

# Árvores



# Árvores Auto-ajustadas

- ◆ As árvores binárias de pesquisa são **projetadas para um acesso rápido à informação**.
- ◆ A velocidade das operações de inserção, remoção e busca realizadas sobre as árvores **é mais importante que sua forma**.
- ◆ O desempenho pode ser melhorado balanceando-se a árvore, mas esse não é o único método que pode ser usado.
- ◆ Nem todos os elementos são usados com **a mesma frequência**.
- ◆ Se um elemento é constantemente acessado, faz diferença se esse elemento está próximo a uma folha ou está próximo a raiz.
- ◆ A estratégia nas árvores auto-ajustadas **é reestruturar as árvores movendo para cima os elementos usados frequentemente**.
- ◆ **As árvores auto-ajustadas são árvores de prioridades**.

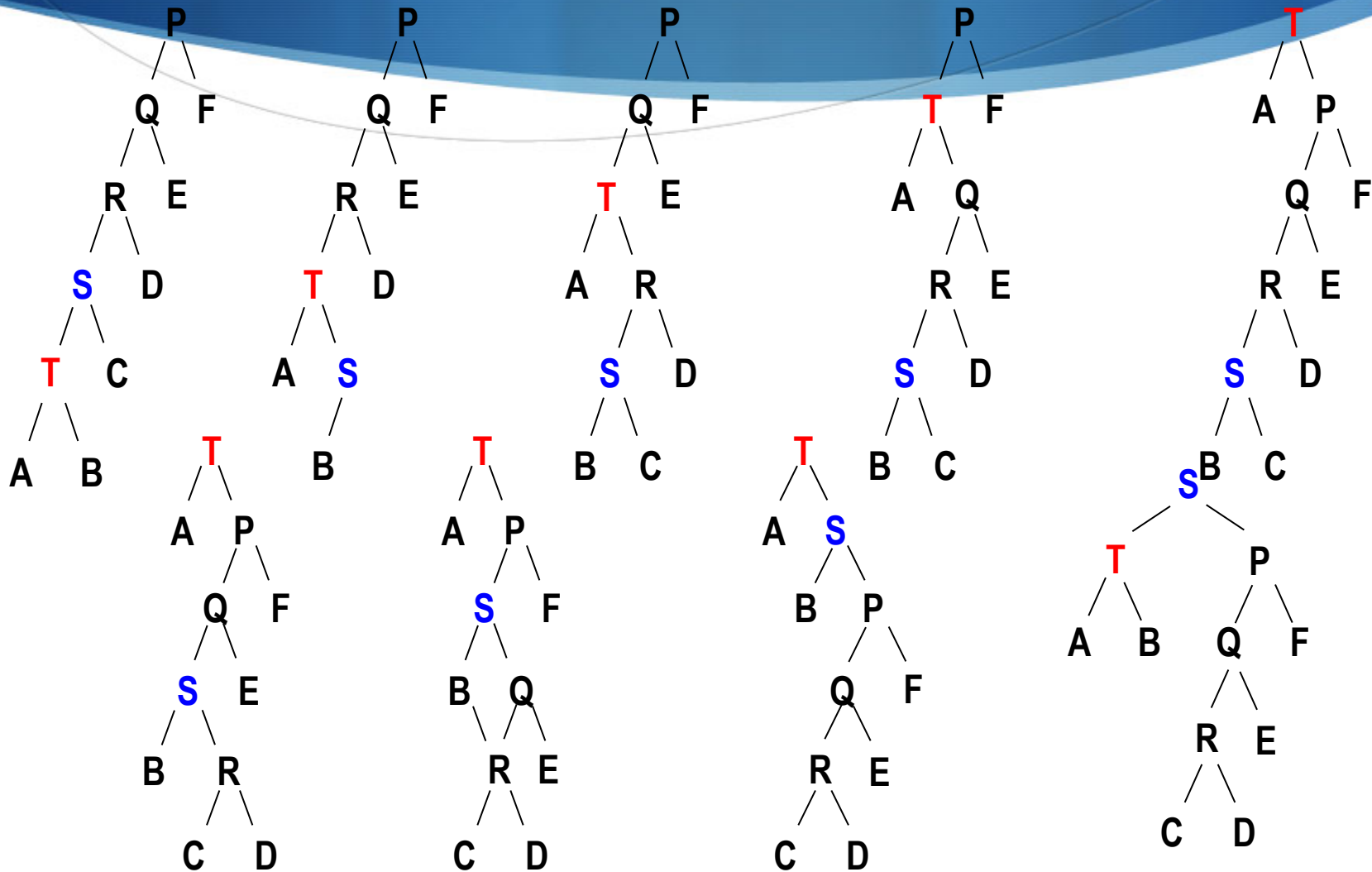
# Árvores Auto-ajustadas

- ◆ Nas **árvores de prioridades** a frequência de acesso aos nós pode ser determinada de diferentes modos:
  - ◆ Uma forma pode ser a **utilização de um campo contador**
    - ◆ que registra o número de vezes que o elemento foi usado para qualquer operação.
  - ◆ A árvore pode ser varrida para mover os elementos mais acessados em direção à raiz.
  - ◆ Outra abordagem mais simples assume que um elemento acessado **tem uma boa chance de ser acessado uma segunda vez**, assim ele é movido para cima na árvore.

# Árvores Auto-ajustadas

- ◆ A estratégia nas árvores auto-ajustadas é reestruturar as árvores **movendo para cima os elementos usados freqüentemente**.
- ◆ **Nenhuma** reestruturação é realizada para os novos elementos.
- ◆ Essa estratégia não funciona muito bem para árvores alongadas.
- ◆ Nesse caso a árvore melhora **vagarosamente**, no entanto o custo de mover um nó para a raiz se aproxima do custo de acessar o nó numa **árvore ótima por 3 vezes**.
- ◆ O esquema mostra a subida do elemento T para a raiz e em seguida do elemento S.

# Árvores Auto-ajustadas

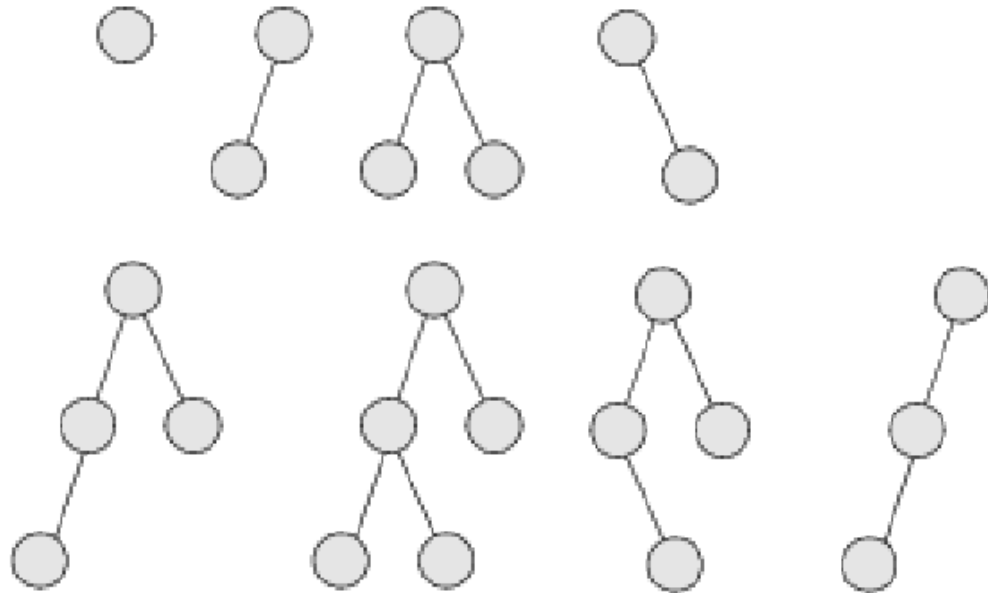


# Heaps

- ◆ Um tipo particular de árvores binárias são as **Heaps**.
- ◆ No caso de heaps a propriedade a ser mantida não se resume ao balanceamento da árvore, mas também às seguintes definições:
  - ◆ ser uma árvore completa
  - ◆ **manter uma propriedade de ordem**

# Heaps

- Uma **árvore completa** é definida como sendo:
  - uma árvore cheia,
  - elementos faltantes aparecem apenas no nível mais baixo e, sempre, na subárvore da direita.



Árvores completas

Árvores incompletas

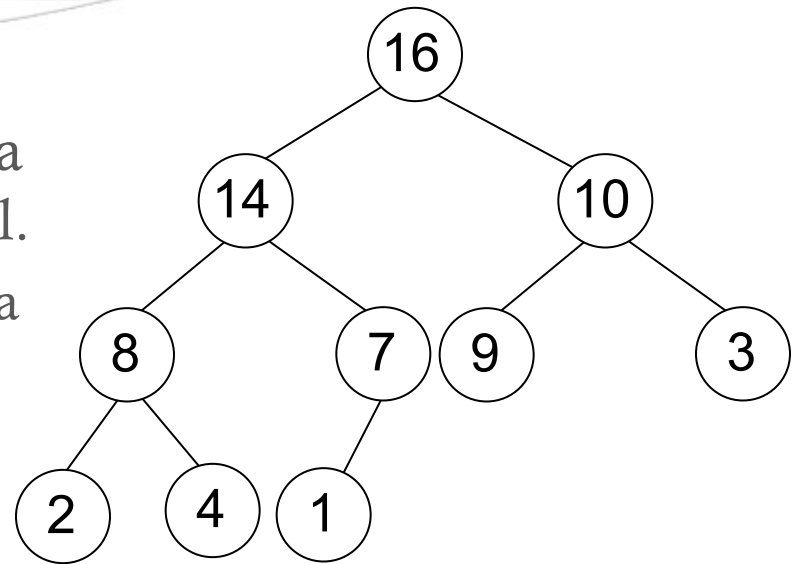
# Heaps

- ◆ Para manter o heap ordenado e facilitar a localização do elemento de maior prioridade, o mesmo é colocado no **nó raiz**.
- ◆ Com isso, a prioridade de um nó X qualquer será, sempre, maior ou igual a de seus dependentes.
- ◆ A raiz é sempre o maior (ou o menor) elemento do heap.
- ◆ Existem dois tipos de heaps:
  - ◆ Heaps de máximo (max heap):
    - ◆ O valor de todos os nós são menores que os de seus respectivos pais;
  - ◆ Heaps de mínimo (min heap):
    - ◆ O valor de todos os nós são maiores que os de seus respectivos pais.
- ◆ Em uma heap de máximo, o maior valor do conjunto está na raiz da árvore.
- ◆ Na heap de mínimo a raiz armazena o menor valor existente.



# Heaps

- Uma heap **pode** ser implementada como uma matriz unidimensional.
- O pai do elemento da posição  $i$  é a parte inteira do elemento  $(i/2)$ ;
- O filho esquerdo do elemento da posição  $i$  é o da posição  $(2 * i)$ ;
- O filho direito do elemento da posição  $i$  é o da posição  $(2 * i)+1$ .



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

# Heaps

- ◆ Para armazenar uma árvore de altura  $h$  precisamos de um array de  $2h - 1$  (número de nós de uma árvore cheia de altura  $h$ ).
- ◆ As operações mais comuns em heaps são:
  - ◆ **Inserção**
  - ◆ **Remoção**
- ◆ Quando um elemento é adicionado a uma heap, pode ser adicionado **de baixo para cima** ou de **cima para baixo**.
- ◆ Quando o elemento é removido, **em geral** remove-se o elemento mais à direita do nível da base.

# Heaps

- ◆ Dois procedimentos de migração do nó são utilizados nas operações de inserção e remoção : “**Subir**” e “**Descer**”:
  - ◆ **Subir** ( $i, n, H$ ) migra o nó  $i$  para cima na heap  $H$
  - ◆ **Descer** ( $i, n, H$ ) migra o nó  $i$  para baixo na heap  $H$ 
    - ◆ (sendo  $n$  o número total de nós da árvore/heap)
- ◆ Para inserir, pode-se colocar o novo valor na posição  $n+1$  da heap e chamar **Subir**.
- ◆ Para remover pode-se substituir a raiz pela folha da posição  $n$  ( $H[1]$ ) por  $H(n)$  e chamar **Descer**.

# Heaps

```
Subir (i, n, H [1 .. n])
{
  se  $i > 1$  e  $H[\text{Pai}(i)] < H[i]$  então
  {
     $H[i], H[\text{Pai}(i)] \leftarrow H[\text{Pai}(i)], H[i]$ 
    Subir (Pai(i),n,H)
  }
}
```

```
Descer (i, n, H [1 .. n])
{
  se  $\text{Dir}(i) \leq n$  e  $H[\text{Dir}(i)] > H[\text{Esq}(i)]$ 
  então filho  $\leftarrow \text{Dir}(i)$ 
  senão filho  $\leftarrow \text{Esq}(i)$ 
  se filho  $\leq n$  e  $H[\text{filho}] > H[i]$  então {
     $H[i], H[\text{filho}] \leftarrow H[\text{filho}], H[i]$ 
    Descer (filho, n, H)
  }
}
```

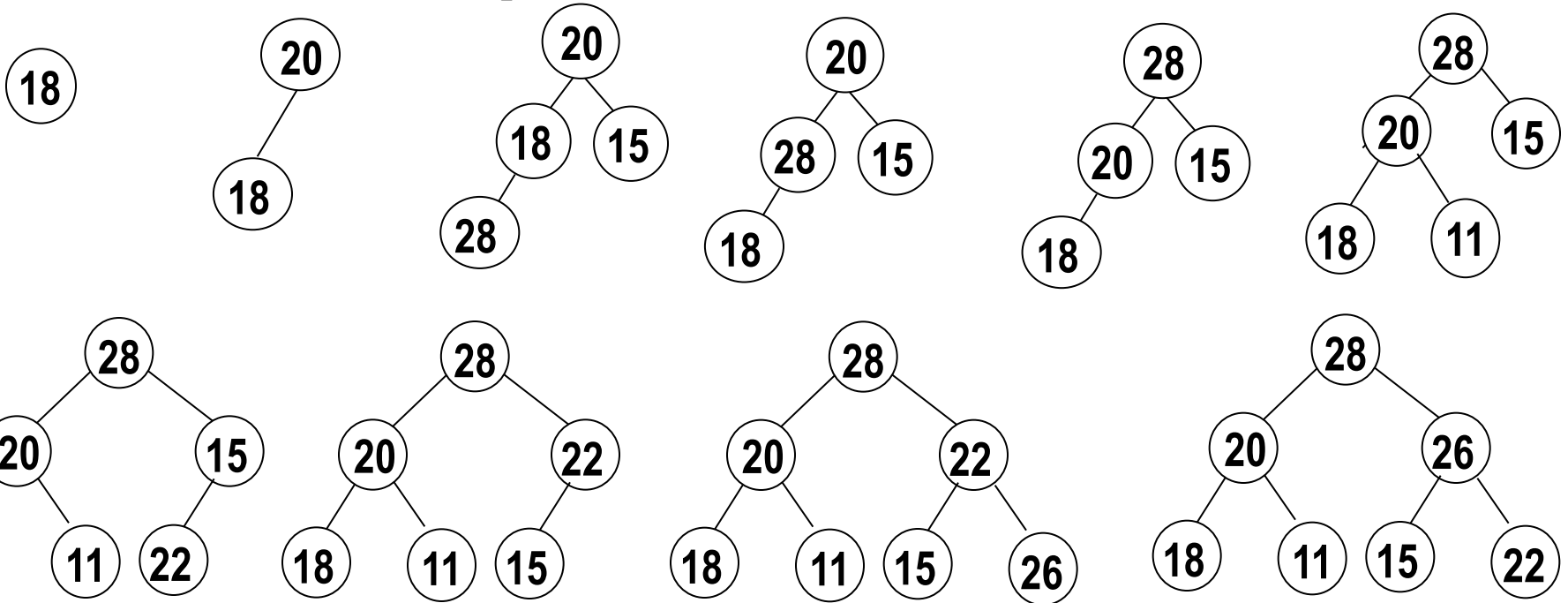
# Heaps

🔹 Criação de heaps:

🔹 inserção de elementos:

🔹 método de cima para baixo.

0	1	2	3	4	5	6
18	20	15	28	11	22	26



# Heaps

- ◆ Criação de heaps: inserção de elementos: método de baixo para cima.

```
BuildHeap(data[ ])
```

```
for (i= n/2-1; i>=0; i--)
```

```
Descer (i,n-1,data)
```

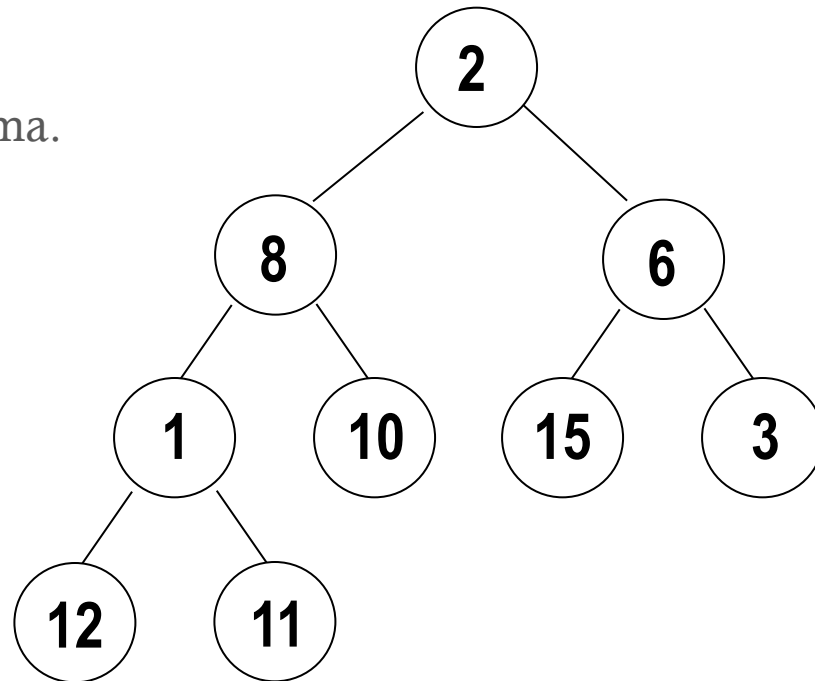
- ◆ Quando um elemento é analisado, suas duas subárvores devem ser convertidas em heaps.
- ◆ Quando a propriedade de heap é desfeita, ela é reestabelecida movendo para baixo o elemento analisado.

# Heaps

data

0	1	2	3	4	5	6	7	8
2	8	6	1	10	15	3	12	11

- Criação de heaps:
- inserção de elementos:
  - método de baixo para cima.



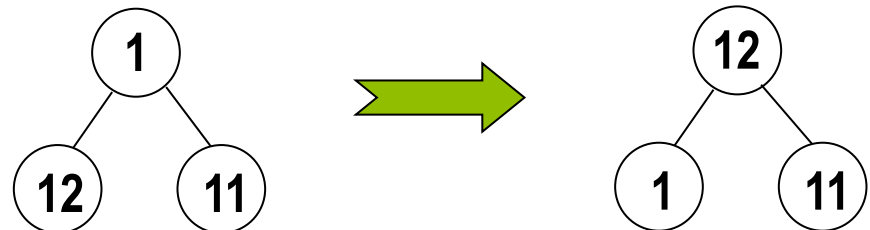
# Heaps

- Criação de heaps:
  - inserção de elementos:
    - método de baixo para cima.

data

0	1	2	3	4	5	6	7	8
2	8	6	1	10	15	3	12	11

- Começa-se a partir do último nó não-folha que é dado por  $n/2-1$ ,  $n$  é o tamanho da matriz.
  - Último nó não-folha  $\rightarrow$  data[3] = 1
  - Se o último nó não-folha é menor que um de seus filhos ele é trocado com seu maior filho.





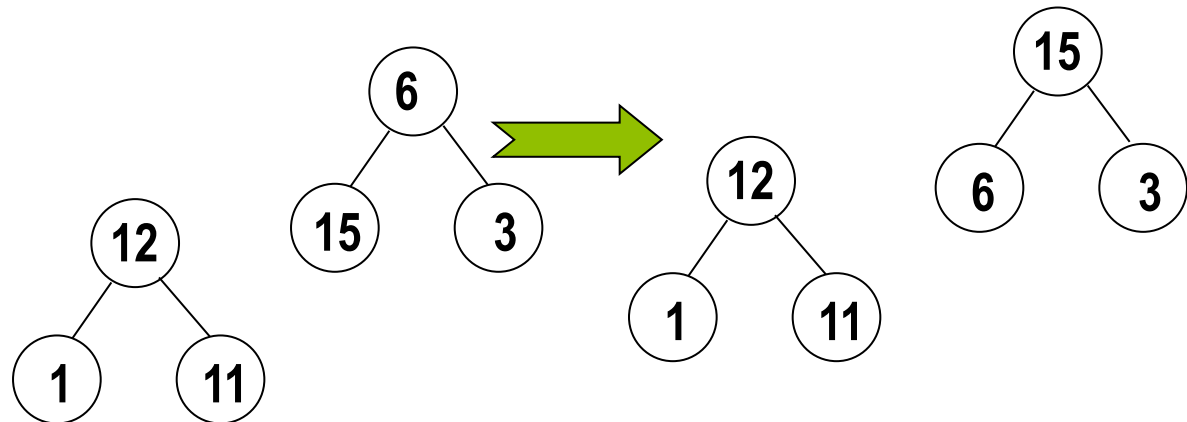
# Heaps

- Criação de heaps:
  - inserção de elementos:
    - método de baixo para cima.

data

0	1	2	3	4	5	6	7	8
2	8	6	12	10	15	3	1	11

- Considera-se o elemento  $n/2 - 2 \rightarrow data[2] = 6$



# Heaps

- 🔥 Criação de heaps:

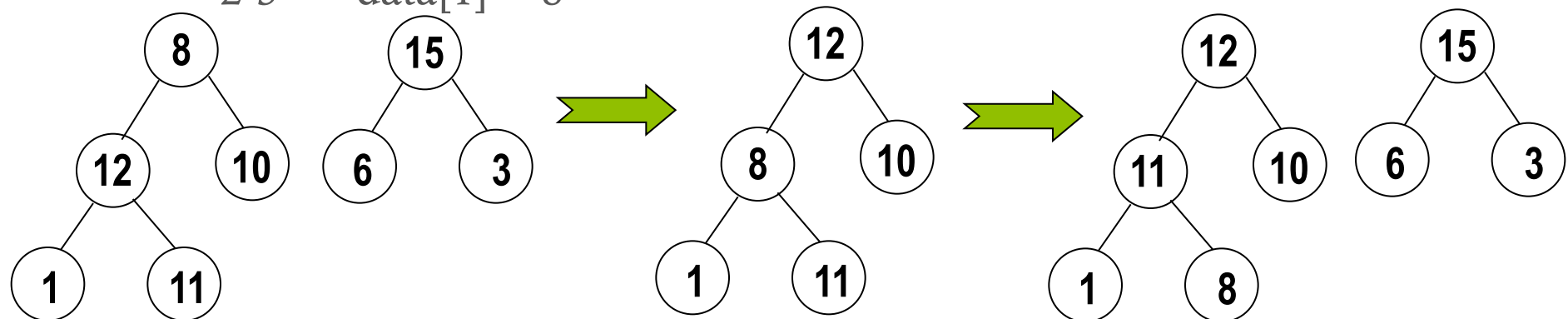
- 🔥 inserção de elementos:

- 🔥 método de baixo para cima.

data

0	1	2	3	4	5	6	7	8
2	8	15	12	10	6	3	1	11

- 🔥 Considera-se o elemento  $n/2-3 \rightarrow \text{data}[1] = 8$

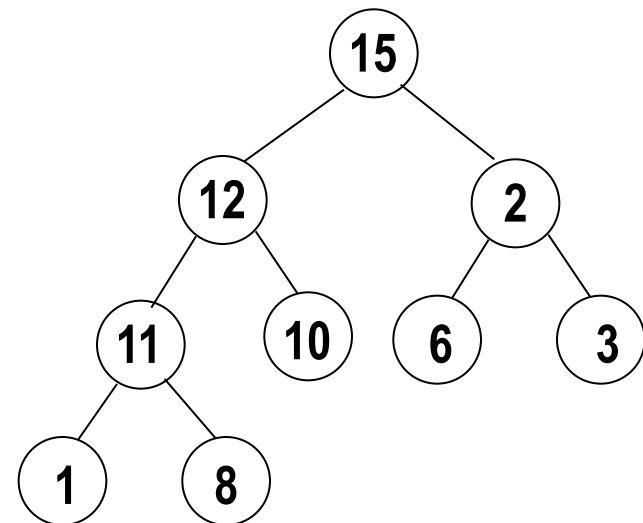
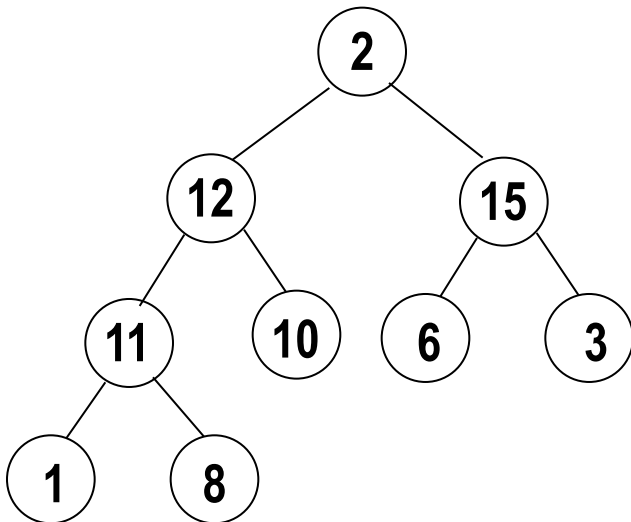


# Heaps

- Criação de heaps:
  - inserção de elementos:
    - método de baixo para cima.

data								
0	1	2	3	4	5	6	7	8
2	12	15	11	10	6	3	1	8

- Considera-se o elemento  $n/2-4$   
→  $data[0] = 2$

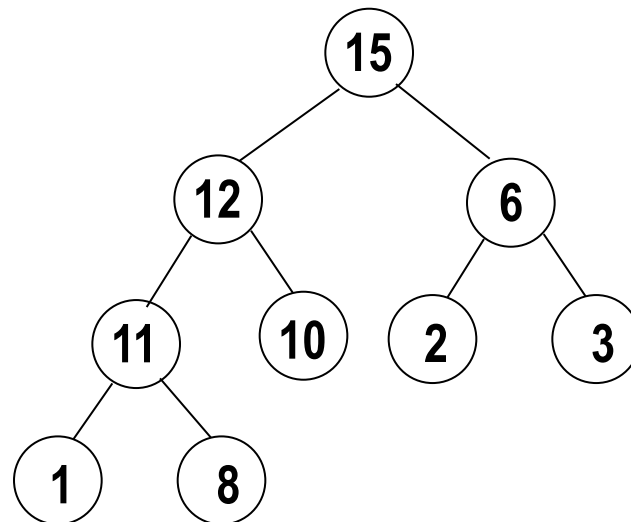


# Heaps

- Criação de heaps:
  - inserção de elementos:
    - método de baixo para cima.

data

0	1	2	3	4	5	6	7	8
15	12	6	11	10	2	3	1	8



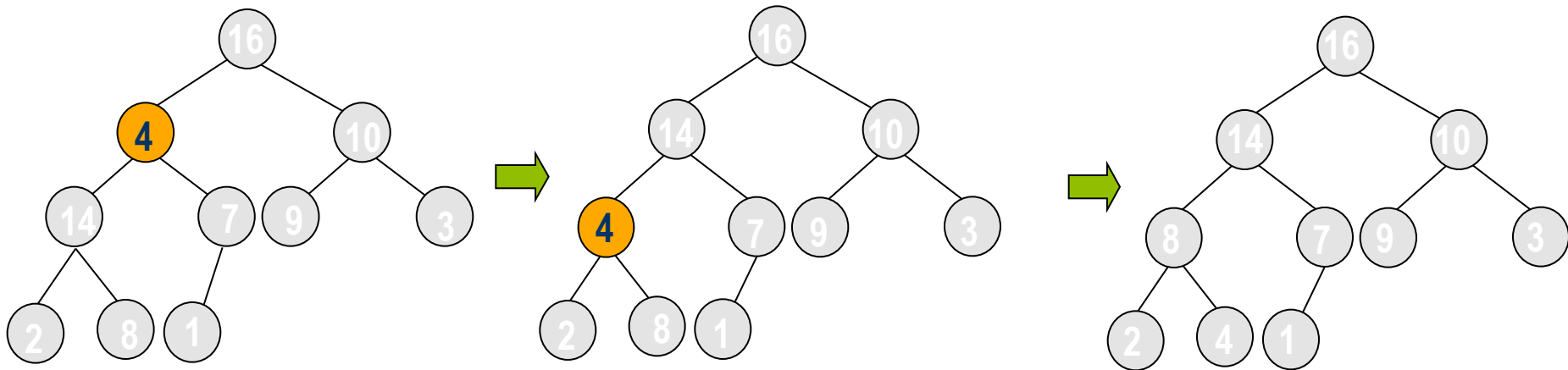
# Heaps

- Heapify

- Reorganiza heaps

- Supõe que as árvores binárias correspondentes a Esquerda(i) e Direita(i) são heaps, mas  $A[i]$  pode ser menor que seus filhos.

- Exemplo:



# Heaps

- Algoritmo
- ```
Heapify(A, i)
  Esquerdo <- Filho Esquerdo(i)
  Direito <- Filho Direito(i)
  Se Esquerdo > i (pai)
    Maior <- Esquerdo
  Senão
    Maior <- i
  Se Direito > que Maior
    Maior <- Direito
  Se Maior != i(pai)
    troca(A,i,Maior)
  Heapify Maximo(A, Maior)
```

# Filas de prioridade

- ◆ Em muitas aplicações, dados de uma coleção são acessados por ordem de prioridade.
- ◆ A prioridade associada a um dado pode ser qualquer parâmetro específico como: tempo, custo, etc.
- ◆ Numa fila de prioridade atribui-se um valor de prioridade a cada elemento e as operações de inserção e remoção passam a ser feitas em função dessa prioridade.

# Filas de prioridade

- ◆ Aplicações:
  - ◆ Escalonamento de processos em um sistema multi-programável.
    - ◆ A fila de prioridade máxima mantém o controle dos processos a serem executados e de suas prioridades relativas.
    - ◆ Quando um processo termina ou é interrompido, o processo de prioridade mais alta é selecionado dentre os processos pendentes.
    - ◆ As chaves podem também representar o tempo em que eventos devem ocorrer.
    - ◆ Métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
    - ◆ Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.



# Filas de prioridade

- ◆ Nesse contexto, as operações que se costuma querer implementar eficientemente são:
  - ◆ Seleção do elemento com maior (ou menor) prioridade
  - ◆ Remoção do elemento de maior (ou menor) prioridade
  - ◆ Inserção de um novo elemento
- ◆ As heaps são estruturas próprias para implementação de listas de prioridade.
- ◆ A raiz de uma heap contém a chave (prioridade) de menor ou maior valor.

# Filas de prioridade

## ◆ Inserção de um novo elemento

- ◆ Para colocar um elemento na fila, o elemento é adicionado no fim da heap como a última folha.
- ◆ A restauração da propriedade de heap é obtida movendo-se a última folha em direção a raiz segundo o seguinte algoritmo:

**heapEnqueue(e1)**

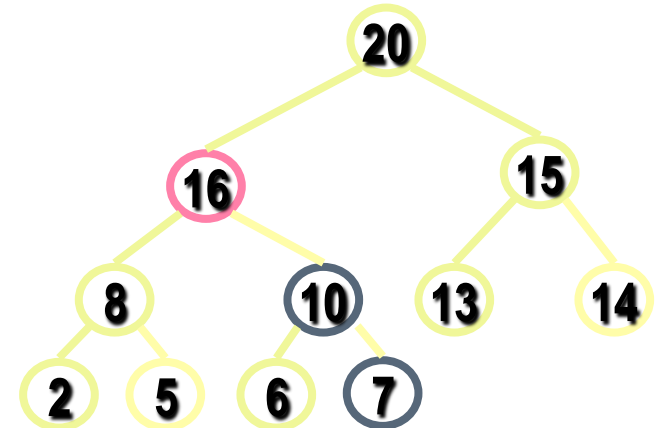
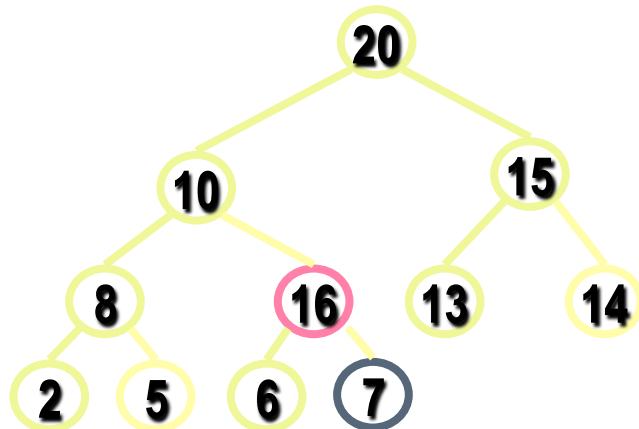
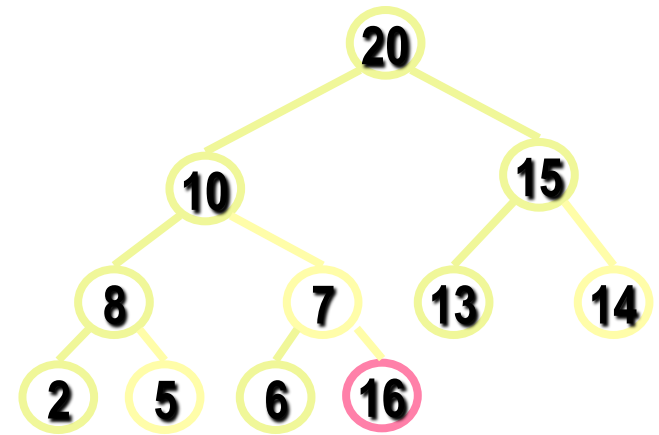
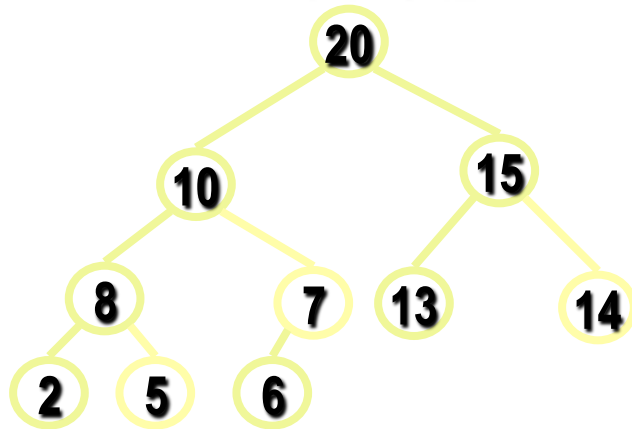
colocar **e1** no final da heap;

while **e1** não é a raiz e **e1** > ascendente(e1);

troca (**e1**, ascendente);

# Filas de prioridade

◆ Inserção de um novo elemento



# Filas de prioridade

- Remoção de um elemento

- Para remover um elemento da heap, deve-se remover a **raiz** da heap, pois esse é o elemento com a **maior prioridade**.
- Coloca-se a última folha no lugar da raiz e restaura-se a propriedade de heap movendo-se da raiz para baixo na árvore:

heapDequeue(**e**<sub>l</sub>)

  Extrai-se a raiz;

  Coloca-se a última folha (**p**) em seu lugar;

  Remove-se a última folha;

  while **p** não é folha e **p** < um dos descendentes

    troque **p** com o maior ascendente

# Filas de prioridade

- Remoção de um elemento

