Estruturas de Dados
Heaps Binários

Universidade Estadual Vale do Acaraú – UVA

Paulo Regis Menezes Sousa

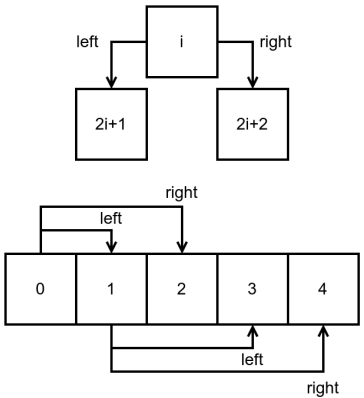
paulo_regis@uvanet.br

A estrutura heap

Filas de prioridades

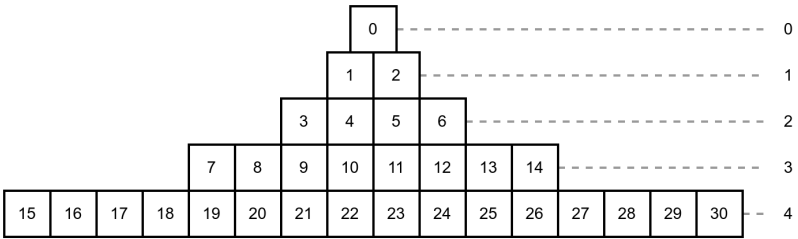
- A estrutura de dados *binary heap* foi inventada em 1964 por John William Joseph Williams.
- A estrutura está no coração do algoritmo **Heapsort** e é muito útil na a construção de filas de prioridades.

- Suponha dado um vetor $A[0 \ldots n-1]$. Onde n é o tamanho do vetor.
- Para todo índice i, diremos que:
  - o pai do índice i é o índice $\lfloor (i-1)/2 \rfloor$,
  - $2i+1$ é o filho esquerdo de i,
  - $2i+2$ é o filho direito de i.
- O índice 0 não tem pai;
- Um índice i só tem filho esquerdo se $2i+1 \leq n-1$;
- i só tem filho direito se $2i+2 \leq n-1$.



Árvore binária armazenada em um vetor

- Cada caixa abaixo é um nó da árvore binária quase completa $A[0 \ldots 30]$.
- O número dentro de cada caixa é i e não $A[i]$.



- Cada nível p, exceto talvez o último, tem exatamente 2^p nós.

- O nó i pertence ao nível

$$\lfloor \log i \rfloor$$

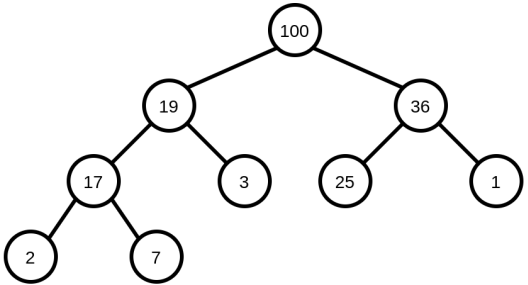
- Como o conceito de altura é complementar ao conceito de nível. A altura de um nó i em $A[0 \ldots n-1]$ é o número

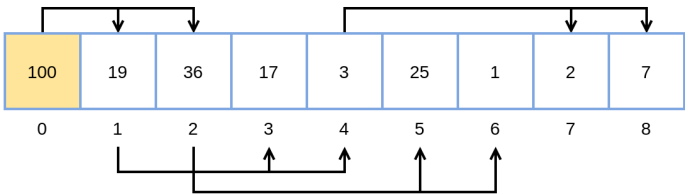
$$\lfloor \log((n-1)/i) \rfloor$$

- Em termos um tanto vagos, um max-heap (ou árvore hierárquica) é uma árvore binária quase completa em que cada pai é maior ou igual que qualquer de seus filhos.
- Um vetor $A[0 \ldots n-1]$ é um max-heap se:

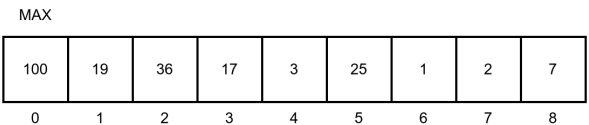
$$A\left[\left\lfloor \frac{(i-1)}{2} \right\rfloor\right] \geq A[i]$$

  para $i = 1, \ldots, n-1$.
- Em outras palavras, $A[0 \ldots n-1]$ é um max-heap se $A[j] \geq A[2j+1]$ e $A[j] \geq A[2j+2]$ sempre que os índices não ultrapassam $n-1$.


100
19
36
17
3
25
1
2
7


100
19
36
17
3
25
1
2
7
0
1
2
3
4
5
6
7
8

- Por que um heap é uma estrutura de dados útil?
  - Se $A[0 \ldots n-1]$ é um max-heap, é muito fácil encontrar um elemento máximo do vetor: $A[0]$ é o máximo.



  - Se $A[0 \ldots n-1]$ é um max-heap e se o valor de $A[0]$ for alterado, o max-heap pode ser "consertado" muito rapidamente.
  - Por consequência um vetor $A[0 \ldots n-1]$ arbitrário pode ser transformado em um max-heap rapidamente.

- A principal ferramenta de manipulação de um heap é o algoritmo que discutiremos a seguir. O algoritmo resolve o seguinte pequeno problema:

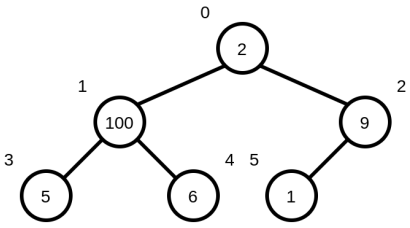
Problema

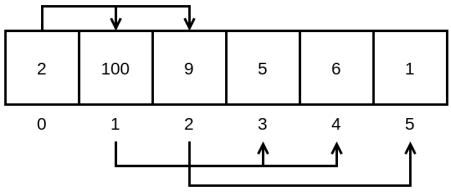
Dado um vetor $A[0 \ldots n-1]$ e um índice i tal que a subárvore com raiz $2i+1$ é um max-heap e a subárvore com raiz $2i+2$ é um max-heap, rearranjar o vetor de modo que a subárvore com raiz i seja um max-heap.

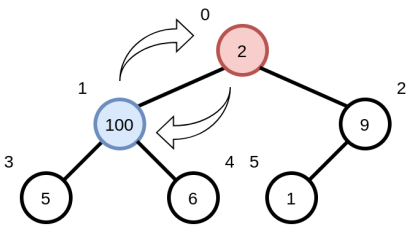
- A ideia do algoritmo é simples:
  1. se $A[i]$ é maior ou igual que seus filhos então não é preciso fazer nada;
  2. senão, troque $A[i]$ com o maior dos filhos e repita o processo para o filho envolvido na troca.

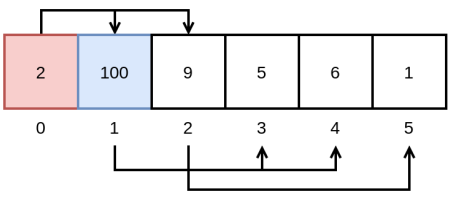
- Em inglês, o algoritmo é conhecido como HEAPIFY, ou SIEVE, ou FIX-DOWN, ou SHAKE-DOWN.

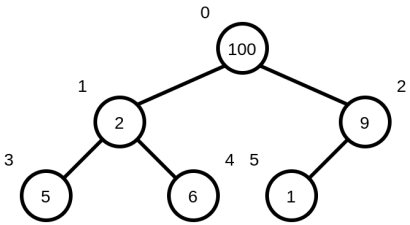
```
1  corrige_descendo (A, n, i)
2       j ← i
3       enquanto (2j+1 < n) faça
4            f ← 2j+1
5            se (f < n-2) e (A[f] < A[f+1]) então
6                 f ← f + 1
7            se (A[j] ≥ A[f]) então
8                 j ← n-1
9            senão
10                troque A[j] ↔ A[f]
11                j ← f
```

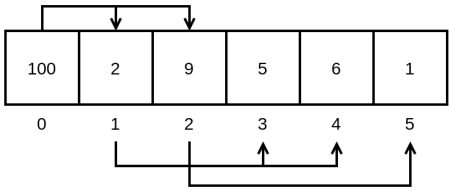

0
2
1
100
2
9
3
5
6
4
5
1

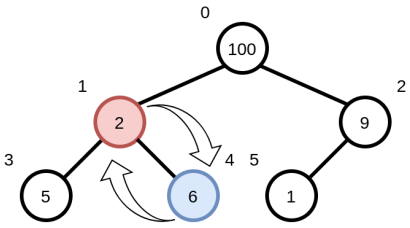

2
100
9
5
6
1
0
1
2
3
4
5

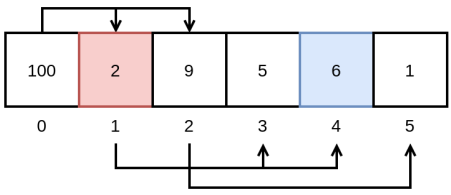

0
2
1
100
2
9
3
5
6
4
5
1

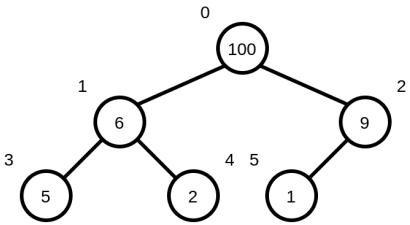

2
100
9
5
6
1
0
1
2
3
4
5

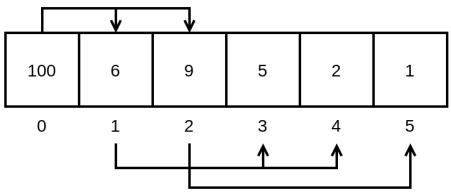

0
100
1
2
9
3
5
6
4
5
1


100
2
9
5
6
1
0
1
2
3
4
5


0
100
1
2
2
9
3
5
4
6
5
1

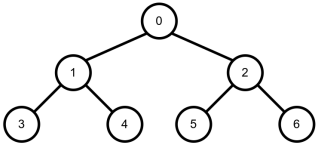

100
2
9
5
6
1
0
1
2
3
4
5


0
100
1
6
9
2
3
5
2
4
5
1


100
6
9
5
2
1
0
1
2
3
4
5

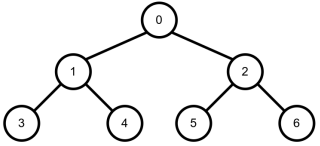
- O algoritmo abaixo recebe um vetor $A[0 \ldots n-1]$ e rearranja seus elementos de modo a transformar o vetor em um max-heap.

```
1  constroi_maxheap (A, n)
2       para i ← ⌊(n − 1)/2⌋ decrescendo até 0 faça
3       corrige_descendo(A, n, i)
```



- O seguinte algoritmo supõe que $A[0 \ldots m-1]$ é um max-heap e rearranja $A[0 \ldots m]$ de modo a transformar o vetor em um max-heap.

```
1  corrige_subindo (A, m)
2       i ← m
3       enquanto (i ≥ 0 e A[⌊(i-1)/2⌋] < A[i]) faça
4            troque A[⌊(i-1)/2⌋] ↔ A[i]
5            i ← ⌊(i-1)/2⌋
```



- Imagine um conjunto S de números. Os elementos de S são às vezes chamados chaves ou prioridades.
- Uma fila de prioridades é um tipo abstrato de dados que permite executar as seguintes operações sobre S:
  - encontrar um elemento máximo de S,
  - extrair um elemento máximo de S,
  - inserir um novo número em S,
  - aumentar o valor de um elemento de S,
  - diminuir o valor de um elemento de S.
- Há uma variante dessa definição em que "máximo" é substituído por "mínimo".
  - A primeira é uma fila de prioridades decrescente (ou "de máximo")
  - e a segunda é uma fila de prioridades crescente (ou "de mínimo").

- Não é difícil imaginar maneiras de implementar uma fila de prioridades.
- Infelizmente, nas implementações mais óbvias, alguma das operações fica rápida mas as outras ficam lentas.
- O desafio está em inventar uma implementação em que todas as operações sejam rápidas.

- Uma maneira muito eficiente de implementar uma fila de prioridades decrescente consiste em manter o conjunto S de chaves em um max-heap.

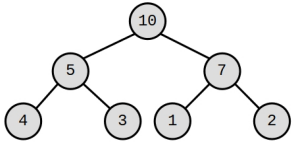
- **Máximo**
  - Eis o algoritmo que encontra (e devolve) o valor de um elemento máximo do max-heap $A[0 \ldots n-1]$ não vazio (ou seja, com $n \geq 1$):

```
1  encontra_maximo (A, n)
2      retorne A[0]
```

Remoção de máximo

- O seguinte algoritmo remove um elemento máximo do max-heap não vazio $A[0 \ldots n-1]$ e devolve o valor desse elemento:

```
1  remove_maximo (A, n)
2      max ← A[0]
3      A[0] ← A[n-1]
4      n ← n-1
5      corrige_descendo(A, n, 0)
6      retorne max
```



Inserção

- O algoritmo abaixo insere um número c no max-heap $A[0 \ldots n-1]$ (ou seja, acrescenta um novo elemento, com valor c, ao vetor) e rearranja o vetor para que ele volte a ser um max-heap:

```
1  inserir_na_fila (A, n, c)
2      A[n] ← c
3      n ← n+1
4      corrige_subindo(A, n-1)
```

Aumento do valor de uma chave

- O algoritmo seguinte recebe um max-heap não vazio $A[0 \ldots n-1]$, um índice i no intervalo $0 \ldots n-1$ e um número $c \geq A[i]$.
- O algoritmo altera para c o valor de $A[i]$ e rearranja o vetor para que ele volte a ser um max-heap:

```
1  aumenta_chave (A, i, c)
2      A[i] ← c
3      corrige_subindo(A, i)
```

Redução do valor de uma chave

- O algoritmo abaixo recebe um max-heap não vazio $A[0 \ldots n-1]$, um índice i no intervalo $0 \ldots n-1$ e um número $c \leq A[i]$.
- O algoritmo altera para c o valor de $A[i]$ e rearranja o vetor para que ele volte a ser um max-heap:

```
1  diminui_chave (A, n, i, c)
2      A[i] ← c
3      corrige_descendo(A, n, i)
```

- Os algoritmos anteriores são todos muito rápidos: no pior caso, cada um consome tempo proporcional à altura do heap.
- Como a altura é $\lfloor \log n \rfloor$, o consumo de tempo está em $O(\log n)$ no pior caso.

Problema

Implemente todos os algoritmos da aula e crie um programa de testes para demonstrar sua implementação.