

## Algoritmos e Estruturas de Dados 2 (AED2)

### Filas de prioridade (heap) e ordenação por seleção eficiente (heapSort)

Filas de prioridade são um tipo abstrato de dados

- em que cada elemento está associado a um valor,
  - que indica sua prioridade,
- e que generaliza tanto filas quanto pilhas.

Uma fila de prioridades suporta operações de:

- inserção de um elemento com um certo valor de prioridade,
- edição da prioridade de um elemento (operação menos comum),
- remoção do elemento com maior (ou menor) prioridade.
  - Esta operação não atende maior e menor simultaneamente.
    - Por isso temos filas de prioridade de máximo
      - e filas de prioridade de mínimo.

Quiz1: Conhecendo a definição e operações suportadas por uma fila de prioridade,

- como definir as prioridades dos elementos,
  - para que uma fila de prioridade se comporte como uma fila?
- E para que ela se comporte como uma pilha?

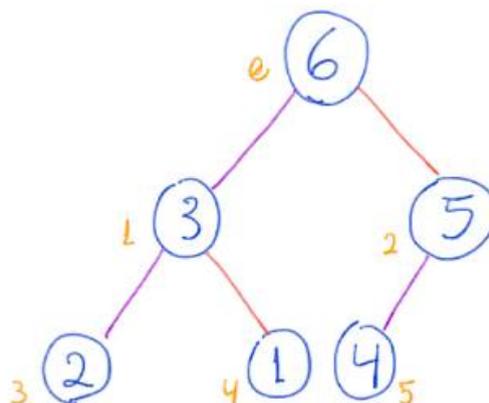
Para simplificar, ao longo desta aula vamos supor que

- o valor de cada elemento corresponde a sua prioridade,
- e vamos focar na versão de **máximo** da fila de prioridade.
  - Destacamos que é simples transformar a implementação
    - de uma fila de prioridade de máximo
      - em uma fila de prioridade de mínimo.
- Quiz2: Como usar uma fila de prioridade de máximo para
  - obter o comportamento de uma fila de mínimo?

### Implementação de uma fila de prioridade usando Heap

Heap é uma estrutura de dados eficiente para implementar Filas de Prioridades.

- Exemplo de um heap de máximo:

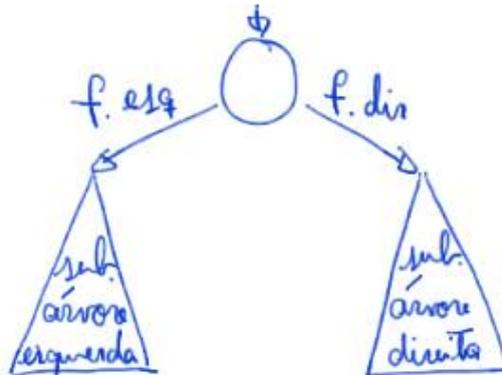


Um heap de máximo é uma árvore binária completa ou quase-completa,

- cujos nós respeitam a propriedade do heap de máximo.

Definição de uma árvore binária:

- Temos a propriedade recursiva, segundo a qual toda árvore binária
  - é um elemento com uma subárvore esquerda e uma subárvore direita



- ou é uma árvore vazia.
- Adicionamos à propriedade recursiva que
  1. cada elemento de uma árvore tem no máximo um pai,
    - a. sendo que o único elemento sem pai é a raiz.
    2. os filhos esquerdo e direito de cada elemento são distintos.
- A propriedade recursiva vai nos ajudar a pensar nas operações.

Numa árvore binária completa

- cada nível  $p$  tem  $2^p$  nós.
- Lembrando que a raiz fica no nível 0
  - e que o nível aumenta cada vez que
    - vamos de um nó para seu filho esquerdo ou direito.

Numa árvore binária quase completa

- cada nível  $p$  tem  $2^p$  nós,
  - com a possível exceção do último nível.
- Se for esse o caso, no último nível as posições dos nós
  - são preenchidas da esquerda para a direita, sem espaços vazios.

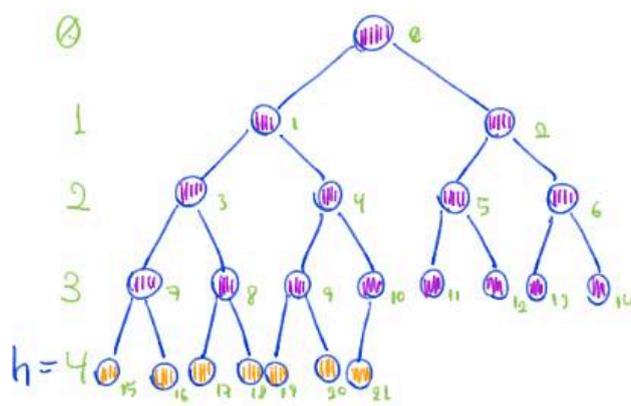
Definimos a **altura** de um nó  $x$  como sendo o comprimento

- do maior caminho de  $x$  até uma folha de sua subárvore,
  - i.e., o número de saltos entre nós em tal caminho.
- A altura ( $h$ ) de uma árvore é a altura do nó raiz da mesma.

Queremos descobrir a altura ( $h$ ) de uma árvore binária quase completa com  $m$  nós,

- ou, de modo equivalente,
  - o número de níveis que um heap com  $m$  elementos possui,
- pois isso será relevante para entender
  - a eficiência de operações que manipulam um heap.

Para tanto, vamos considerar algumas questões:



$$\# \text{ nós nível } h \leq 2^h = 2^4$$

$$\# \text{ nós total} \leq 2^{h+1} - 1 = 2^5 - 1$$

$$\text{índice do último nó do nível } h \leq \# \text{ nós total} - 1$$

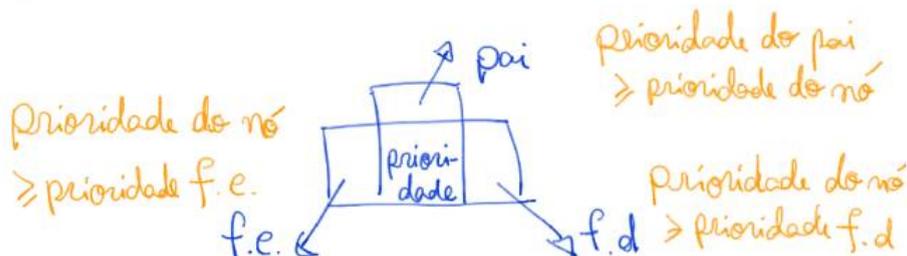
$$\text{índice do primeiro nó do nível } h \geq \# \text{ nós total} - \# \text{ nós nível } h$$

Lembre que o índice do último nó da árvore/heap é  $m-1$

- Quantos nós cabem no nível  $h$  de uma árvore binária completa?
  - Já vimos que são  $2^h$  nós.
- Quantos nós cabem numa árvore binária completa com último nível (ou altura)  $h$ ?
  - Também já vimos que são  $2^{(h+1)} - 1$  nós.
    - Então, o número de nós  $m \leq 2^{(h+1)} - 1$ .
- E quantos nós cabem nos primeiros  $h$  níveis dessa árvore,
  - i.e., nos níveis entre 0 e  $h-1$ ?
  - São  $2^{(h-1+1)} - 1 = 2^h - 1$  nós.
    - Assim, o número de nós  $m \geq (2^h - 1) + 1 = 2^h$ .
- Portanto,
  - $2^h \leq m \leq 2^{(h+1)} - 1$
  - $2^h \leq m < 2^{(h+1)}$
  - $h \leq \lg(m) < h+1$
- Ou seja, o maior nível de um heap com  $m$  elementos é **piso( $\lg m$ )**.

A propriedade do heap de máximo diz que

- o valor da prioridade de um nó é  $\geq$  que a prioridade de seus filhos.



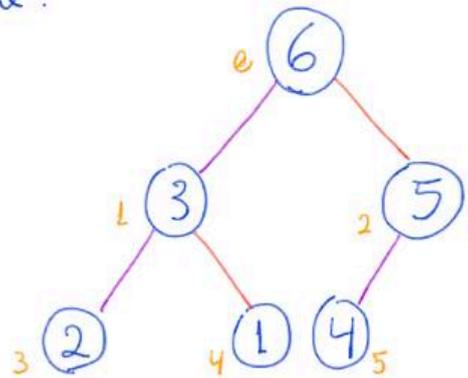
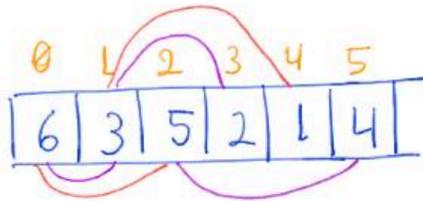
- Note que a propriedade do heap não nos permite comparar
  - os valores de um filho esquerdo e de um filho direito.

O fato do heap ser uma árvore binária quase completa,

- permite que ele seja implementado em um vetor,
  - como mostra o seguinte exemplo:

- Exemplo de heap na visão de:

- árvore binária
- linearizado em vetor



- Em tal implementação o vetor é preenchido da esquerda para a direita,
  - e os nós da árvore são contados/numerados de cima para baixo
    - e, em cada nível, também da esquerda para a direita.
- Desse modo, o número associado a cada nó da árvore
  - corresponde a seu índice no vetor.

De modo geral, implementamos um heap com  $m$  elementos

- em um vetor  $v$  que começa em 0 e vai até  $m - 1$ .
- Para tanto, dado um elemento na posição  $i$ ,
  - é essencial saber quem é pai, filho esquerdo e filho direito de  $i$ .
- Para tanto, podemos usar as seguintes fórmulas:

```
#define FILHO_ESQ(i) (2 * i + 1)
```

```
#define FILHO_DIR(i) (2 * i + 2)
```

```
#define PAI(i) ((i - 1) / 2)
```

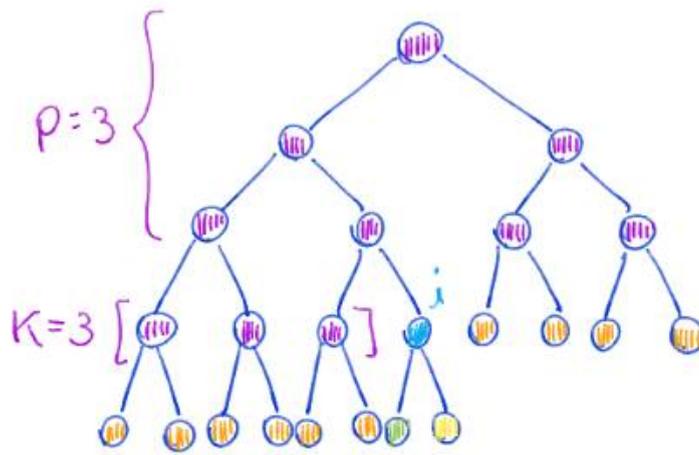
- Verifique que elas funcionam corretamente no heap do exemplo anterior.

Observe que, o nó raiz, que não tem pai, fica na posição 0.

- Além disso, se  $FILHO\_ESQ(i)$  ou  $FILHO\_DIR(i)$  forem  $\geq m$ ,
  - então  $i$  não tem filho esquerdo ou direito, respectivamente.
- Note que os nós da segunda metade do vetor não tem filhos, já que
  - para  $i \geq m / 2$  temos  $FILHO\_ESQ(i) = 2 * i + 1 \geq 2 * m / 2 + 1 \geq m$ .
- De fato, em um heap (e em toda árvore binária quase completa),
  - o número de folhas (nós sem filhos) é pelo menos metade do total.

Para obter uma intuição do porque o índice de um nó  $i$

- é aproximadamente metade do índice de seus filhos, observe que,
  - numa árvore binária quase completa o número de nós antes de  $i$  é
    - aproximadamente igual ao número de nós entre  $i$  e seus filhos.
- Para uma explicação mais precisa, considere a análise do seguinte exemplo:



elementos antes de  $i$

$$10 = 7 + 3 = (2^3 - 1) + 3 = (2^p - 1) + k$$

$$i = (2^p - 1) + k \text{ (índices começam em 0)}$$

elementos depois de  $i$

$$10 = 4 + 6$$

$$= (2^3 - 3 - 1) + 2 \cdot 3$$

$$= (2^p - k - 1) + 2k = 2^p + k - 1$$

$$f(i) = (2^p + k - 1) + 1 + (2^p + k - 1)$$

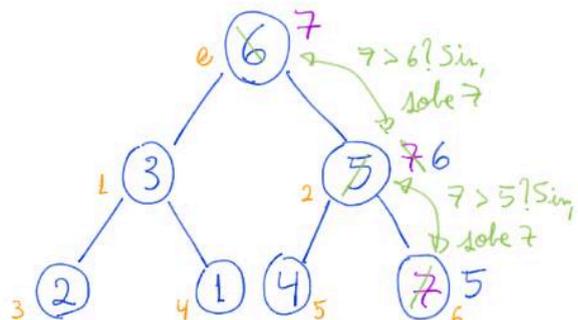
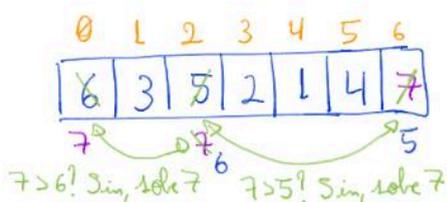
$$= 2i + 1$$

Agora vamos estudar as duas funções mais importantes para manutenção do heap.

### Sobe Heap

- Veremos esta função aplicada à inserção de um novo elemento no heap,
  - que é seu uso mais comum.
- Também a utilizaremos para construir um heap a partir de um vetor.

- Exemplo de inserção no heap usando função sobe heap



Código da `sobeHeap`:

```
// sobe o elemento em v[pos] até restaurar a propriedade do heap
void sobeHeap(int v[], int pos) {
    int corr = pos;
    while (corr > 0 && v[PAI(corr)] < v[corr]) {
        troca(&v[corr], &v[PAI(corr)]);
        corr = PAI(corr);
    }
}
```

- Exemplo de uso da `sobeHeap`:

```
printf("Testando sobeHeap com elemento da ultima posicao\n");
sobeHeap(v, m - 1);
```

Corretude e invariante da `sobeHeap`:

- O invariante principal que vale no início de cada iteração é
  - todo elemento em  $v[0 .. pos]$  respeita a propriedade do heap,
    - exceto, possivelmente, pelo elemento `corr`.
  - Isto é,  $v[i] \leq v[PAI(i)] = v[(i - 1) / 2]$  vale para todo  $i \neq corr$ .

Eficiência da `sobeHeap`: número de operações é  $O(\log pos) = O(\log m)$ ,

- pois no início `corr = pos` e em cada iteração `corr` é dividido por 2.

A seguir apresentamos o código da função `insereHeap`, que

- implementa uma das operações fundamentais da fila de prioridade.
- Esta função coloca o novo elemento na próxima posição disponível no vetor,
  - e invoca `sobeHeap` para restabelecer a propriedade do Heap.
    - Por isso, sua eficiência é  $O(\lg m)$ .

```
int insereHeap(int v[], int m, int x) {
    v[m] = x;
    sobeHeap(v, m);
    return m + 1;
}
```

- Exemplos de uso da `insereHeap`:

```
printf("Inserindo novo elemento no max heap\n");
m = insereHeap(v, m, 999);

printf("Criando novo max heap usando insereHeap - ordem
direta\n");
m = 0;
for (i = 0; i < n; i++)
    m = insereHeap(v, m, i);
```

```

printf("Criando novo max heap usando insereHeap - ordem
inversa\n");
m = 0;
for (i = 0; i < n; i++)
    m = insereHeap(v, m, n - i - 1);

```

Uso da `sobeHeap` para reorganizar um vetor transformando-o em um Heap:

```

printf("Criando um max heap mandando todos subirem da esquerda
pra direita\n");
for (i = 1; i < m; i++)
    sobeHeap(v, i);

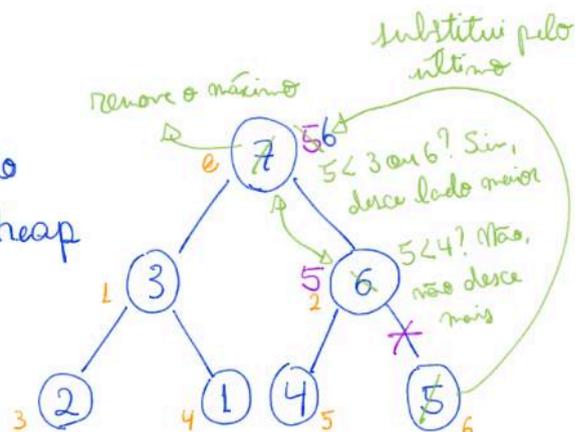
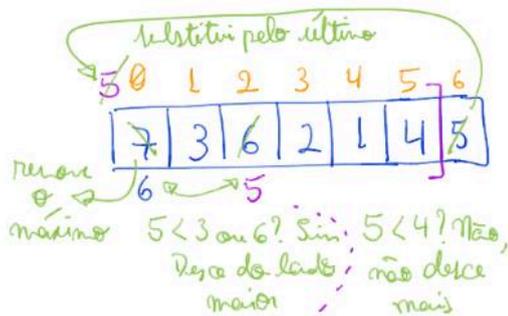
```

- Quiz3: Qual a eficiência deste algoritmo?
  - $O(m \lg m)$ , pois invoca `sobeHeap`  $m$  vezes.

### Desce Heap

- Veremos esta função aplicada à remoção de um elemento do heap,
  - que é seu uso mais comum.
- Também a utilizaremos para construir um heap a partir de um vetor.

- Exemplo de remoção do máximo de um heap seguida de restauração do heap usando a função `desce heap`



Código da `desceHeap`:

```

// desce o elemento em v[pos] até restaurar a propriedade do heap
void desceHeap(int v[], int m, int pos) {
    int corr = pos, filho;
    while (FILHO_ESQ(corr) < m && (v[FILHO_ESQ(corr)] > v[corr]
        || (FILHO_DIR(corr) < m && v[FILHO_DIR(corr)] > v[corr])))
    {
        filho = FILHO_ESQ(corr);
        if (FILHO_DIR(corr) < m && v[FILHO_DIR(corr)] > v[filho])
            filho = FILHO_DIR(corr);
        troca(&v[corr], &v[filho]);
        corr = filho;
    }
}

```

```
}  
}
```

- Exemplo de uso da `desceHeap`:

```
printf("Testando desceHeap com elemento da primeira posicao\n");  
v[0] = 0;  
desceHeap(v, m, 0);
```

Corretude e invariante da `desceHeap`:

- O invariante principal que vale no início de cada iteração é
  - todo elemento em  $v[0 .. m - 1]$  respeita a propriedade do heap,
    - exceto, possivelmente, pelo elemento `corr`.
  - Isto é,  $v[i] \geq v[\text{FILHO\_ESQ}(i)] = v[2 * i + 1]$ 
    - e  $v[i] \geq v[\text{FILHO\_DIR}(i)] = v[2 * i + 2]$  vale para todo  $i \neq \text{corr}$ .

Eficiência da `desceHeap`: número de operações é  $O(\lg m)$ ,

- pois em cada iteração descemos um nível na árvore do heap
  - e o maior nível é  $\text{piso}(\lg m)$ .

A seguir apresentamos o código da função **`removeHeap`**, que

- implementa uma das operações fundamentais da fila de prioridade.
- Esta função remove e devolve o elemento máximo,
  - que está na primeira posição do vetor.
- Para ocupar essa posição vaga,
  - ela move o último elemento do vetor para a primeira posição.
- Então, ela invoca `desceHeap` para restabelecer a propriedade do Heap.
  - Por isso, sua eficiência é  $O(\lg m)$ .

```
int removeHeap(int v[], int m, int *px) {  
    *px = v[0];  
    troca(&v[0], &v[m - 1]);  
    desceHeap(v, m, 0);  
    return m - 1;  
}
```

- Exemplo de uso do `removeHeap`:

```
printf("Removendo o maior elemento do vetor usando  
removeHeap\n");  
m = removeHeap(v, m, &x);
```

Curiosidade: É possível usar a função `desceHeap` para reorganizar um vetor

- de modo a transformá-lo em um heap.
- De fato, essa implementação é particularmente eficiente.

Heaps são estruturas de dados muito eficientes

- para implementar filas de prioridade.
- Por isso, sempre que estiverem resolvendo um problema
  - e perceberem que seu algoritmo busca repetidamente
    - pelo maior (ou menor) elemento de um conjunto,
  - devem desconfiar que um heap tornará seu algoritmo mais rápido.

Quiz4: Como alterar os códigos para transformar

- um heap de máximo em um heap de mínimo?

Quiz5: Considere que uma operação de edição alterou a prioridade

- de um elemento  $i$  de um Heap, alocado em um vetor  $v[0 \dots m - 1]$ .
- Podemos usar as funções `sobeHeap` e `desceHeap`
  - para restaurar a propriedade do Heap.
- Supondo que a edição reduziu a prioridade de  $v[i]$ ,
  - qual função usar e quais os parâmetros da chamada?
- Responda a mesma questão,
  - no caso da edição ter aumentado a prioridade de  $v[i]$ .

## Ordenação por seleção eficiente (heapSort)

Na maioria das aplicações do Heap,

- percebemos que ele pode ajudar a melhorar um algoritmo,
  - quando o mesmo realiza sucessivas requisições
    - pelo elemento máximo (ou mínimo) de um conjunto.
- Isto acontece em inúmeras situações,
  - como quando temos que decidir o próximo evento a ocorrer,
    - sendo cada evento associado a uma importância ou tempo.

A aplicação do heap que veremos envolve o problema da ordenação,

- no qual queremos colocar em ordem crescente um vetor  $v$  de tamanho  $n$ .

Começamos lembrando a ideia do `selectionSort`,

- que percorre o vetor da esquerda para a direita
  - e em cada iteração busca o menor elemento do sufixo do vetor
    - colocando este na posição corrente.

Como este algoritmo realiza sucessivas buscas pelo menor elemento,

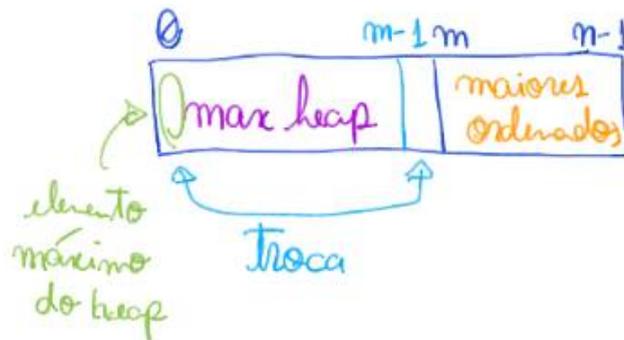
- é um candidato natural a ser melhorado usando um Heap.

Da união da ideia do `selectionSort` com a estrutura de dados heap

- surge o algoritmo `heapSort`, cuja ideia é:
  - Colocar os elementos do vetor em um heap,
  - em cada iteração extrair um elemento do heap,
  - e colocá-lo na posição correta no vetor ordenado.
- Como cada extração do heap leva tempo  $O(\log n)$ ,
  - e são necessárias  $O(n)$  extrações,
- esse algoritmo deve levar tempo  $O(n \log n)$  para ordenar o vetor.

Entrando um pouco mais nos detalhes técnicos desse algoritmo,

- primeiro re-organizamos os elementos do vetor
  - de modo a construir um heap de máximo.
- Então, em cada iteração, extraímos o maior elemento do heap
  - e o colocamos na última posição do vetor corrente.



- Quiz6: Por que usamos um heap de máximo, e não de mínimo?

Código do heapSort1:

```
void heapSort1(int v[], int n) {
    int i, m;
    for (i = 1; i < n; i++) // construindo o heap em tempo O(n lg n)
        sobeHeap(v, i);
    for (m = n; m > 0; m--) {
        troca(&v[0], &v[m - 1]); // colocando o máximo no final
        desceHeap(v, m - 1, 0); // restaurando o Heap
    }
}
```

- Exemplo de uso do heapSort1:

```
printf("Ordenando com heapSort1\n");
heapSort1(v, n);
```

Corretude e invariante do heapSort1:

- Os invariantes principais, que valem no início do segundo laço são
  - $v[0 .. n - 1]$  é uma permutação do vetor original,
  - $v[m .. n - 1]$  está ordenado em ordem crescente,
  - $v[0 .. m - 1]$  é um heap de máximo,
  - $v[0 .. m - 1] \leq v[m .. n - 1]$ .
- Note que esses invariantes implicam a ordenação do vetor na última iteração.

Eficiência de tempo do heapSort1:

- O algoritmo executa da ordem de  $n \lg n$  operações, i.e.,  $O(n \log n)$ ,
  - pois tanto o primeiro quanto o segundo laços executam  $O(n)$  vezes
- e em cada iteração invocam uma operação do heap que leva tempo  $O(\log n)$ .

Animação: Visualization and Comparison of Sorting Algorithms -

[www.youtube.com/watch?v=ZZuD6iUe3Pc](http://www.youtube.com/watch?v=ZZuD6iUe3Pc)