Projeto de Algoritmos II
Listas de Prioridades

Nelson Cruz Sampaio Neto
nelsonneto@ufpa.br

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

19 de agosto de 2019

- Em muitas aplicações, uma característica importante que distingue os dados armazenados em uma certa estrutura é uma **prioridade** atribuída a cada um deles.
- P.e., 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 pode ser definida como uma estrutura de dados para manutenção de um conjunto de dados, cada qual com um valor associado chamado **chave**.
- Essa chave é, em geral, definida através 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 dados 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; e
  - **alteração** da prioridade de um dado elemento.
- 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, mas também requer a busca do elemento.

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: $O(n)$
  - construção: $O(n)$

Implementação por lista ordenada

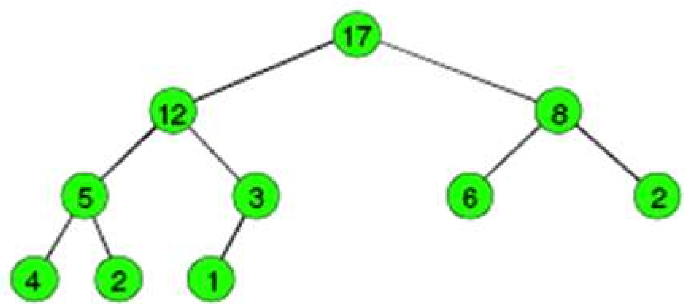
- A remoção e a seleção são imediatas porque, estando as prioridades já ordenadas, o primeiro elemento é o que interessa.
- A inserção, entretanto, obriga a um percurso pela lista para procurar sua posição correta.
- A alteração de prioridade é semelhante a uma nova inserção.
- A construção exige uma ordenação prévia da lista. Logo, sua complexidade no tempo 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: $O(n \log n)$ utilizando o algoritmo de ordenação *Heapsort*, por exemplo.

Implementação por *heap*

- A estrutura de dados *heap* é uma lista linear que pode ser visualizada 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.

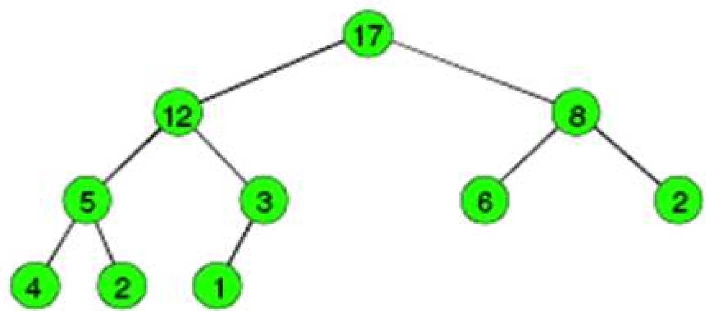
Implementação por *heap*

- Tendo em vista que um *heap* de n elementos é uma árvore binária completa, sua altura h é $\lfloor \log n \rfloor$.
- 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*, tem-se uma lista que será representada por árvore 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: A lista de prioridades $A = [17\ 12\ 8\ 5\ 3\ 6\ 2\ 4\ 2\ 1]$ é um *max-heap* e pode ser visualizada pela árvore abaixo.



Implementação por *heap*

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

- 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.

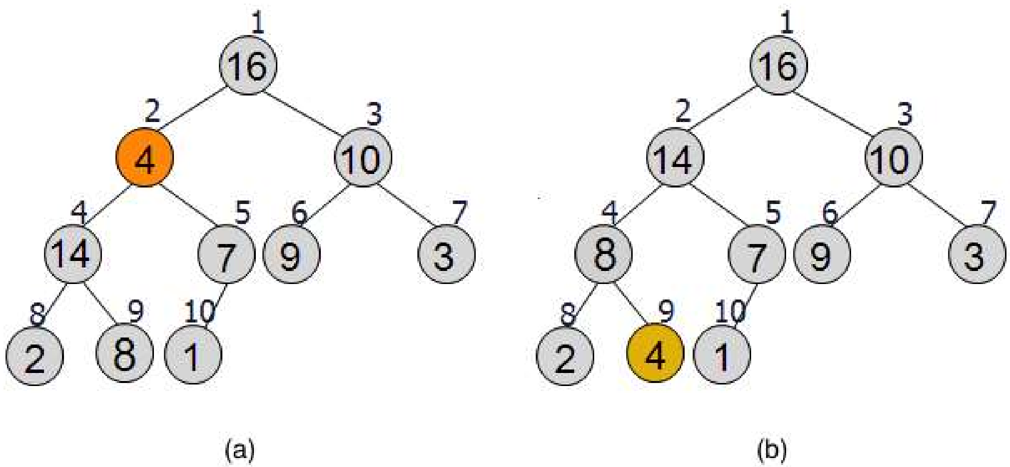
- De fato, a construção de um *heap* requer não mais do que tempo $O(n)$, como será mostrado mais a frente.

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: $O(n)$

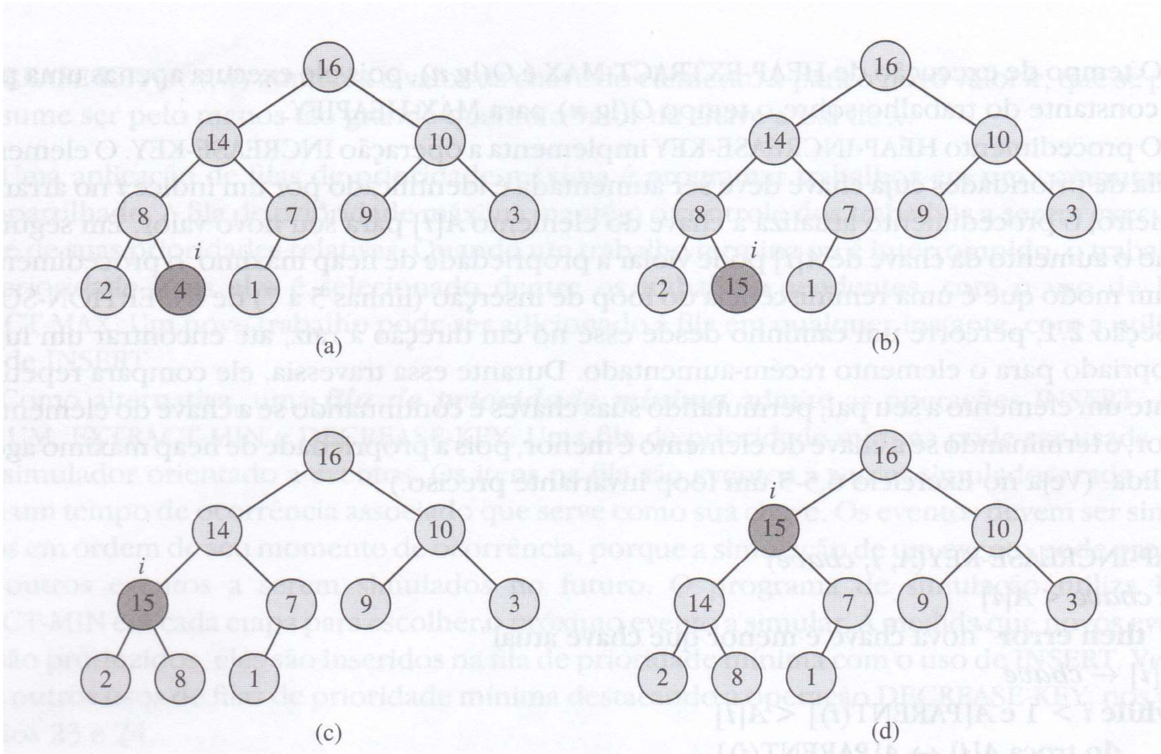
Alteração de prioridade

- Seja a lista de prioridades $A = [16\ 15\ 10\ 14\ 7\ 9\ 3\ 2\ 8\ 1]$, onde se deseja alterar a prioridade do nó 2 de 15 para 4, como é visto na figura abaixo.



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 então 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
```

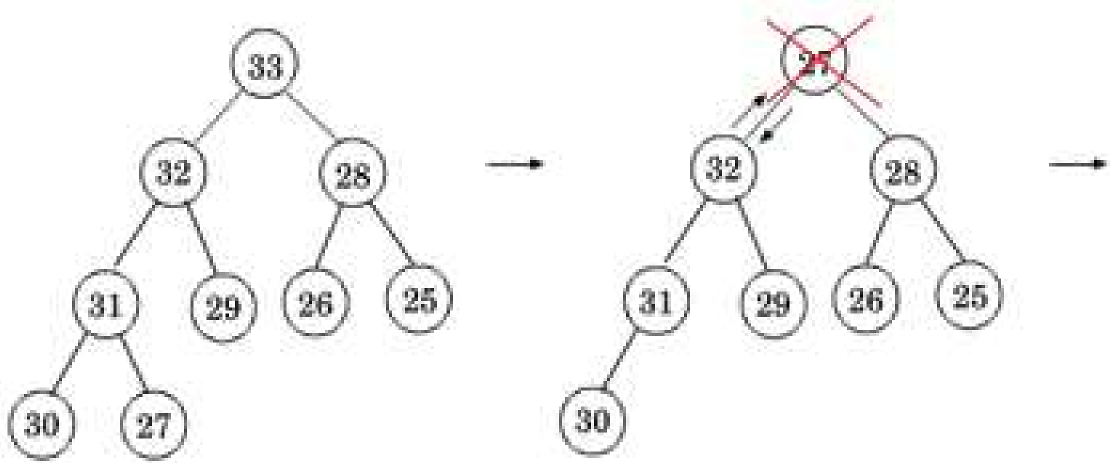
Algoritmo: Inserção de um elemento

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

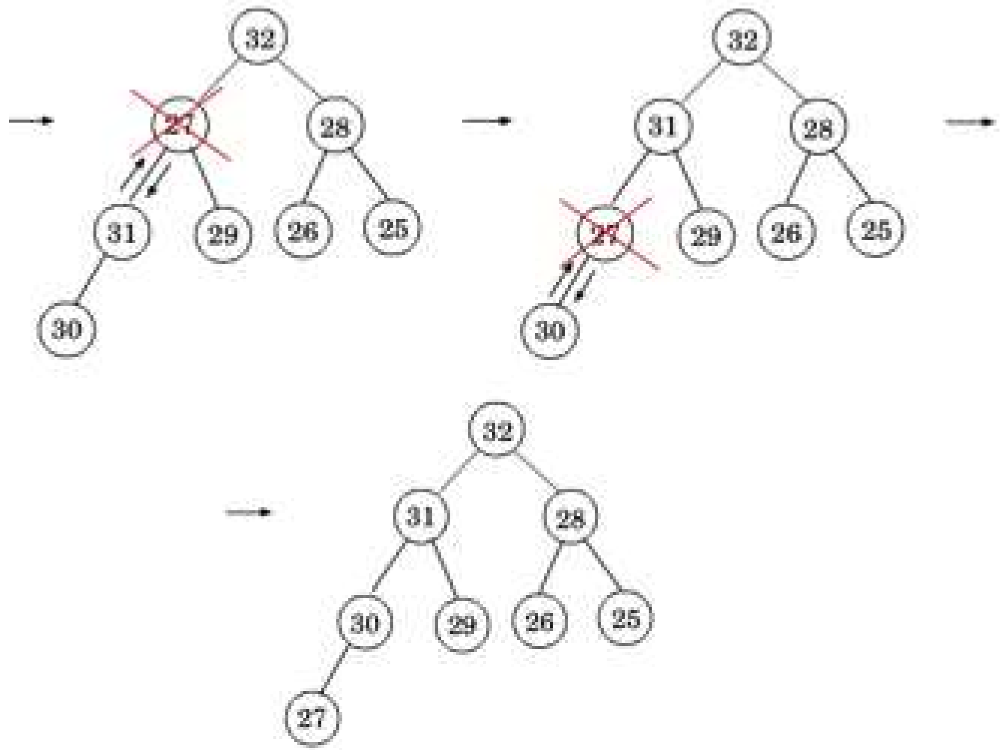
```
HEAP-INSERT (A, n, chave)
1. A[n + 1] := chave
2. n := n + 1
3. HEAP-INCREASE-KEY (A, n)
```

Remoção de um elemento

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



Remoção de um elemento



Algoritmo: Remoção de um elemento

- Com a remoção da raiz, o último elemento será o substituto do primeiro e o *heap* passa a ter $n - 1$ posições.
- Em seguida, claro que a prioridade do primeiro elemento deve 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)
8. retorna (maximo)
```

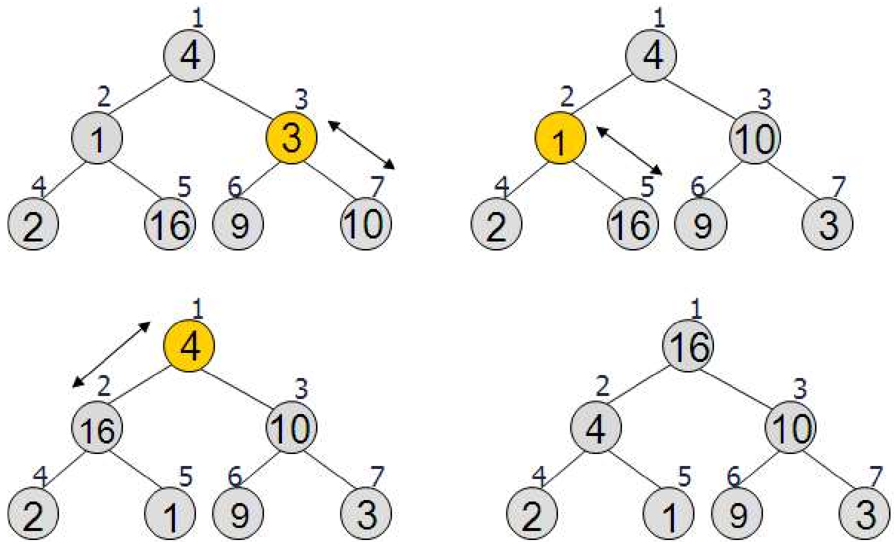
- O procedimento MAX-HEAP abaixo converte uma lista A de n elementos em um *heap* máximo.

```
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 $A[1]$.

- Exemplo: A figura abaixo ilustra a operação do procedimento MAX-HEAP para a lista $A = [4\ 1\ 3\ 2\ 16\ 9\ 10]$.



Construção de um *heap*

- O tempo de execução do MAX-HEAP é $O(n \log n)$.
- Contudo, 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.
- Então, o tempo de execução do MAX-HEAP é da ordem do somatório das alturas dos vértices analisados no algoritmo.
- 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*

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

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

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

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 $O(n)$.

- Ou seja, podemos transformar uma lista desordenada em um *heap* máximo em tempo linear!

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	$O(n)$	$O(nlog(n))$	$O(nlog(n))$	$O(n)$

- 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. A sequência $[23, 17, 14, 6, 13, 10, 1, 5, 7]$ é um *heap* máximo? Caso negativo, transforme-a em um *heap* máximo.
3. Onde em um *heap* máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?
4. Todo *heap* é uma árvore binária de pesquisa? Por quê?
5. Provar ou dar contra-exemplo:

   Seja S um *heap* máximo e s_i, s_j chaves de S tais que $i < j$ e $s_i < s_j$. Então, a sequência obtida pela troca de posições de s_i com s_j é também um *heap* máximo.

Exercícios

6. Seja S o *heap* especificado a seguir:

   92 85 90 47 91 34 20 40 46

   Determine o *heap* resultante da alteração de prioridade do seu quinto nó de 91 para: (i) 93; (ii) 19.

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

8. Ilustre a operação do algoritmo HEAP-INSERT sobre o vetor $A = [15\ 13\ 9\ 5\ 12\ 1]$ ao inserir um elemento com chave 16.

9. Mostre o *heap* min-max resultante da alteração de prioridade do nó-raiz da árvore da Figura 1, de 4 para 41.

10. Construir um *heap* min-max com as seguintes prioridades:
    10 11 2 8 16 3 24 42 68 14 3 1 15 16 71 59 63