DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Ordenação: Heapsort

Professores: Luiz Chaimowicz e Raquel Prates

Heapsort

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- Algoritmo:
  1. Selecione o menor (maior) item do vetor.
  2. Troque-o com o item da primeira (última) posição do vetor.
  3. Repita estas operações com os n – 1 itens restantes, depois com os n – 2 itens, e assim sucessivamente.
- No seleção, o custo para encontrar o menor (ou o maior) item entre n itens é n-1 comparações.
- Isso pode ser reduzido utilizando uma fila de prioridades *(heap).*

Fila de Prioridades

- Aplicações:
  - Atendimento: prioridade de idosos, triagem
  - SO: alocação de processos, fila de impressão
  - Etc....

- Definição:
  - Estrutura de dados composta de itens, cuja chave reflete a prioridade com que se deve tratar aquele item.
  - Suporta duas operações principais: inserção de um novo item e remoção do item com a maior chave.

Fila de Prioridades

- Operações
  - Constrói a fila de prioridade com N itens
  - Insere um novo item
  - Retira o maior item
  - Altera a prioridade de um item
  - ...

Fila de Prioridades

- Representações
  - Lista sequencial ordenada
  - Lista sequencial não ordenada
  - **Heap**

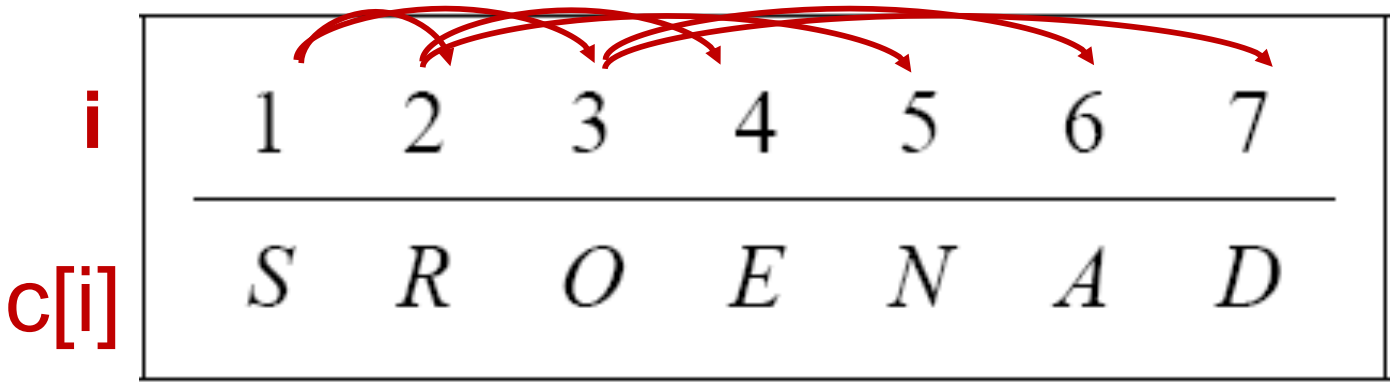
	Constrói	Insere	Retira máximo	Altera prioridade
Lista ordenada	O(N log N)	O(N)	O(1)	O(N)
Lista não ordenada	O(N)	O(1)	O(N)	O(1)
Heaps	O(N log N)	O(log N)	O(log N)	O(log N)

O QUE É UM HEAP?

Heap

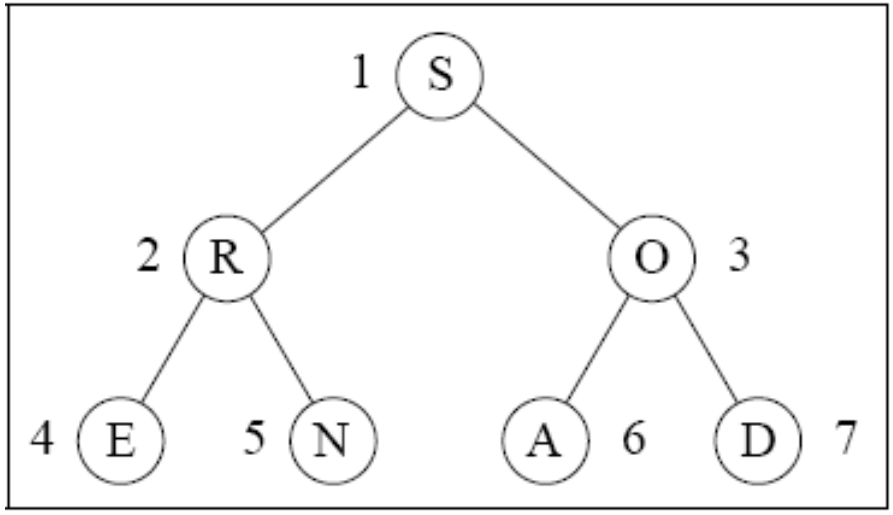
- É uma sequência de itens com chaves c[1], c[2], ... , c[n], tal que, para todo i = 1, 2, ..., n/2 :

$$c[i] >= c[2i], \quad c[i] >= c[2i + 1],$$

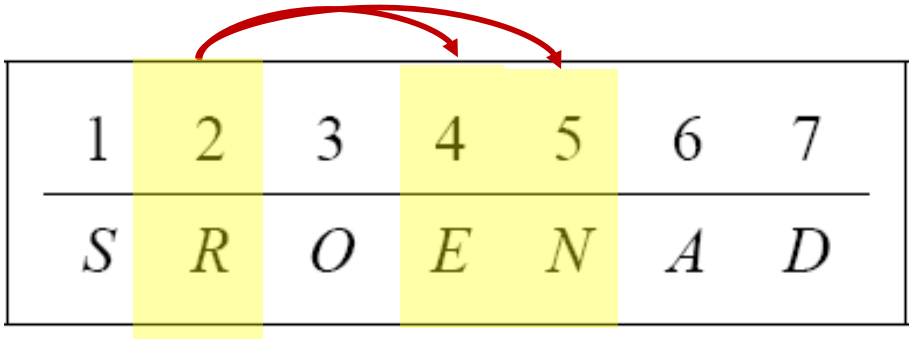


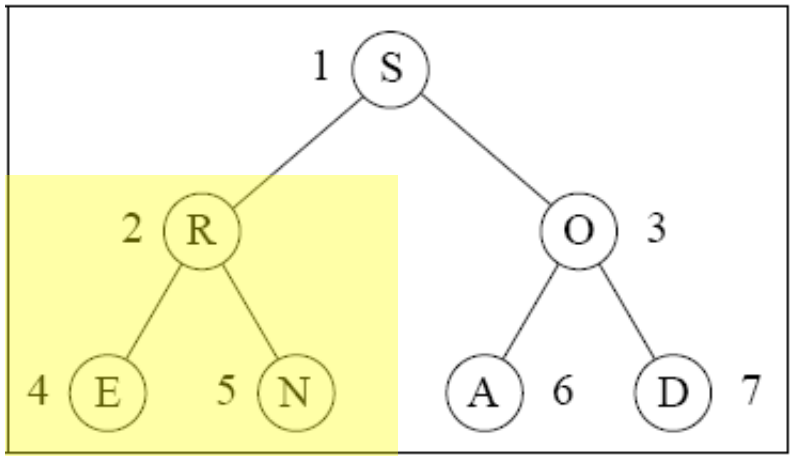
Heap

- Essa definição pode ser facilmente visualizada em uma **árvore binária completa**:
  - Será um heap se cada nó for maior ou igual seus filhos.
  - Com isso, a maior chave estará na raiz



Heap



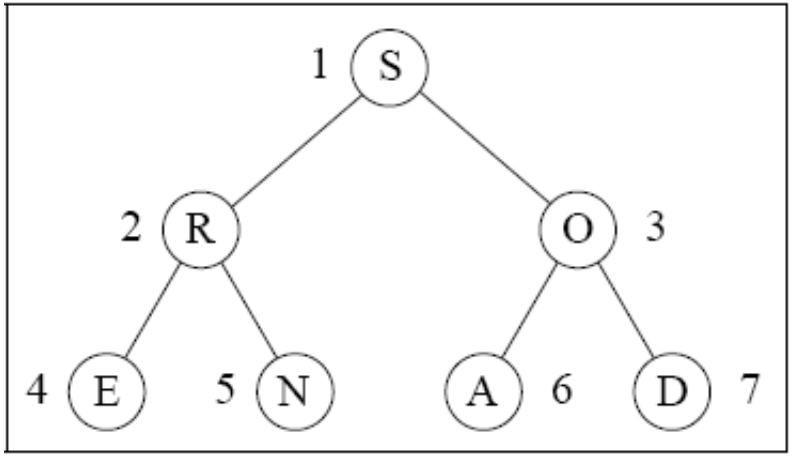


- Representação vetorial para de árvore
  - Nós são numerados de 1 a n
  - O primeiro é chamado raiz
  - Os nós 2k e 2k+1 são filhos da esquerda e direita do nó k, para $1 \leq k \leq n/2$.
  - O nó k/2 é o pai do nó k, $1 < k \leq n$

Heap

- Representação por meio de vetores é compacta
- Permite caminhar pelos nós da árvore facilmente
  - Filhos de um nó i estão nas posições $2i$ e $2i + 1$
  - O pai de um nó i está na posição $i/2$
  - A maior chave sempre está na posição 1

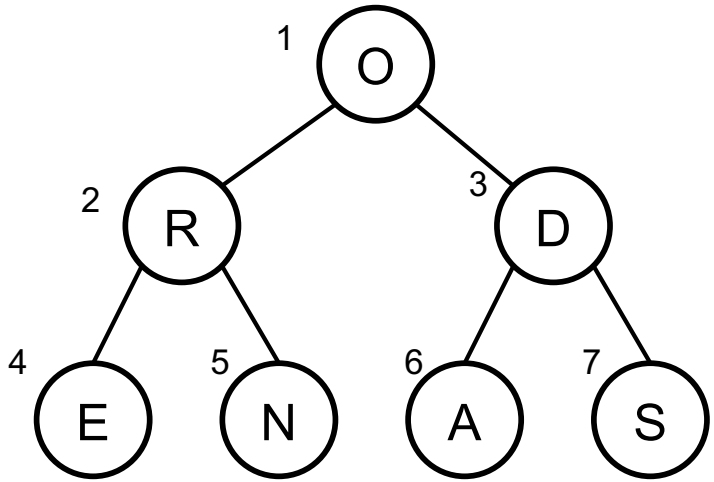
1	2	3	4	5	6	7
S	*R*	*O*	*E*	*N*	*A*	*D*



Construção do Heap

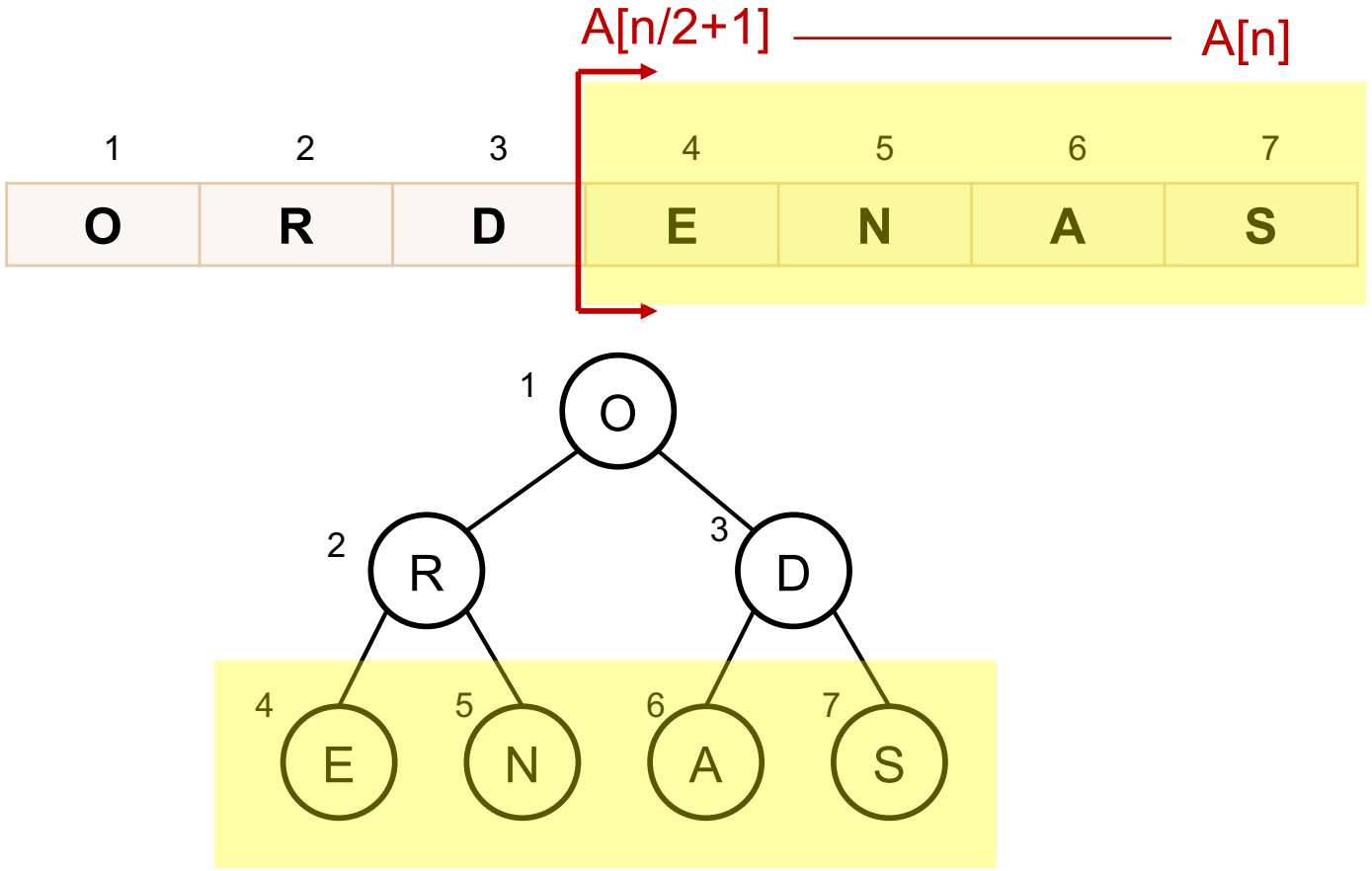
- Condição para ser heap: Nó pai (c[i]) deve ser maior que seus filhos (c[2*i] e c[2*i+1])

1	2	3	4	5	6	7
O	**R**	**D**	**E**	**N**	**A**	**S**



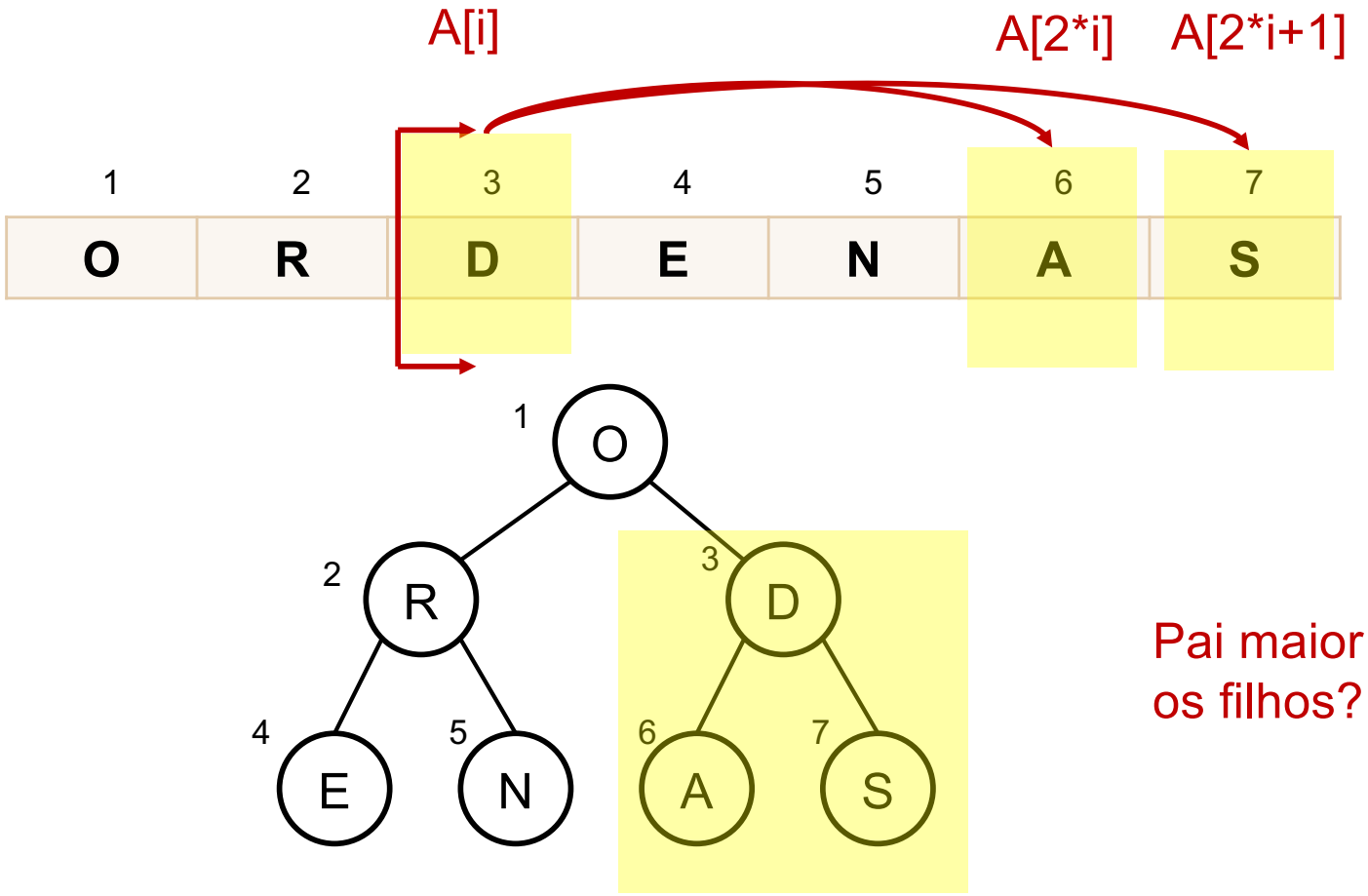
Construção do Heap

- As folhas da árvore, nunca violam a condição do heap (não tem filhos)



Construção do Heap

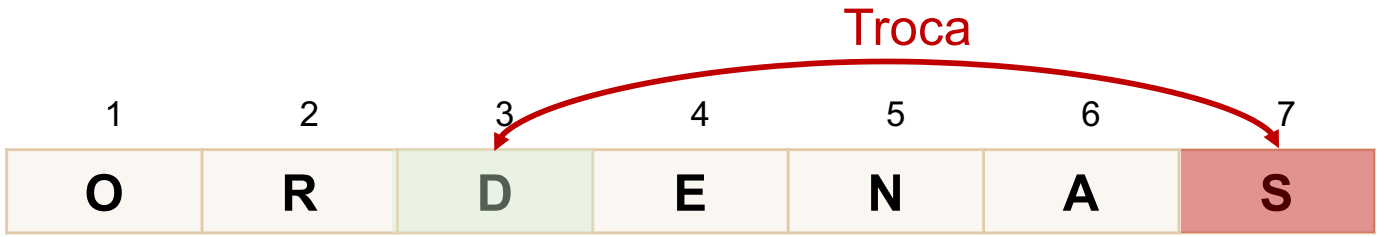
- Considerando um pai por vez vamos:
  1) Verificar se a condição de heap está mantida

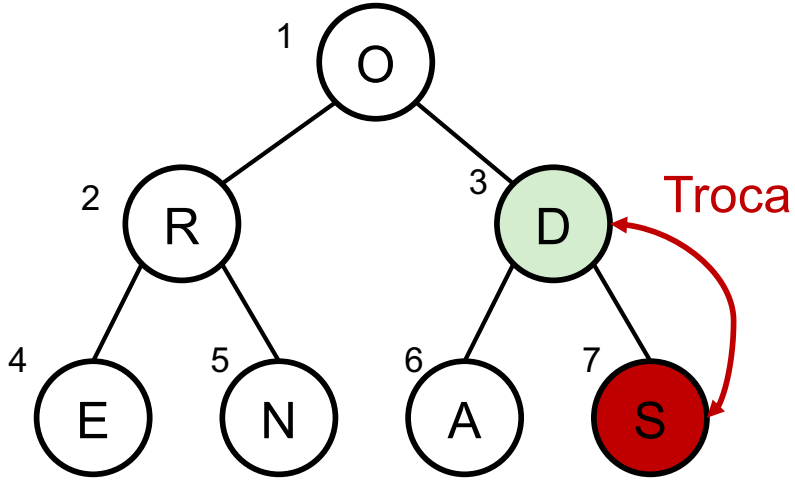


Pai maior que os filhos?

Construção do Heap

- Considerando um pai por vez vamos:
  1) Condição violada



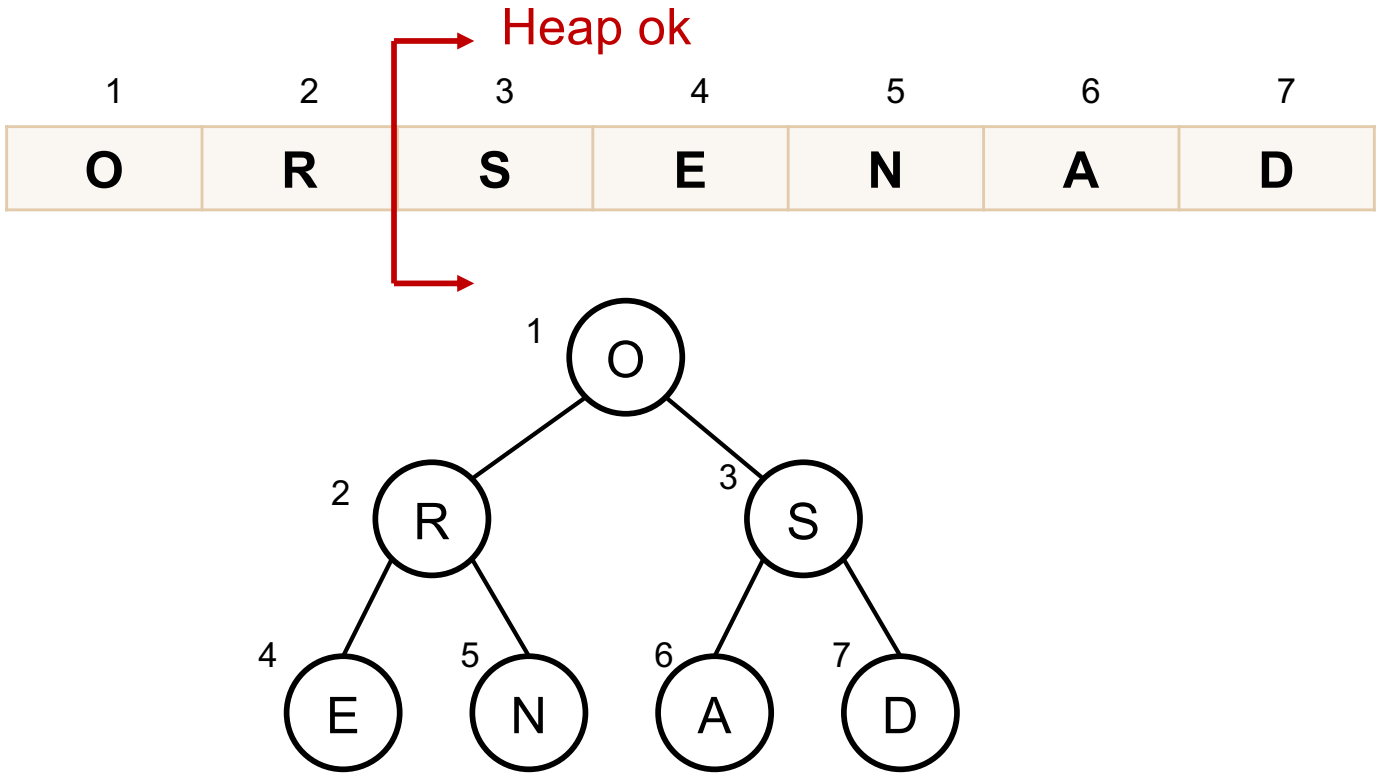


Pai maior que os filhos?

Não!

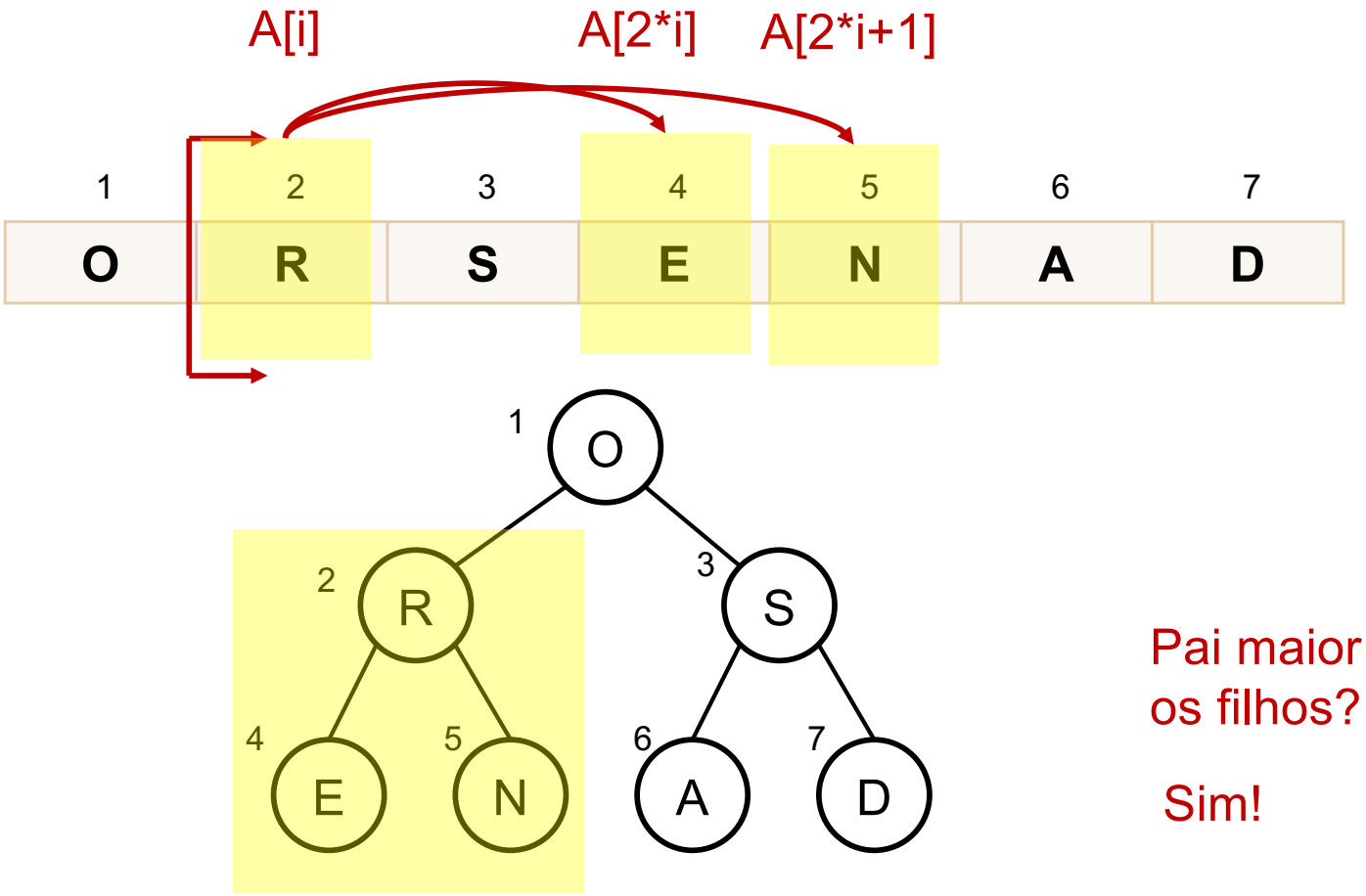
Construção do Heap

- Considerando um pai por vez vamos:
  1) Refaz condição do heap



Construção do Heap

- Considerando um pai por vez vamos:
  1) Verificar se a condição de heap está mantida

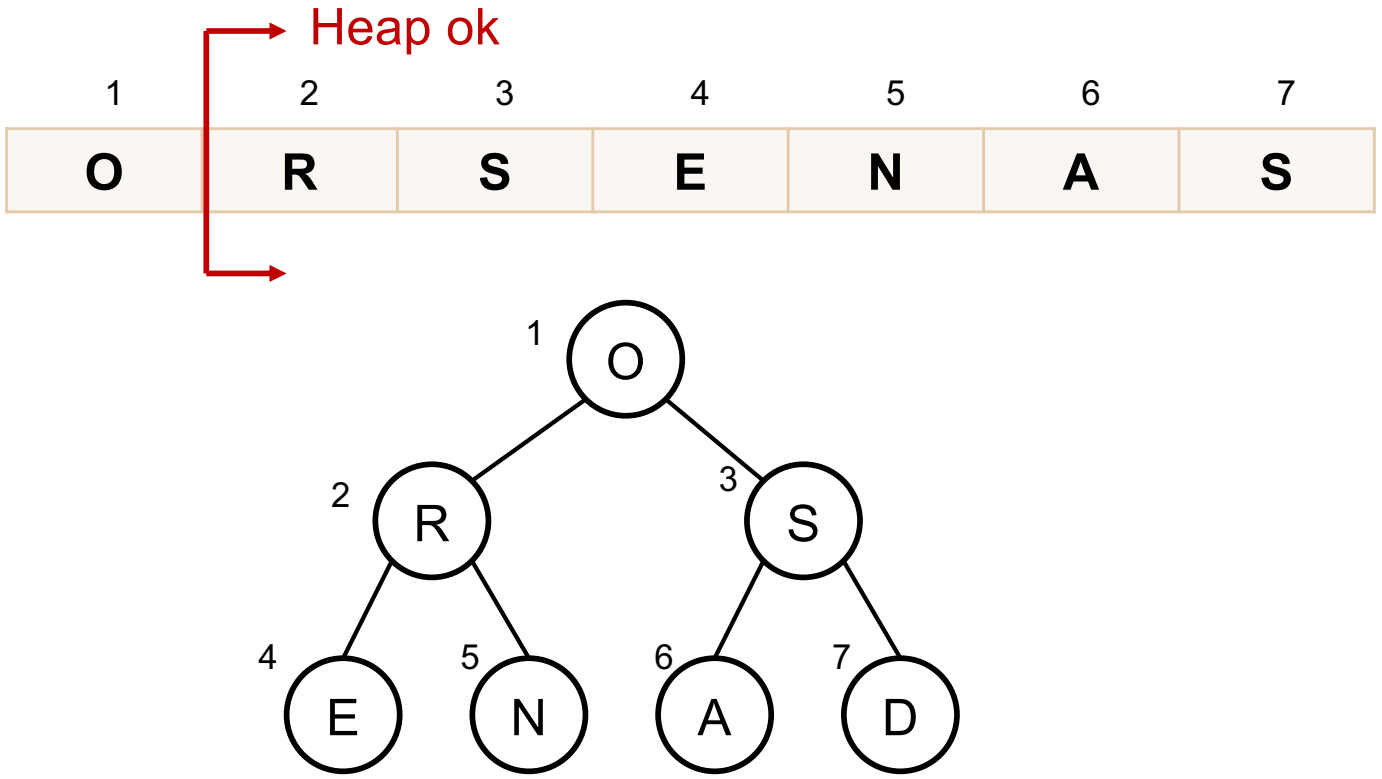


Pai maior que os filhos?

Sim!

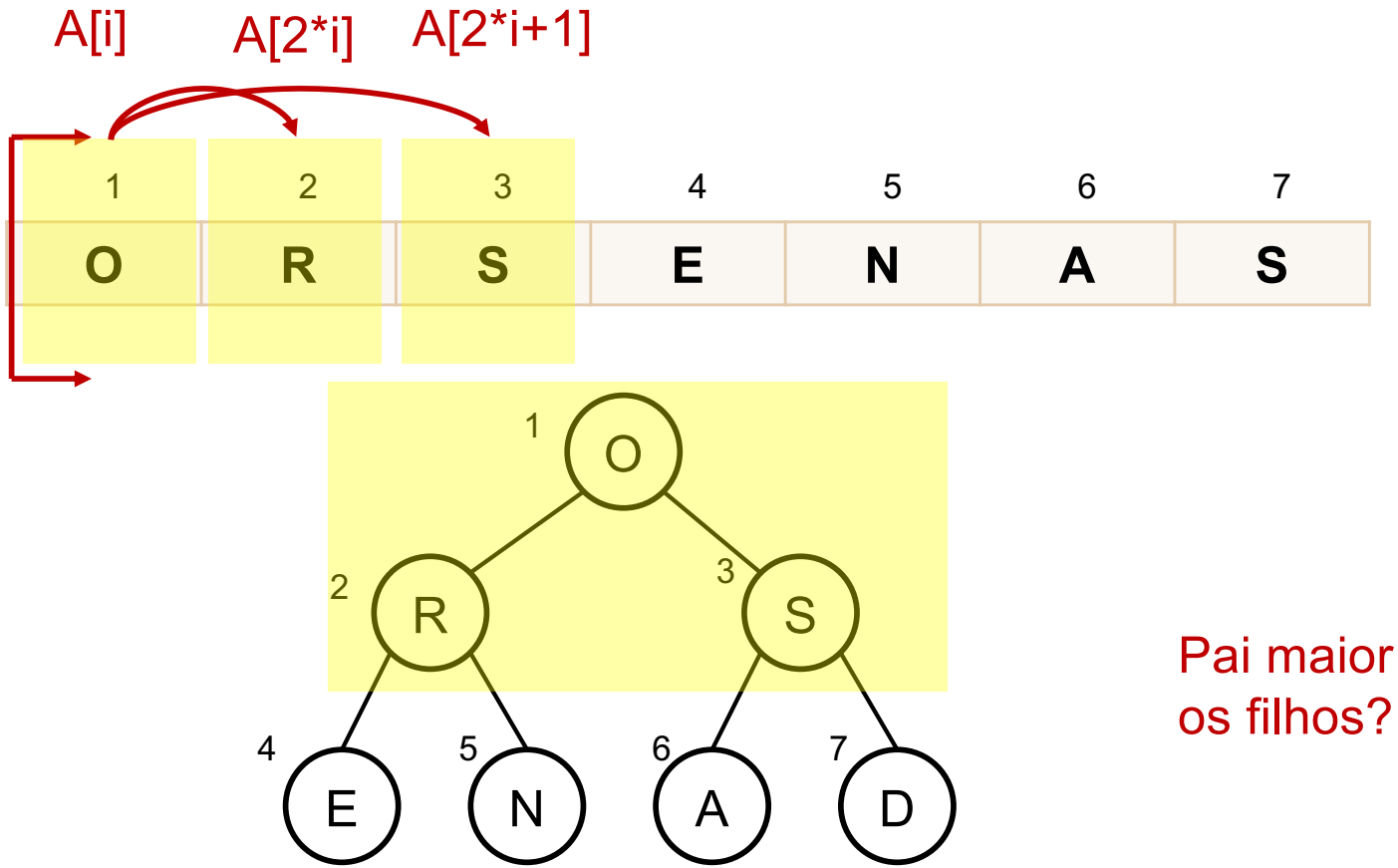
Construção do Heap

- Considerando um pai por vez vamos:
  1) Refaz condição do heap



Construção do Heap

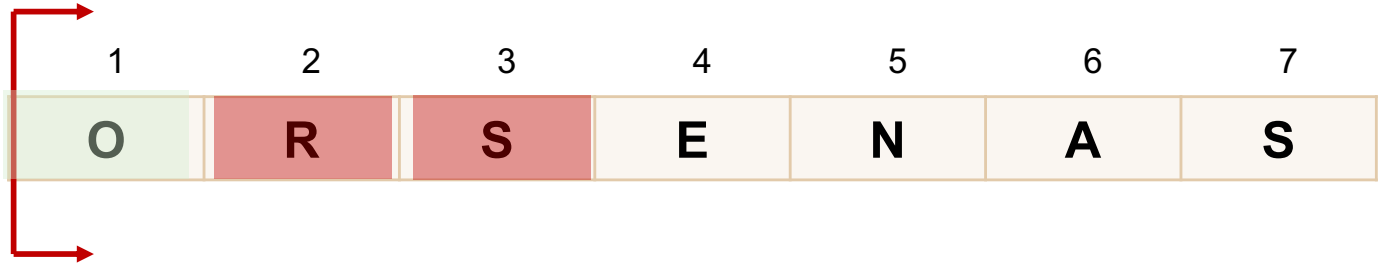
- Considerando um pai por vez vamos:
  1) Verificar se a condição de heap está mantida

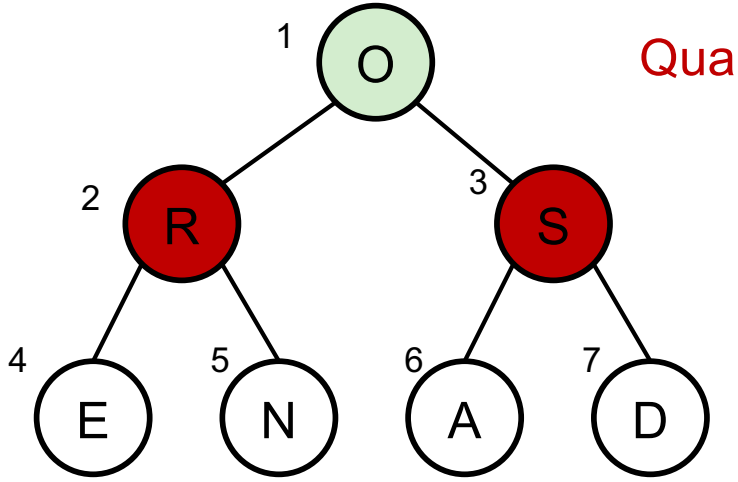


Pai maior que os filhos?

Construção do Heap

- Considerando um pai por vez vamos:
  1) Condição violada





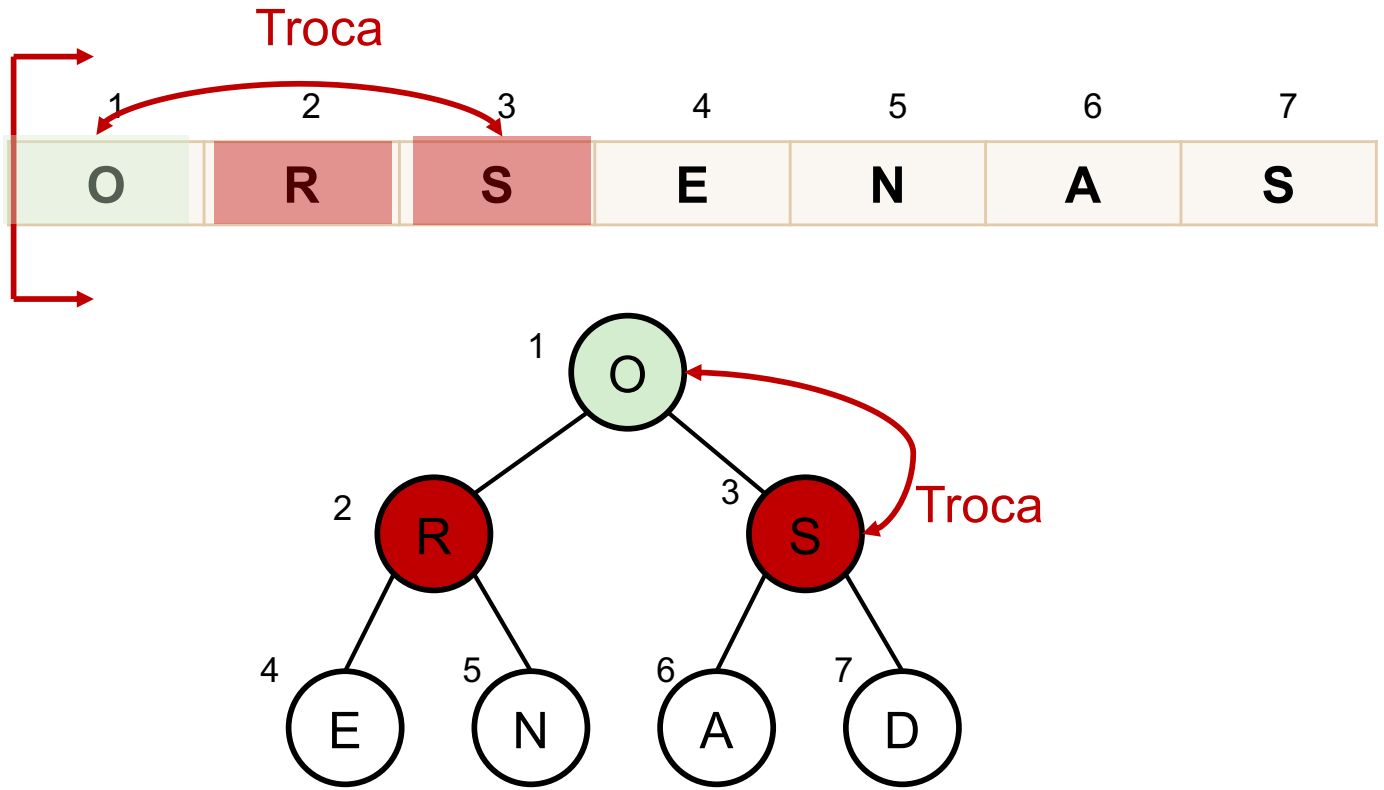
Qual filho trocar?

Pai maior que os filhos?

Não!

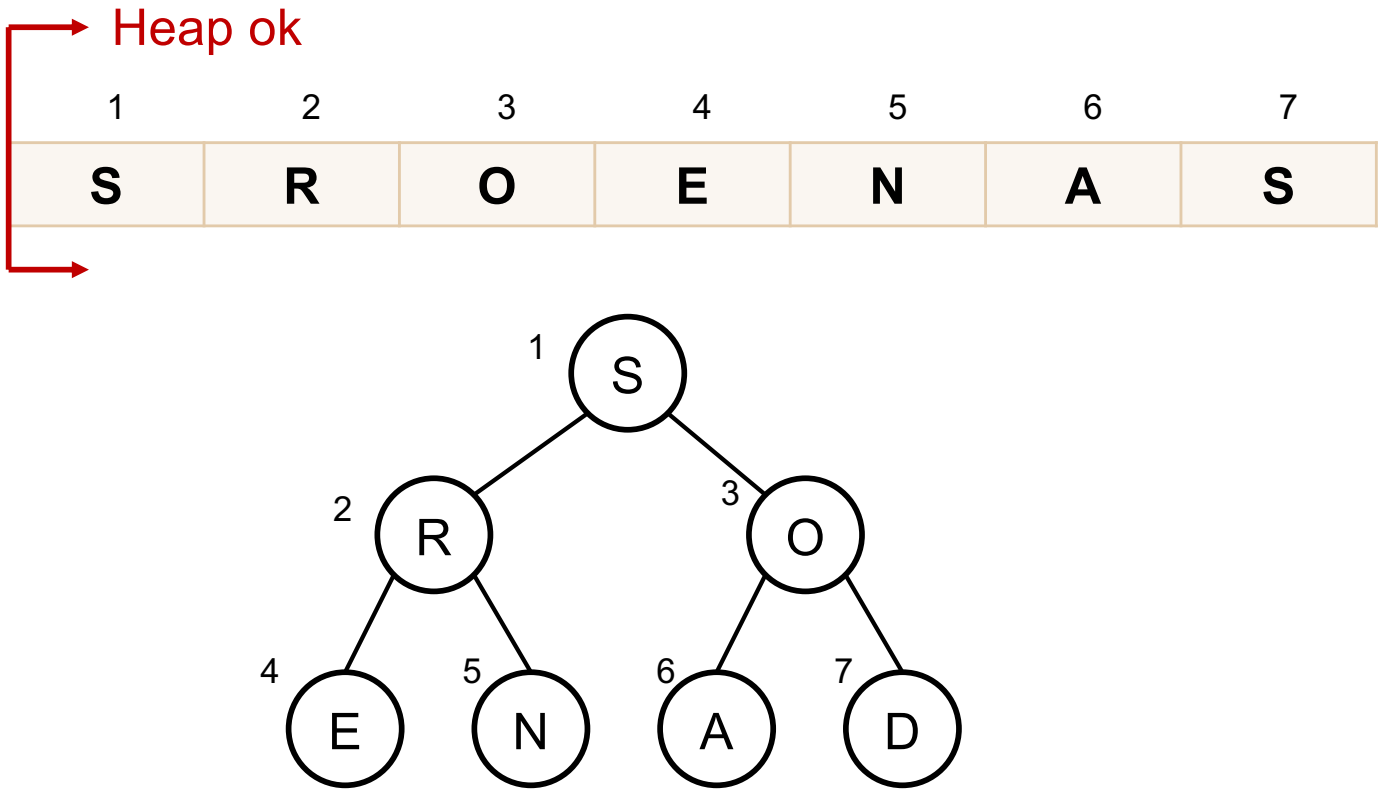
Construção do Heap

- Considerando um pai por vez vamos:
  1) Refaz condição do heap



Construção do Heap

- Considerando um pai por vez vamos:
  1) Refaz condição do heap



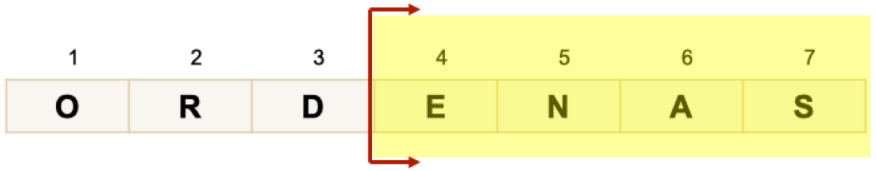
HEAP - CÓDIGO

Heap – Construção do heap

```
void Constroi(Item *A, int n) {
  int Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```

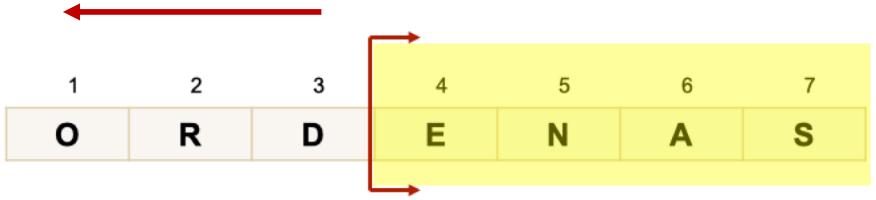
Heap – Construção do heap

```
void Constroi(Item *A, int n) {
  int Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```



Heap – Construção do heap

```
void Constroi(Item *A, int n) {
   int Esq;

   Esq = n / 2 + 1;
   while (Esq > 1) {
      Esq--;
      Refaz(Esq, n, A);
   }
}
```



Refaz a condição do heap

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
       if (A[j].Chave < A[j+1].Chave) j++;
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```

Refaz a condição do heap

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
       if (A[j].Chave < A[j+1].Chave) j++;
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```

Vetor

Último elemento do vetor

Recebe o elemento a partir do qual vai considerar a condição do heap (raiz da subárvore)

Refaz a condição do heap

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
       if (A[j].Chave < A[j+1].Chave) j++;
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```

Nó pai, sendo considerado

Filho da esquerda

Chave do nó pai

Enquanto j ainda representa um filho

Se existe um filho da direita

Verifica o maior entre os filhos, para ser comparado com o pai

Refaz a condição do heap

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
       if (A[j].Chave < A[j+1].Chave) j++;
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```

Se o pai for maior que os filhos, termina o loop

Chave do filho passa para o pai

Atualiza índice do pai e filhos, para percorrer a subárvore

Coloca a chave do pai, na posição do descendente

Refaz

Trecho do código

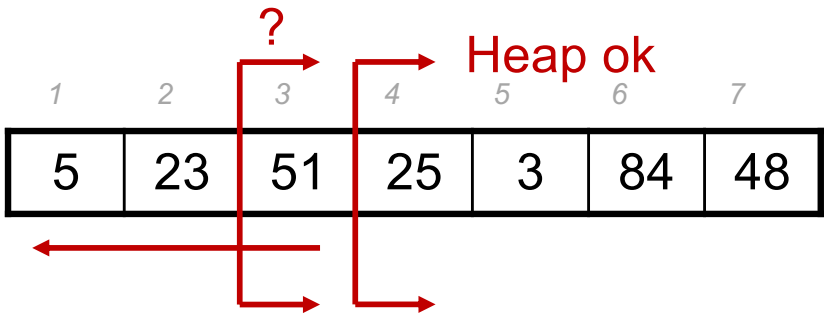
1	2	3	4	5	6	7
5	23	51	25	3	84	48

RAM

Construindo o Heap

Trecho do código

```
void Constroi(Item *A, int n)
{
  int Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```



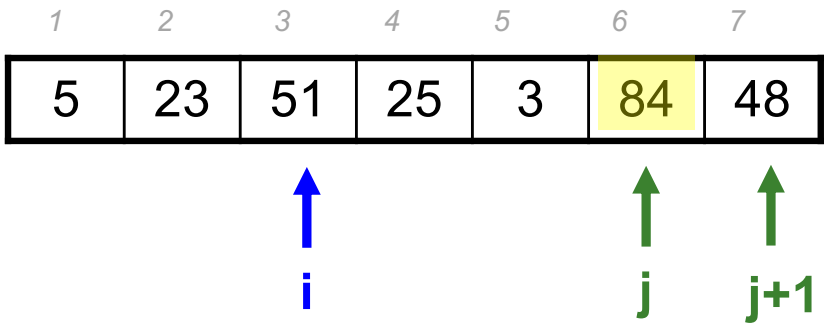
RAM

```
n = 7
Esq = 43
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
      if (A[j].Chave < A[j+1].Chave) j++;
      if (x.Chave >= A[j].Chave) break;
      A[i] = A[j];
      i = j;
      j = i *2;
    }
    A[i] = x;
}
```



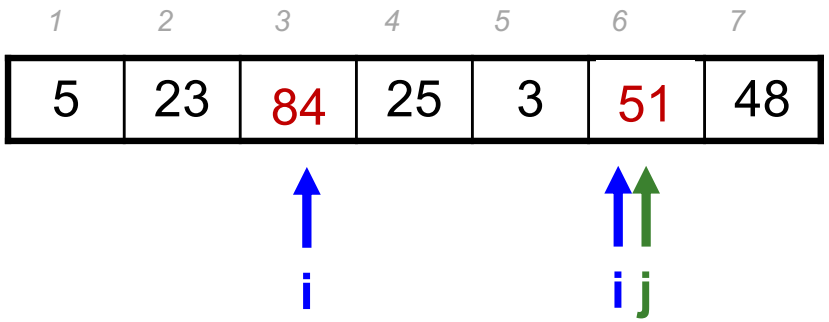
RAM

```
Esq = 3              i = 3
Dir = 7              j = 6
                     x = [51]
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
 int i, j;
 Item x;
 i = Esq;
 j = i * 2;
 x = A[i];
 while (j <= Dir){
    if (j < Dir)
     if (A[j].Chave < A[j+1].Chave) j++;
     if (x.Chave >= A[j].Chave) break;
     A[i] = A[j];
     i = j;
     j = i *2;
   }
   A[i] = x;
}
```



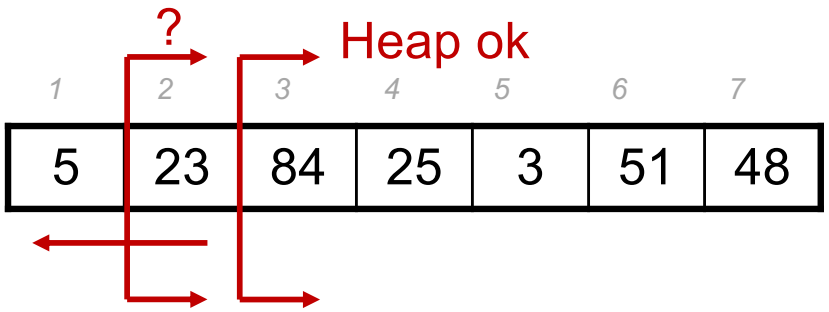
RAM

```
Esq = 3                 i = 3 6
Dir = 7                 j = 6 12
                        x = [51]
```

Construindo o Heap

Trecho do código

```
void Constroi(Item *A, int n)
{
  int Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```



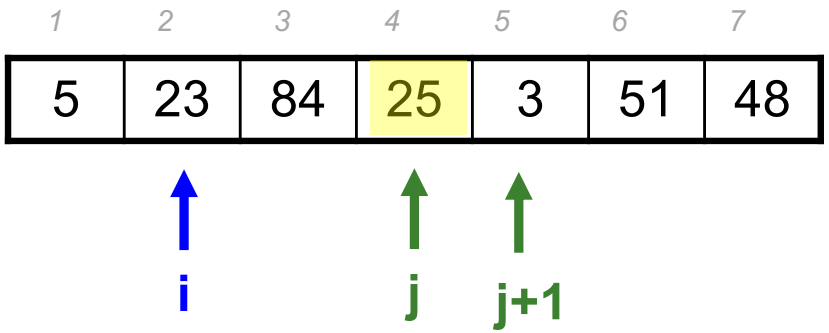
RAM

```
n = 7
Esq =  3 2
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
      if (A[j].Chave < A[j+1].Chave) j++;
      if (x.Chave >= A[j].Chave) break;
      A[i] = A[j];
      i = j;
      j = i *2;
    }
    A[i] = x;
}
```



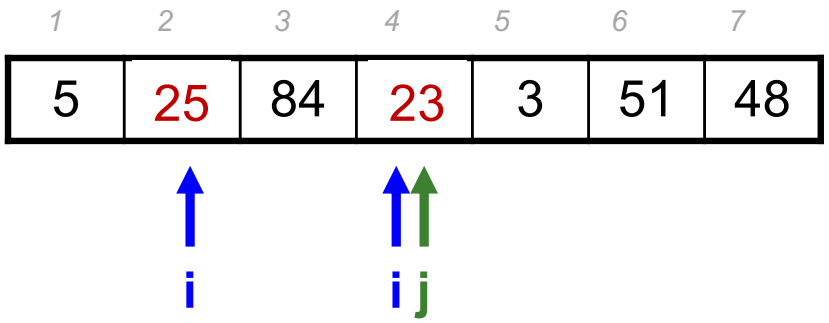
RAM

```
Esq = 2              i = 2
Dir = 7              j = 4
                     x = [23]
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
 int i, j;
 Item x;
 i = Esq;
 j = i * 2;
 x = A[i];
 while (j <= Dir){
    if (j < Dir)
     if (A[j].Chave < A[j+1].Chave) j++;
     if (x.Chave >= A[j].Chave) break;
     A[i] = A[j];
     i = j;
     j = i *2;
   }
   A[i] = x;
}
```



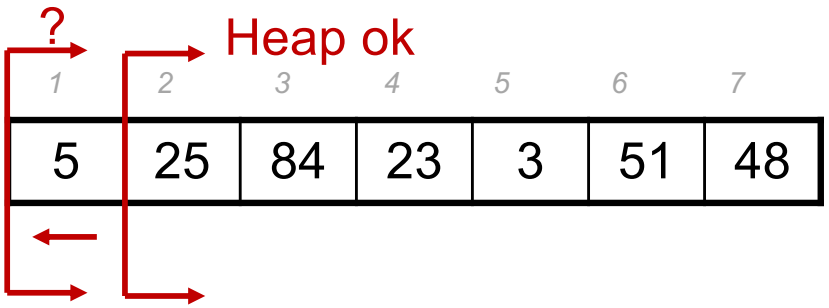
RAM

```
Esq = 2                 i = 2 4
Dir = 7                 j = 4 8
                        x = [23]
```

Construindo o Heap

Trecho do código

```
void Constroi(Item *A, int n)
{
  int Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```



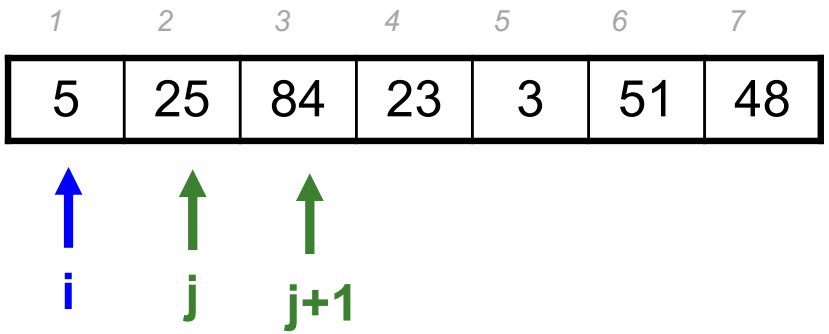
RAM

```
n = 7
Esq = 2 1
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
  int i, j;
  Item x;
  i = Esq;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir)
      if (A[j].Chave < A[j+1].Chave) j++;
      if (x.Chave >= A[j].Chave) break;
      A[i] = A[j];
      i = j;
      j = i *2;
    }
    A[i] = x;
}
```



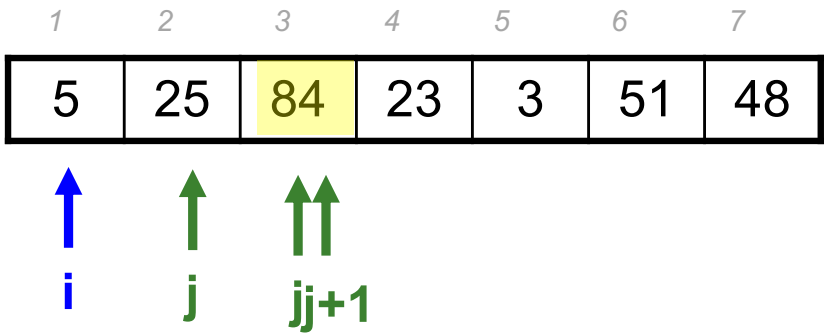
RAM

```
Esq = 1                 i = 1
Dir = 7                 j = 2
                        x = [5]
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
 int i, j;
 Item x;
 i = Esq;
 j = i * 2;
 x = A[i];
 while (j <= Dir){
   if (j < Dir)
    if (A[j].Chave < A[j+1].Chave) j++;
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```



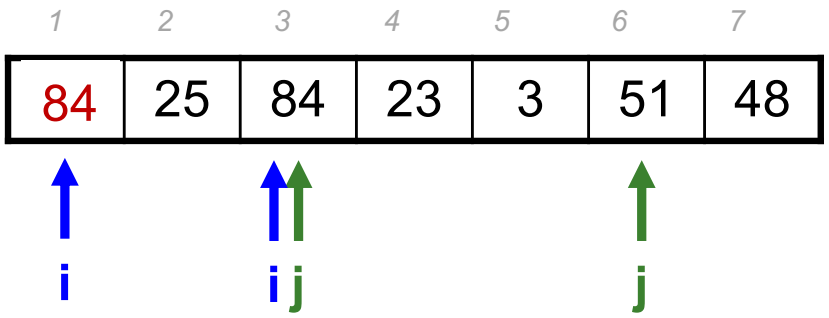
RAM

```
Esq = 1              i = 1
Dir = 7              j = 2 3
                     x = [5]
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
 int i, j;
 Item x;
 i = Esq;
 j = i * 2;
 x = A[i];
 while (j <= Dir){
    if (j < Dir)
     if (A[j].Chave < A[j+1].Chave) j++;
     if (x.Chave >= A[j].Chave) break;
     A[i] = A[j];
     i = j;
     j = i *2;
   }
   A[i] = x;
}
```



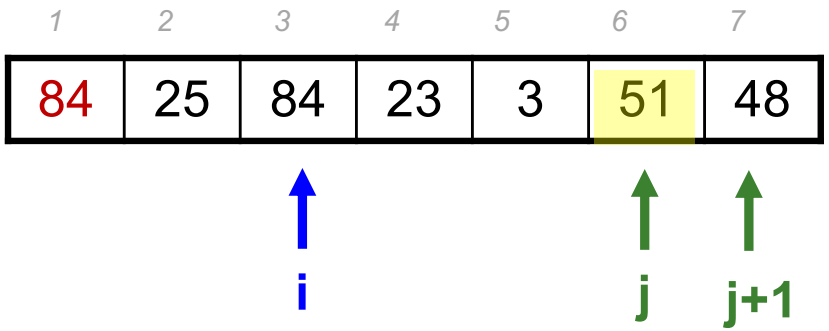
RAM

```
Esq = 1                 i = 1 3
Dir = 7                 j = 3 6
                        x = [5]
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
 int i, j;
 Item x;
 i = Esq;
 j = i * 2;
 x = A[i];
 while (j <= Dir){
   if (j < Dir)
    if (A[j].Chave < A[j+1].Chave) j++;
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```



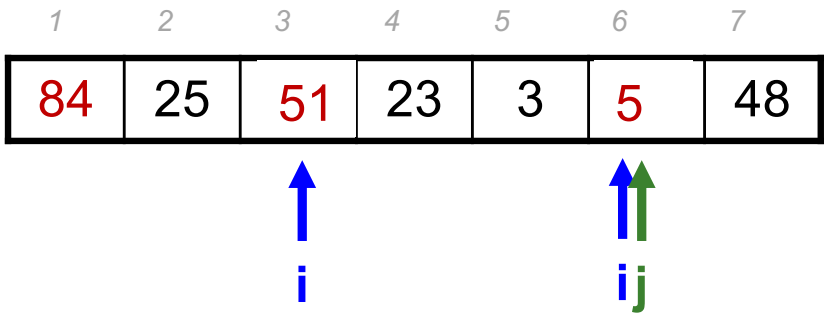
RAM

```
Esq = 1              i = 3
Dir = 7              j = 6
                     x = [5]
```

Refaz

Trecho do código

```
void  Refaz(int Esq, int Dir, Item *A){
 int i, j;
 Item x;
 i = Esq;
 j = i * 2;
 x = A[i];
 while (j <= Dir){
    if (j < Dir)
     if (A[j].Chave < A[j+1].Chave) j++;
     if (x.Chave >= A[j].Chave) break;
     A[i] = A[j];
     i = j;
     j = i *2;
   }
   A[i] = x;
}
```



RAM

```
Esq = 1                 i = 3 6
Dir = 7                 j = 6 12
                        x = [5]
```

Construindo o Heap

Trecho do código

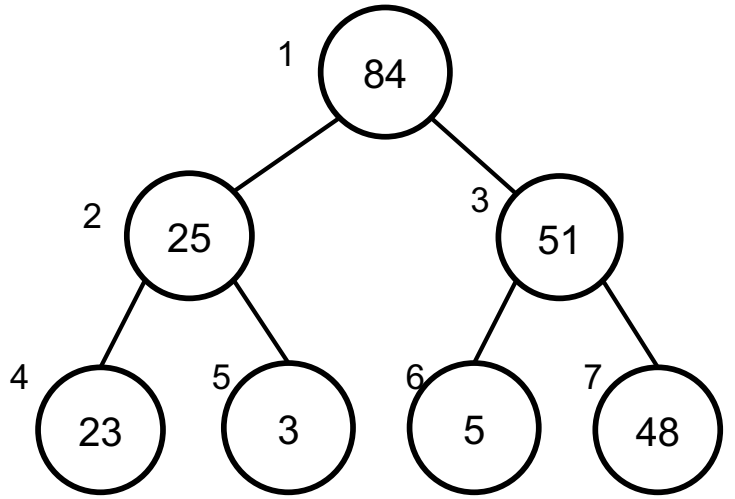
```
void Constroi(Item *A, int n)
{
  int Esq;

  Esq = n / 2 + 1;
  while (Esq > 1) {
    Esq--;
    Refaz(Esq, n, A);
  }
}
```

RAM

```
n = 7
Esq = 1
```

Heap ok

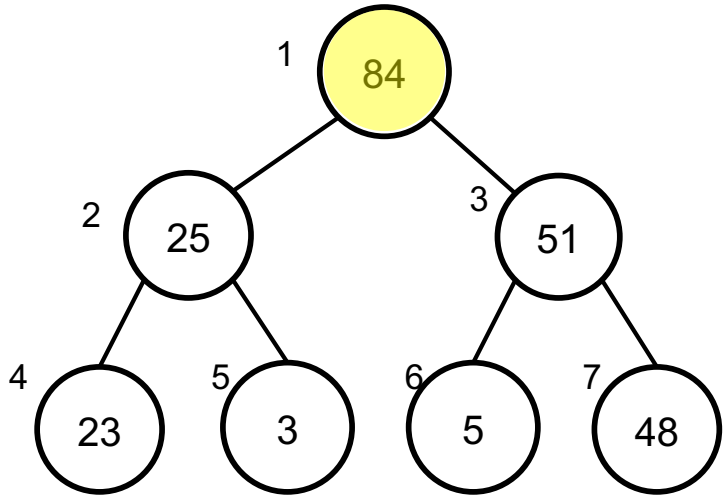
1	2	3	4	5	6	7
84	25	51	23	3	5	48



Fila de prioridades com Heap

- Elemento de maior prioridade?

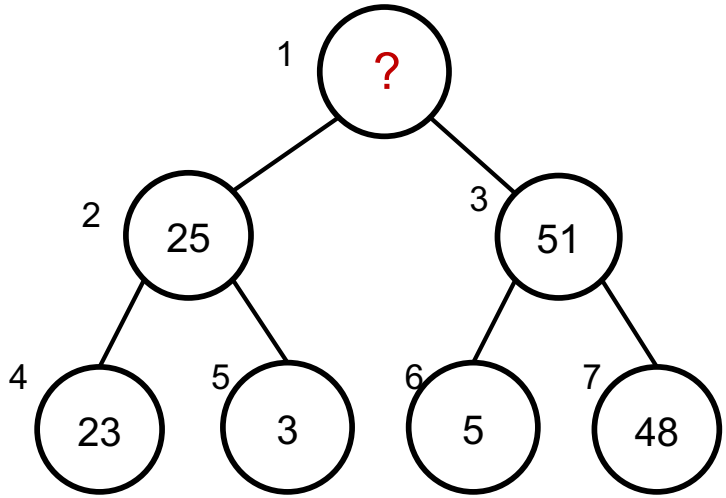
1	2	3	4	5	6	7
84	25	51	23	3	5	48



Fila de prioridades com Heap

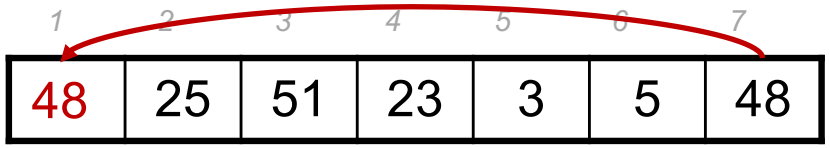
- Retira o de maior prioridade

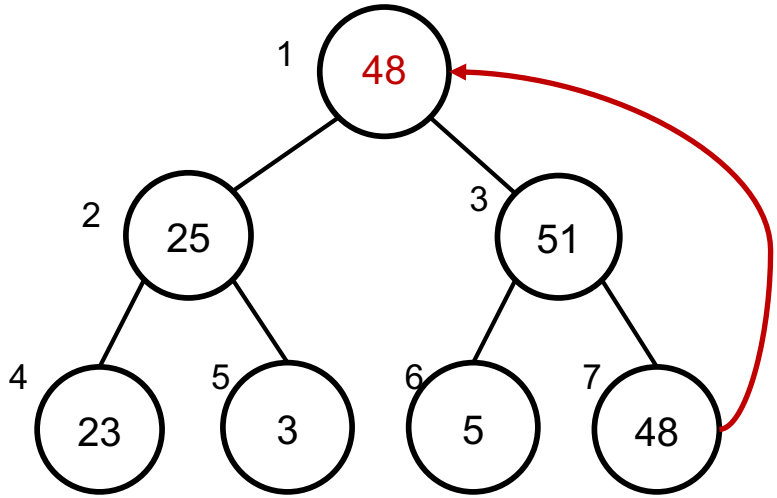
1	2	3	4	5	6	7
?	25	51	23	3	5	48



Fila de prioridades com Heap

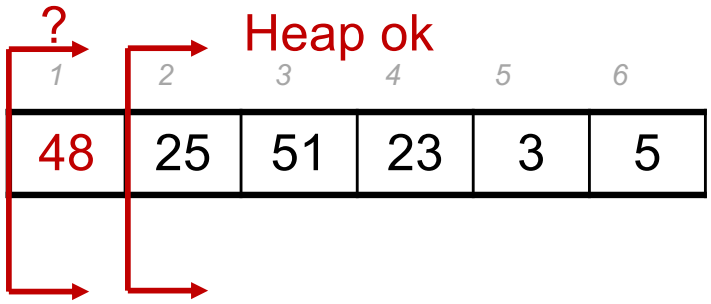
- Como reconstruir o heap?



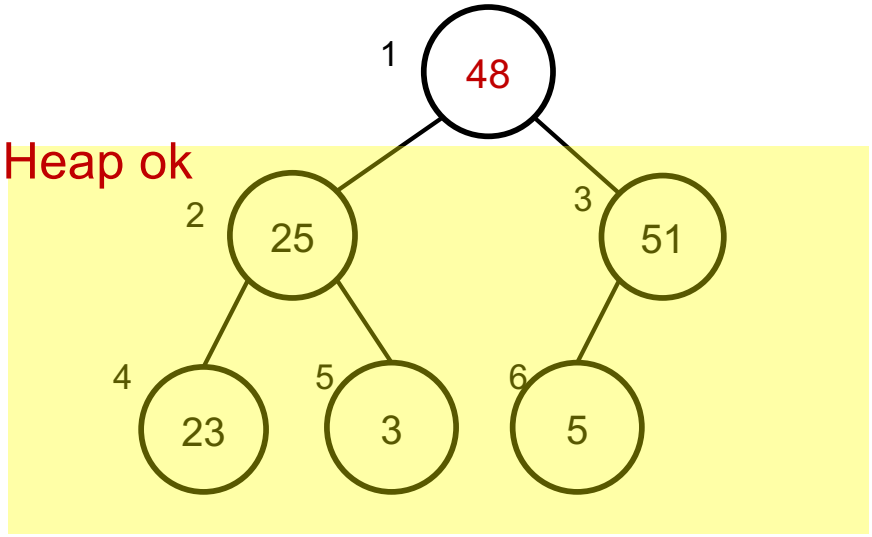


Fila de prioridades com Heap

- Como reconstruir o heap?

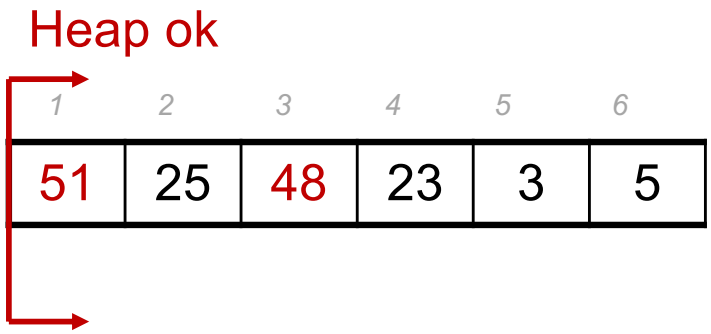


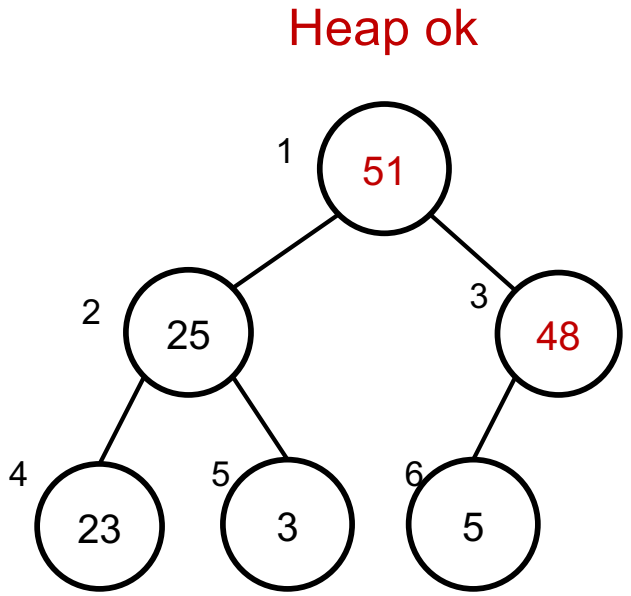
Refaz(1, 6, A)



Fila de prioridades com Heap

- Como reconstruir o heap?





Fila de prioridades com Heap

- Para obter o elemento com maior prioridade, remove ele e reconstrua o heap

```
Item RetiraMax(Item *A, int *n) {
  Item Maximo;
  if (*n < 1)
    printf("Erro: heap vazio\n");
  else {
    Maximo = A[1];
    A[1] = A[*n];
    (*n)--;
    Refaz(1, *n, A);
  }
  return Maximo;
}
```

Testa se o heap está vazio

Máximo recebe o valor do elemento de maior prioridade

Passa o último elemento para a primeira posição

Fila de prioridades com Heap

- Para obter o elemento com maior prioridade, remove ele e reconstrua o heap

```
Item RetiraMax(Item *A, int *n) {
  Item Maximo;
  if (*n < 1)
    printf("Erro: heap vazio\n");
  else {
    Maximo = A[1];
    A[1] = A[*n];
    (*n)--;
    Refaz(1, *n, A);
  }
  return Maximo;
}
```

Decrementa o número de elementos

Chama Refaz (1x) para reconstruir o heap

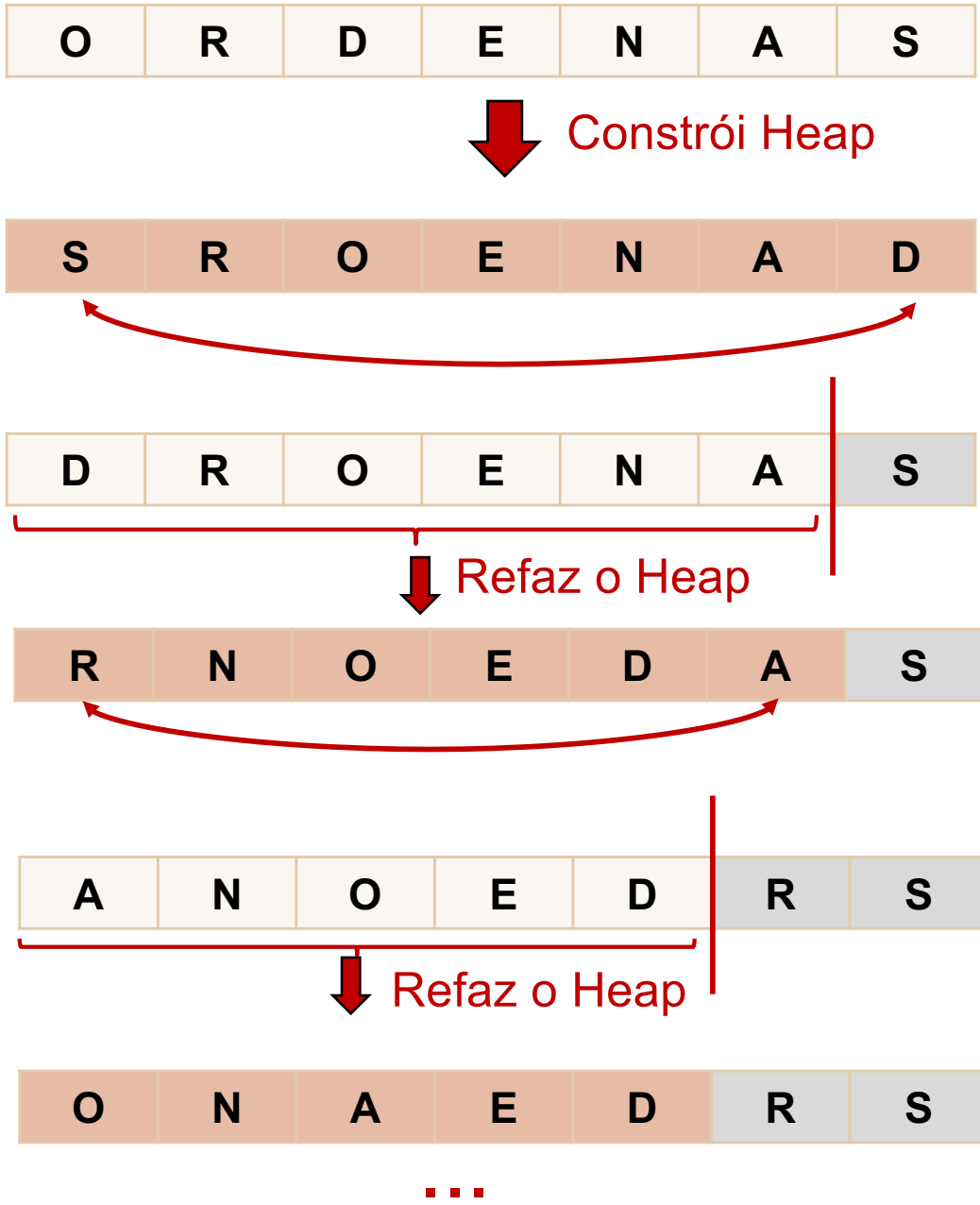
Retorna o elemento de maior prioridade

HEAPSORT – ORDENANDO COM O HEAP

Heapsort

- Algoritmo:
  1. Construir o *heap.*
  2. Troque o item na posição 1 do vetor (raiz do *heap) com o item da posição n.*
  3. Use o procedimento Refaz para reconstituir o *heap para os itens* A[1], A[2], ..., A[n - 1].
  4. Repita os passos 2 e 3 com os n - 1 itens restantes, depois com os n - 2, até que reste apenas um item.

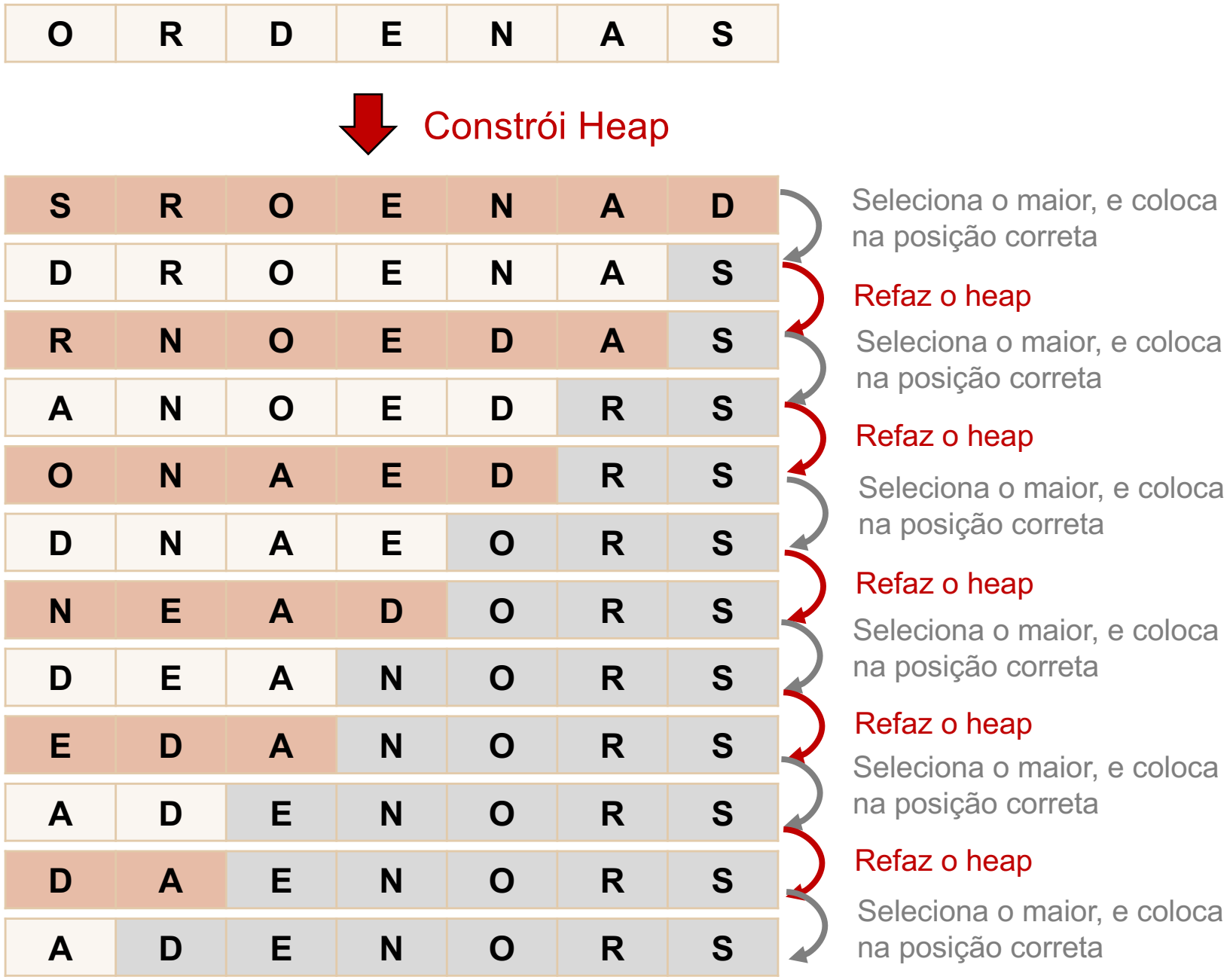
Heapsort - Exemplo



Seleciona o maior, e troca com o que está na última posição

Seleciona o 2º. maior, e troca com o que está na penúltima posição

Heapsort - Exemplo

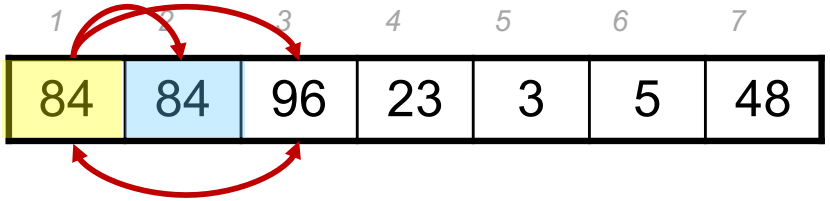


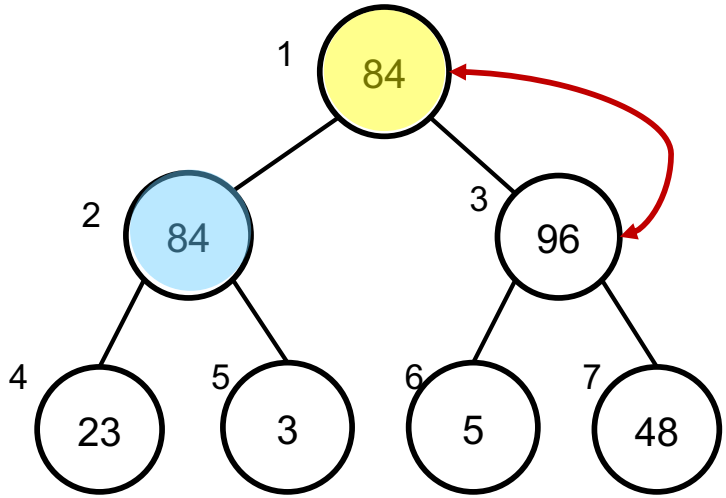
Heapsort

```
void Heapsort(Item *A, int *n) {
  int Esq, Dir;
  Item x;
  Constroi(A, n); /* constroi o heap */
  Esq = 1; Dir = *n;
  while (Dir > 1)
  { /* ordena o vetor */
    x = A[1];
    A[1] = A[Dir];
    A[Dir] = x;
    Dir--;
    Refaz(Esq, Dir, A);
  }
}
```

HEAPSORT - ANÁLISE

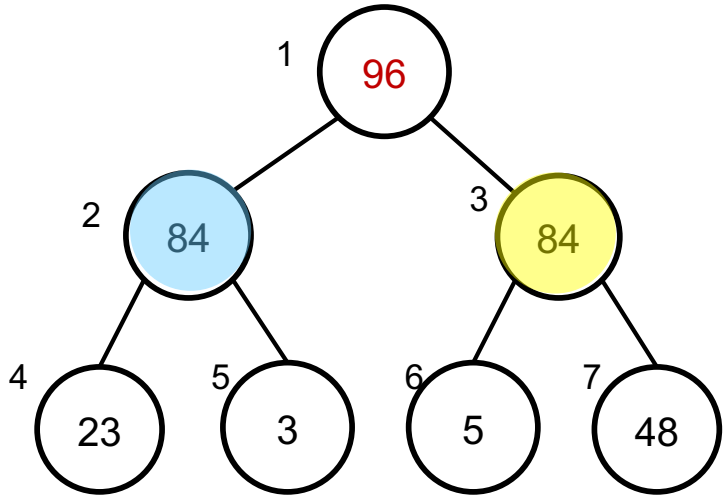
O método é estável?





O método é estável?

1	2	3	4	5	6	7
96	84	84	23	3	5	48



Inverteu a posição dos '84's

NÃO É ESTÁVEL!

Heapsort – Análise de Complexidade

- Refaz:
  - No pior caso, percorre todo um galho da árvore binária, ou seja, executa log n operações ➡ ***C(n) = O(log n)***
- Constrói:
  - Para os nós internos (n/2 elementos), chama refaz, logo executa: n/2 log n ➡ ***C(n) = O(n log n)***
- Heapsort
  - Chama Constroi – uma vez
  - Chama Refaz n-1 vezes

  } ➡ ***C(n) = O(n log n)***

Heapsort

- Vantagens:
  - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- Desvantagens:
  - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.
  - O Heapsort não é **estável**.
- Recomendado:
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap.*