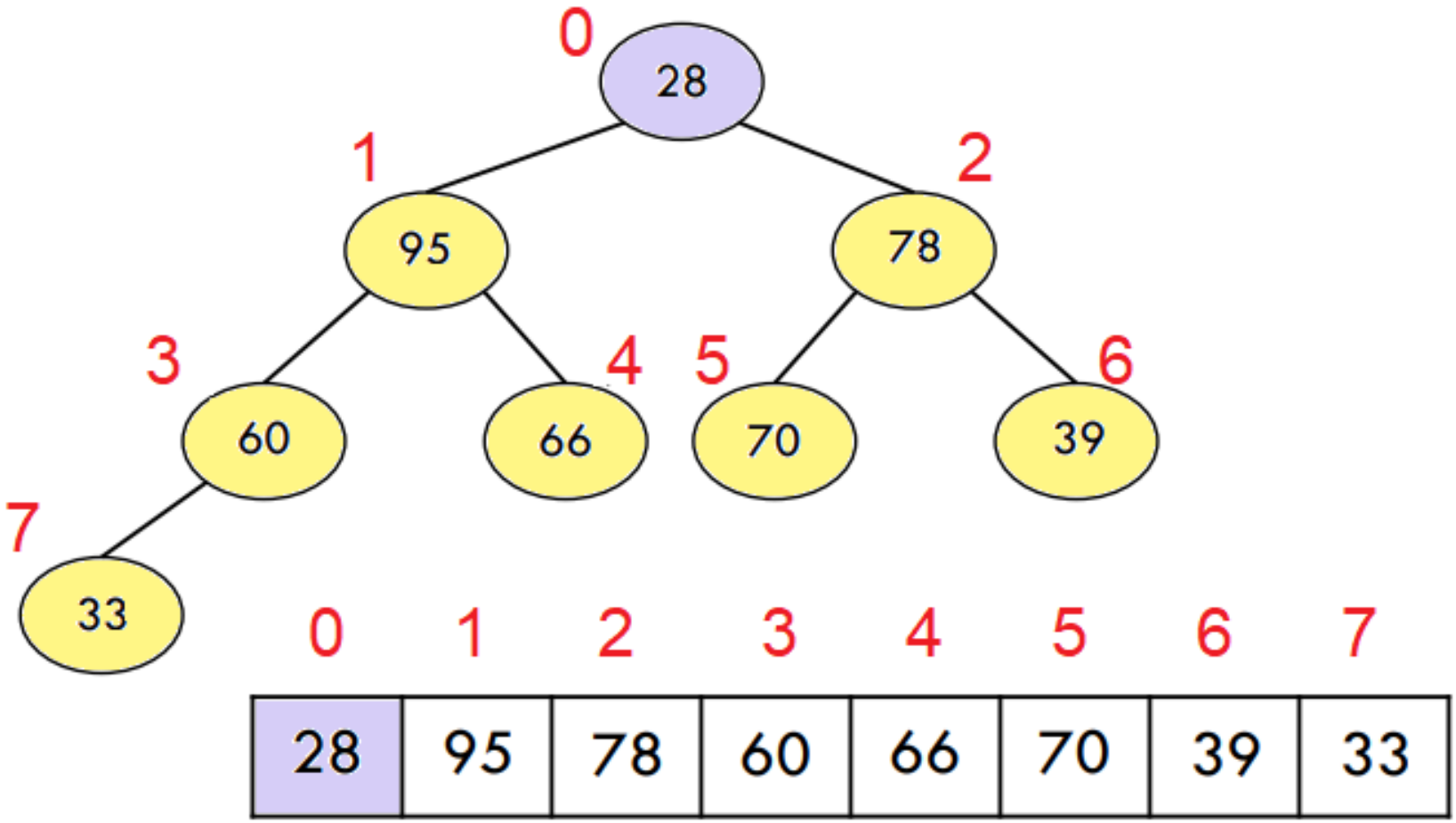
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



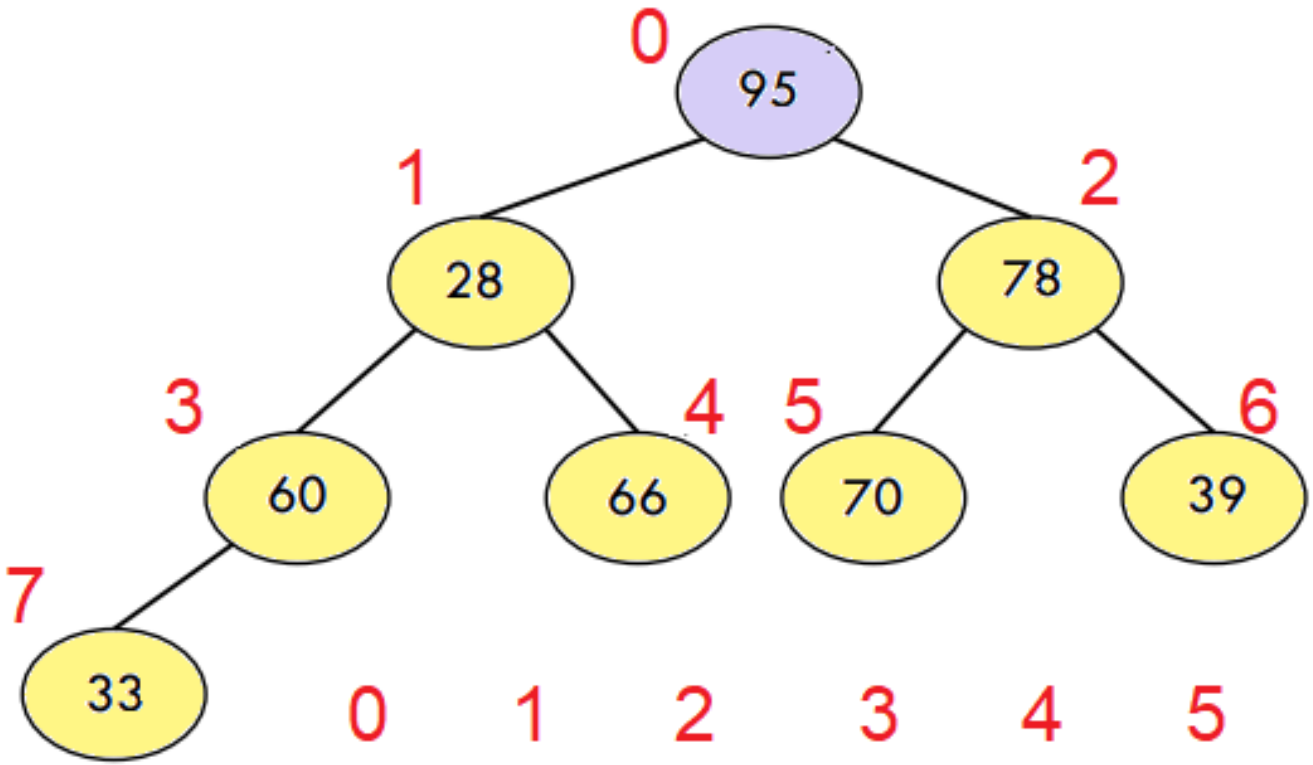
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



Construção de uma lista de prioridades via heap

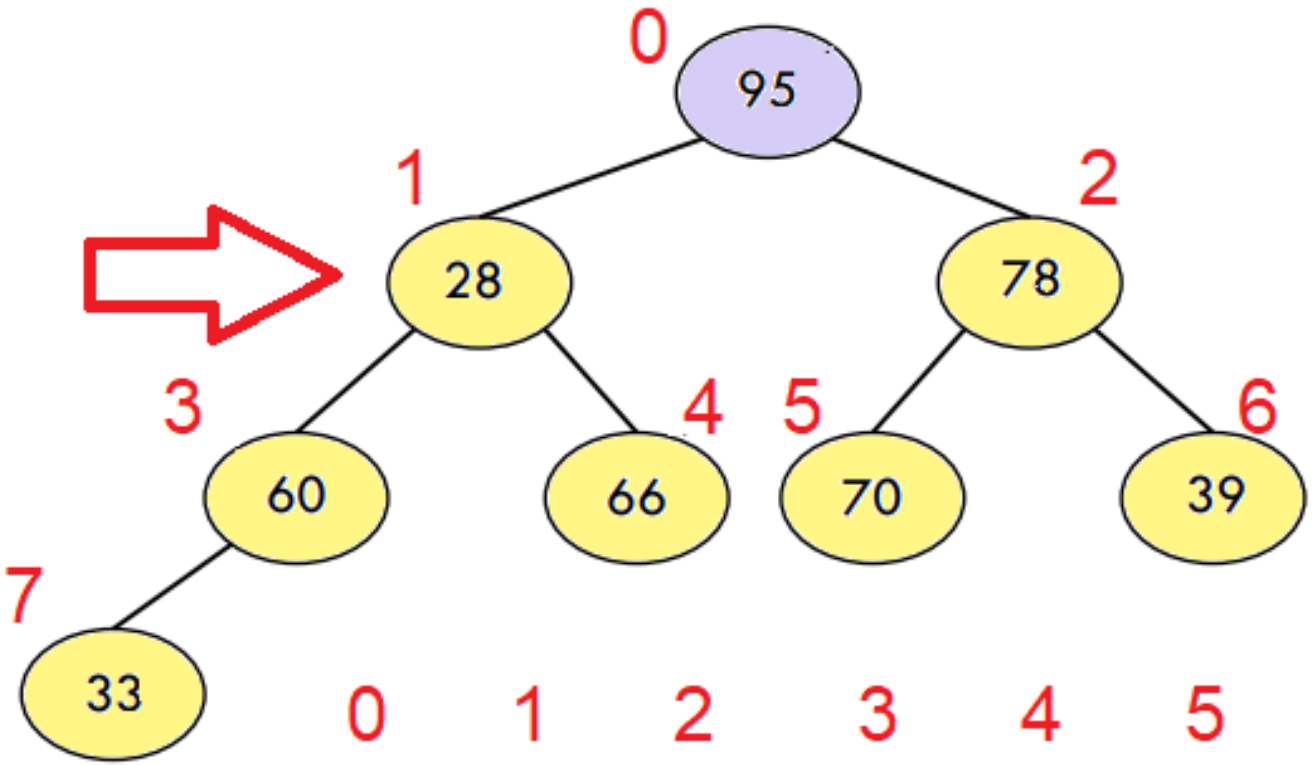
- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando descidas quando necessário.



0	1	2	3	4	5	6	7
95	28	78	60	66	70	39	33

Construção de uma lista de prioridades via heap

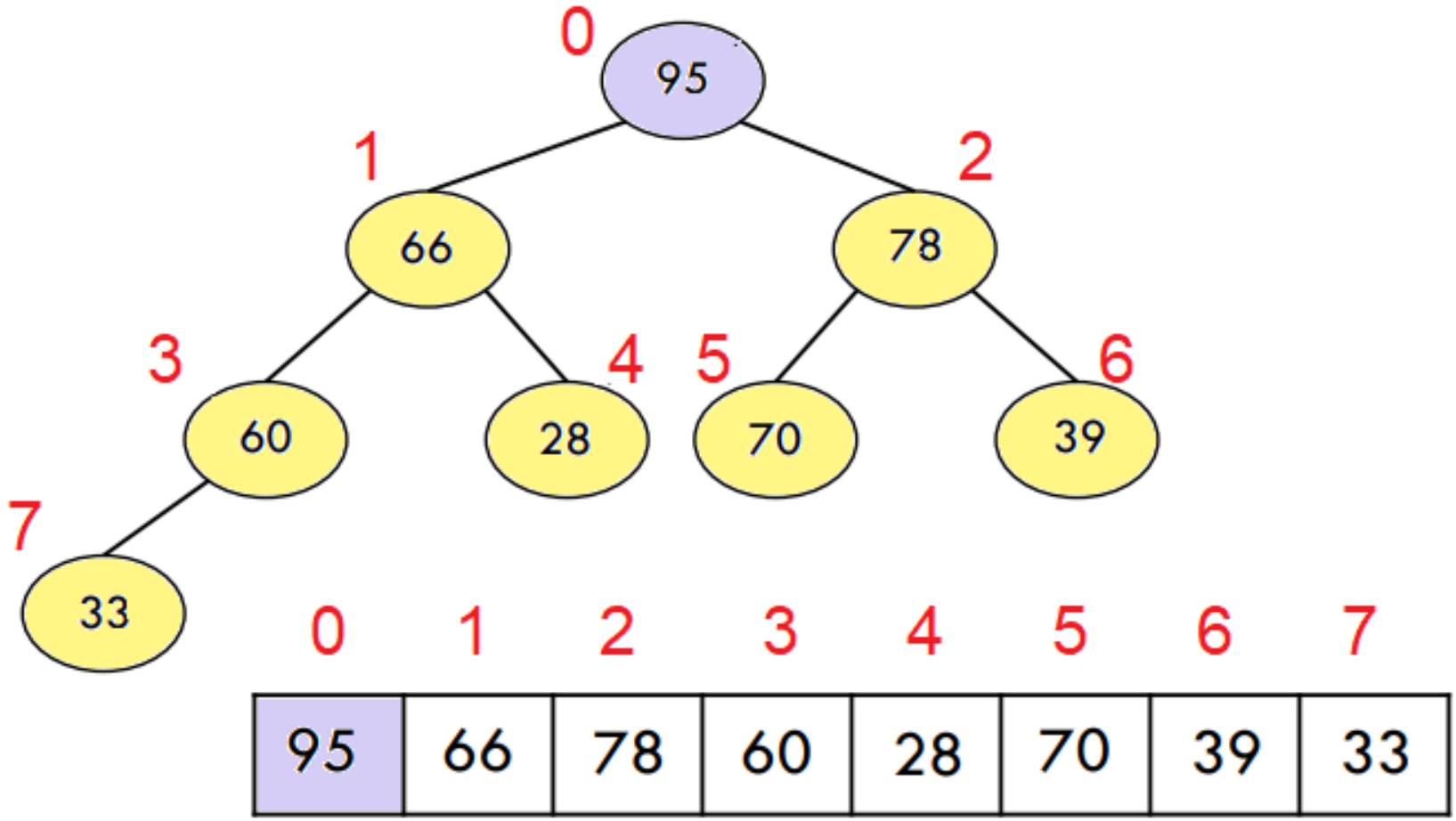
- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando **descidas quando necessário**.



0	1	2	3	4	5	6	7
95	28	78	60	66	70	39	33

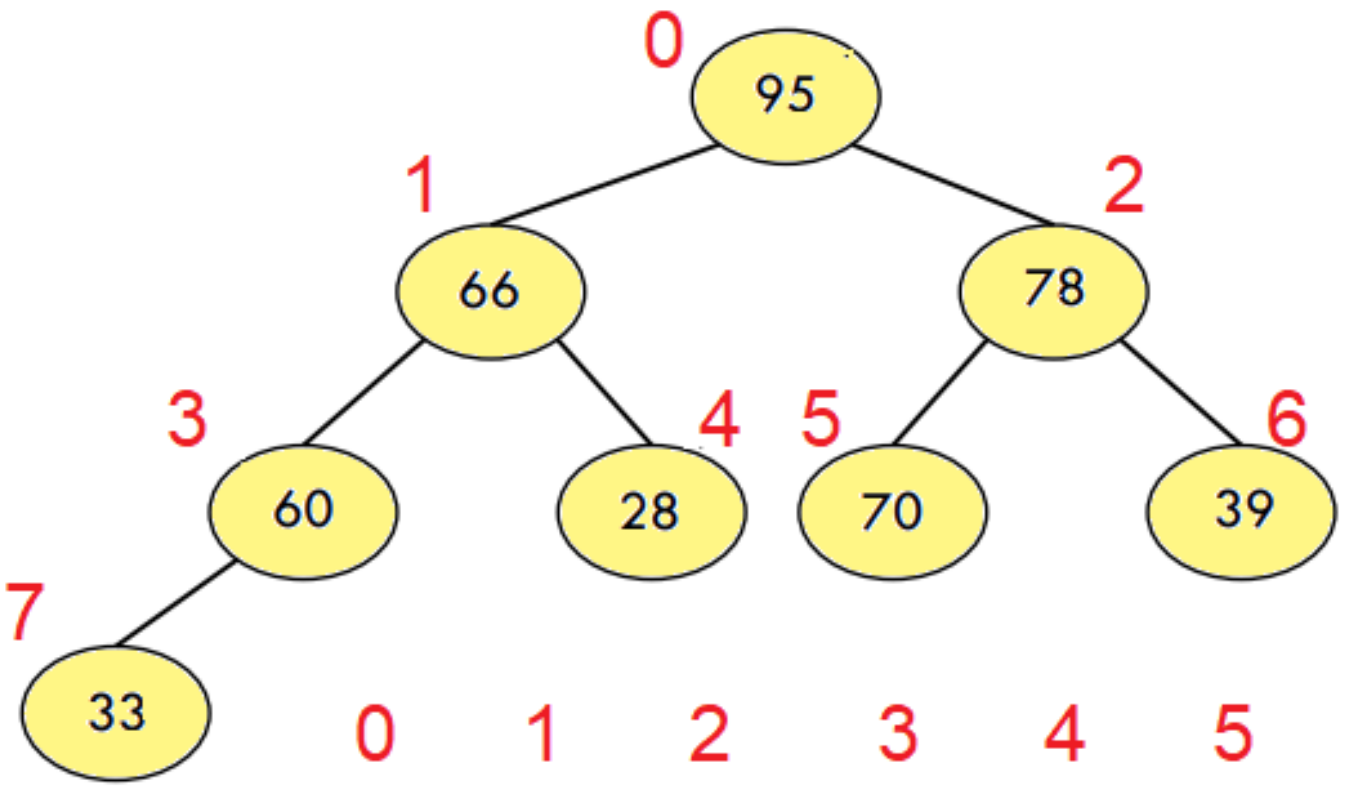
Construção de uma lista de prioridades via heap

- Ajusta a prioridade do nó $(n-1)/2$ até o nó 0, nessa ordem, realizando **descidas quando necessário**.



Construção de uma lista de prioridades via heap

- A estrutura agora é uma heap!



0	1	2	3	4	5	6	7
95	66	78	60	28	70	39	33

Heap: construção a partir de uma lista

```
//Constroi heap máximo
void Constroi_heap (fp filapri) {
    int k;
    int meio = (filapri->n-1)/2;
    for (k = meio; k >= 0; k--)
        Desce_no_heap (filapri, k);
}
```

Tempo da construção: O(n)

Outras utilidades do heap: algoritmo heapsort

- A partir de um heap, é possível realizar ordenação de um conjunto de dados através do algoritmo abaixo:
  - Elemento de maior prioridade (raiz) é permutado com o último elemento do array.
  - Esse elemento então já estará fixado na posição correta (ordenação crescente).
  - Considerar que o vetor tenha tamanho n-1 e ir descendo a raiz para que o heap fique consistente.
  - Repetir esses passos n-1 vezes.

Tempo do heap sort: O (n log n)!

Heap: construção a partir de uma lista

```
//Heapsort
void Heapsort (fp filapri) {
     int k;
     for (k = (filapri->n-1)/2; k >= 0; k--)
          Desce_no_heap (filapri, k);
     k = filapri->n;
     while (k > 1) //Extrai o máximo
     {
          Permuta (&filapri->A[0], &filapri->A[k-1]);
          k--;
          Desce_no_heap (filapri, 0);
     }
}
```


BUBBLE SORT
HEAP SORT
BOGO SORT
ARRAYS.SORT()
imgflip.com