# Análise de Algoritmos
# Listas de Prioridades

Nelson Cruz Sampaio Neto
nelsonneto@ufpa.br

Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Faculdade de Computação

10 de junho de 2025

# Introdução

- Em muitas aplicações, uma característica importante que distingue os dados armazenados em uma certa estrutura é uma **prioridade** atribuída a cada um deles.
- Suponha que os dados de uma tabela correspondam a tarefas a serem realizadas com uma certa prioridade.
- Para encontrar a ordem desejada de execução das tarefas, um algoritmo deve, sucessivamente, escolher o dado de maior (ou menor) prioridade e retirá-lo da tabela.
- Além disso, o algoritmo deve ser capaz de introduzir novos dados e alterar a prioridade das tarefas.

# Introdução

- Lista de prioridades é uma estrutura para manutenção de um conjunto de dados, cada qual com um valor associado chamado **prioridade** ou **chave**.
- Essa prioridade é, em geral, definida por meio de um valor numérico e armazenada em algum campo da estrutura.
- A prioridade associada ao dado pode ser qualquer coisa: tempo, custo... mas precisa ser um escalar.
- Dois tipos de listas de prioridades: **máxima** e **mínima**.
- Por exemplo, execução de processos (máxima) e simulação orientada a eventos (mínima).

# Introdução

- As operações básicas a serem efetuadas com os elementos de uma lista de prioridades são as seguintes:
  - **seleção** do elemento de maior (ou menor) prioridade;
  - **inserção** de um novo elemento;
  - **remoção** do elemento de maior (ou menor) prioridade;
  - **alteração** da prioridade de um dado elemento;
  - **construção** de uma lista de prioridades.
- Por conveniência, consideraremos daqui pra frente apenas listas de prioridades máximas.

# Implementação por lista não ordenada

- A inserção e, consequentemente, a construção são triviais.
- O novo nó da lista pode ser colocado em qualquer posição conveniente, dependendo do tipo de alocação utilizada, sequencial ou encadeada.
- A seleção e a remoção, entretanto, implicam percorrer a lista em busca do elemento de maior prioridade.
- A alteração de prioridade não afeta a organização da lista.

# Implementação por lista não ordenada

- Então, para uma lista não ordenada de $n$ elementos, cada uma das operações requer o seguinte número de passos:
  - seleção: $O(n)$
  - inserção: $O(1)$
  - remoção: $O(n)$
  - alteração: $O(1)$
  - construção: $O(n)$

# Implementação por lista ordenada

- A remoção e a seleção são imediatas porque, estando as prioridades ordenadas, o elemento que interessa é o primeiro (ou o último).
- A inserção, entretanto, obriga a um percurso pela lista para procurar sua posição correta.
- A alteração de prioridade é semelhante a uma inserção, ou seja, é preciso percorrer a lista para trás ou para frente.
- A construção exige uma ordenação da lista e a complexidade depende do algoritmo de ordenação empregado.

# Implementação por lista ordenada

- Então, para uma lista ordenada de $n$ elementos, cada uma das operações requer o seguinte número de passos:
  - seleção: $O(1)$
  - inserção: $O(n)$
  - remoção: $O(1)$
  - alteração: $O(n)$
  - construção: $O(n \, log \, n)$, sendo que essa complexidade pode mudar dependendo do algoritmo de ordenação utilizado.

# Implementação por árvore balanceada

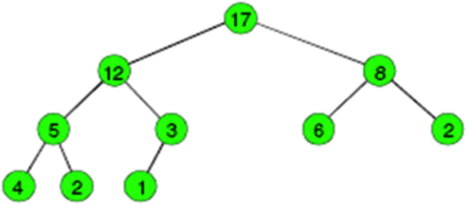
- Uma árvore balanceada deve manter o custo de seleção na mesma ordem de grandeza de uma árvore ótima, ou seja, $O(log\ n)$.
- Esse custo deve se manter ao longo de toda utilização da estrutura, inclusive após inserções e remoções.
- Para alcançar essa finalidade, a estrutura deve ser alterada, periodicamente, de forma a se moldar aos novos dados.
- O custo dessas alterações, contudo, se mantém em $O(log\ n)$.

# Implementação por árvore balanceada

- Então, para uma árvore balanceada de $n$ elementos, cada uma das operações requer o seguinte número de passos:
  - seleção: $O(log\ n)$
  - inserção: $O(log\ n)$
  - remoção: $O(log\ n)$
  - alteração: $O(log\ n)$
  - construção: $O(n\ log\ n)$

# Implementação por *heap*

- A estrutura de dados *heap* é um objeto arranjo que pode ser visualizado como uma árvore binária completa.
- Um *heap* deve satisfazer uma das seguintes condições:
  - Todo nó deve ter valor maior ou igual que seus filhos (**heap máximo**). O maior elemento é armazenado na raiz.
  - Todo nó deve ter valor menor ou igual que seus filhos (**heap mínimo**). O menor elemento é armazenado na raiz.




- Acima, um *heap* máximo de altura 3. Note que o último nível pode não conter os nós mais à direita.

- Tendo em vista que um *heap* de $n$ elementos é uma árvore binária completa, sua altura $h$ é $\lfloor log_2(n) \rfloor$.

- A altura de um nó $i$ é o número de nós do maior caminho de $i$ até um de seus descendentes. As folhas têm altura zero.

- Se o *heap* for uma árvore binária completa com o último nível cheio, seu número total de elementos será $2^{h+1} - 1$.

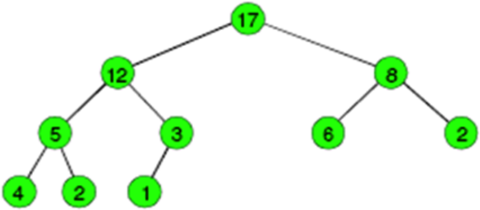
- Se o último nível do *heap* tiver apenas um elemento, seu número total de elementos será $2^h$.

- Logo, o número $n$ de elementos de um *heap* é dado por

$$2^h \leq n \leq 2^{h+1} - 1$$

# Implementação por *heap*

- Na prática, quando se trabalha com *heap*, recebe-se um vetor que será representado por árvore binária da seguinte forma:
  - Raiz da árvore: primeira posição do vetor;
  - Filhos do nó na posição $i$: posições $2i$ e $2i + 1$; e
  - Pai do nó na posição $i$: posição $\lfloor \frac{i}{2} \rfloor$.
- Exemplo: O vetor $A = [17\ 12\ 8\ 5\ 3\ 6\ 2\ 4\ 2\ 1]$ pode ser representado pela árvore binária (*max-heap*) abaixo.



# Algoritmos para *heap*

- A representação em árvores permite relacionar os nós do *heap* da seguinte forma:

```
Pai (i)
1. retorne (int) i/2

Esq (i)
1. retorne 2*i

Dir (i)
1. retorne 2*i + 1
```

# Implementação por *heap*

- A seleção é trivial, dado que o elemento de maior prioridade está na raiz da árvore.

```
HEAP-MAXIMUM (A)
1. retorne (A[1])
```

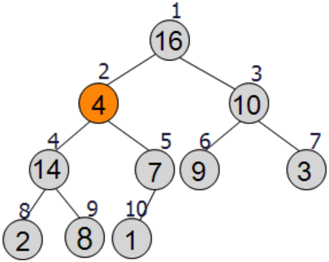
- As operações de alteração, inserção e remoção são realizadas em tempo logarítmico, já que a altura de *heap*, que é uma árvore binária completa, é $O(log\ n)$.
- Contrariando a intuição, a construção de um *heap* pode ser realizada em tempo inferior ao da ordenação.
- De fato, a construção de um *heap* requer não mais do que tempo $O(n)$, como será mostrado mais a frente.

# Implementação por *heap*

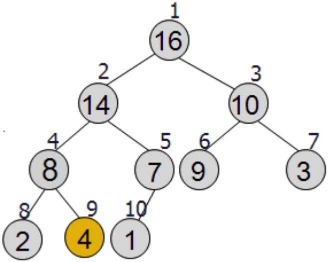
- Então, os parâmetros indicadores de eficiência de uma lista de prioridades implementada por *heap* são os seguintes:
  - seleção: $O(1)$
  - inserção: $O(\log n)$
  - remoção: $O(\log n)$
  - alteração: $O(\log n)$
  - construção: $O(n)$

# Alteração de prioridade

- Seja o vetor $A = [16\ 15\ 10\ 14\ 7\ 9\ 3\ 2\ 8\ 1]$, onde a prioridade do nó 2 foi alterada de 15 para 4 (figura a). A figura b mostra o resultado da correção.



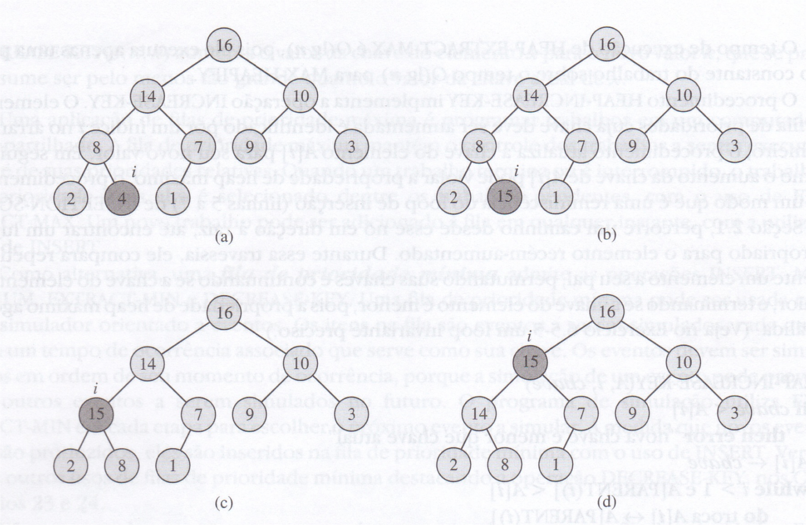
(a)



(b)

# Alteração de prioridade

- Agora, a prioridade do nó 9 é alterada de 4 para 15.



# Algoritmo: Alteração de prioridade

- Essa operação altera a chave do elemento *i* e, em seguida, o situa na posição correta de sua nova prioridade.
- Associa-se a diminuição de prioridade à "descida" na árvore, e o aumento à "subida".

```
HEAP-CHANGE (A, i, chave)
1. se (A[i] > chave)
2.    A[i] := chave
3.    HEAP-DECREASE-KEY (A, n, i)
4. senão
5.    A[i] := chave
6.    HEAP-INCREASE-KEY (A, i)
7. fim
```

# Algoritmo: Decremento de prioridade

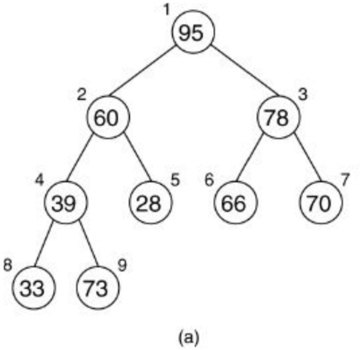
```
HEAP-DECREASE-KEY (A, n, i)
  j := 2 * i
  enquanto j <= n faça
    maior := j
    se (maior < n) e (A[maior] < A[maior + 1])
       maior := maior + 1
    fim
    se (A[i] < A[maior])
       troca A[i] com A[maior]
       i := maior
       j := 2 * i
    senão
       j := n + 1
    fim
  fim
```

# Algoritmo: Incremento de prioridade

```
HEAP-INCREASE-KEY (A, i)
1. j := (int) i/2
2. enquanto (i > 1) e (A[j] < A[i])
3.    troca A[i] com A[j]
4.    i := j
5.    j := (int) i/2
6. fim
```

# Inserção de um elemento

- Suponha um vetor com oito elementos e a inserção de um novo elemento de prioridade 73 na posição 9 (figura a). A figura b mostra o resultado da correção.



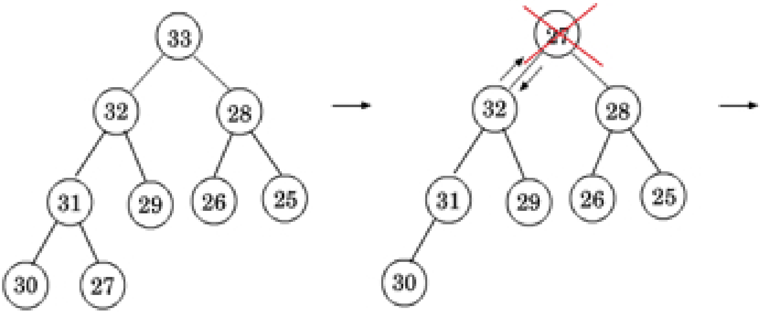
# Algoritmo: Inserção de um elemento

- A inserção de um novo elemento corresponde a assumir o *heap* com $n + 1$ elementos e corrigir a prioridade do último elemento, supondo-a aumentada.

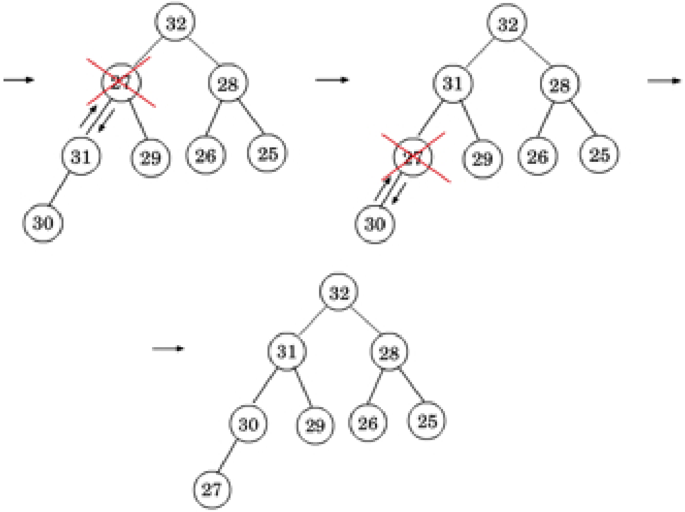
```
HEAP-INSERT (A, n, chave)
1. n := n + 1
2. A[n] := chave
3. HEAP-INCREASE-KEY (A, n)
```

# Remoção do elemento com maior prioridade

- Exemplo: As figuras abaixo ilustram a operação de remoção do elemento com maior prioridade, ou seja, a raiz.



# Remoção do elemento com maior prioridade



# Algoritmo: Remoção do elemento com maior prioridade

- Com a remoção da raiz, o último elemento será o substituto do primeiro e o *heap* passa a ter $n - 1$ posições.
- Em seguida, a prioridade do primeiro elemento precisa ser corrigida, supondo-a diminuída.

```
HEAP-EXTRACT-MAX (A, n)
1. se (n < 1)
2.    então erro "heap vazio"
3. fim
4. maximo := A[1]
5. A[1] := A[n]
6. n := n - 1
7. HEAP-DECREASE-KEY (A, n, 1)
```

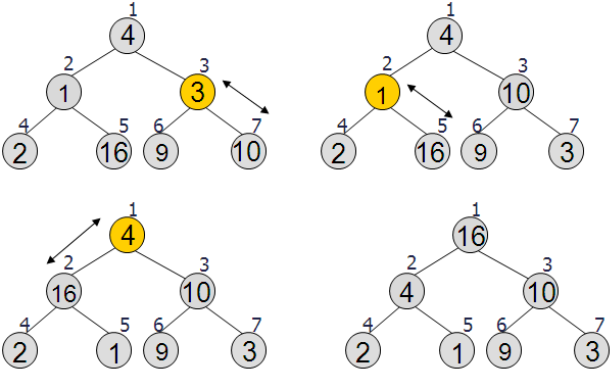
- O procedimento MAX-HEAP abaixo converte um vetor $A$ de $n$ elementos em um *heap* máximo.

```
MAX-HEAP (A, n)
1. para i = n/2 até 1 faça
2.     HEAP-DECREASE-KEY (A, n, i)
3. fim
```

- Os elementos de $A[\frac{n}{2} + 1]$ até $A[n]$ correspondem às folhas da árvore e, portanto, são *heaps* de um elemento.

- Logo, basta chamar o procedimento HEAP-DECREASE-KEY para os elementos de $A[\frac{n}{2}]$ até $A[1]$.

- As figuras abaixo ilustram a operação do procedimento MAX-HEAP para o vetor $A = [4\ 1\ 3\ 2\ 16\ 9\ 10]$.



# Construção de um *heap*

- O tempo de execução do MAX-HEAP é $O(n \log n)$.
- Embora correto, esse limite superior **não** é assintoticamente restrito.
- De fato, o tempo de execução do HEAP-DECREASE-KEY sobre um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas.
- A análise mais restrita se baseia em duas propriedades:
  - Um *heap* de $n$ elementos tem altura $\lfloor log_2(n) \rfloor$; e
  - No máximo $\lceil \frac{n}{2^{h+1}} \rceil$ nós de qualquer altura $h$.

# Construção de um *heap*

- O tempo exigido pelo MAX-HEAP quando é chamado em um nó de altura $h$ é $O(h)$.

- Assim, expressa-se o custo total do MAX-HEAP por

$$\sum_{h=0}^{\lfloor log_2(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor log_2(n) \rfloor} \frac{h}{2^h})$$

$$\leq O(n \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Sabe-se que $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

Desse modo, o tempo de execução do MAX-HEAP pode ser limitado como $O(n)$.

# Resumo

- A tabela abaixo mostra a complexidade no tempo para diferentes implementações de listas de prioridades.

| **Operação** | **Lista** | **Lista ordenada** | **Árvore balanceada** | **Heap binário** |
|---|---|---|---|---|
| Seleção-max | $O(n)$ | $O(1)$ | $O(\log n)$ | $O(1)$ |
| Remoção-max | $O(n)$ | $O(1)$ | $O(\log n)$ | $O(\log n)$ |
| Alteração | $O(1)$ | $O(n)$ | $O(\log n)$ | $O(\log n)$ |
| Inserção | $O(1)$ | $O(n)$ | $O(\log n)$ | $O(\log n)$ |
| Construção | $O(n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n)$ |

- Percebe-se um *trade-off* na implementação por listas, apesar de serem extremamente simples de codificar.
- Para maior eficiência, usa-se a implementação por *heap*.

# Exercícios

1. Um vetor de números inteiros que está ordenado de forma crescente é um *heap* mínimo?
2. Onde em um *heap* máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?
3. Todo *heap* é uma árvore de busca binária? Por quê?
4. O vetor $[23, 17, 14, 6, 13, 10, 1, 5, 7]$ é um *heap* máximo? Caso negativo, transforme-o em um *heap* máximo.

# Exercícios

5. Seja $S$ o *heap* especificado a seguir:

   $S = [92 \;\; 91 \;\; 90 \;\; 47 \;\; 85 \;\; 34 \;\; 20 \;\; 40 \;\; 46]$.

   Determine o *heap* resultante da alteração de prioridade do seu quinto nó de 85 para 93.

6. Ilustre a operação do procedimento HEAP-EXTRACT-MAX sobre o *heap* $A = [15\ 13\ 9\ 5\ 12\ 1]$.

7. Ilustre a operação do procedimento HEAP-INSERT sobre o *heap* $A = [15\ 13\ 9\ 5\ 12\ 1]$ ao inserir um elemento com prioridade igual a 16.