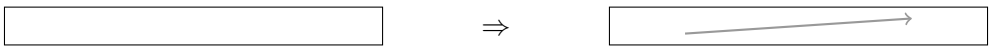
Estruturas de Dados

aula 22: Heaps

1 Introdução

Hoje nós vamos olhar para o problema da ordenação de uma lista um pouco mais de perto



Quer dizer, a ideia natural para fazer isso é:

- localizar o maior elemento, e colocar na última posição
- localizar o segundo maior elemento, e colocar na penúltima posição
- e por aí vai ...

```
func  Ord-Selecao ( L )
     Para  i = N  Até  2
        M = L[1];    posM = 1
        Para  j = 2  Até  i
           Se ( L[j] > M )
              M = L[j];   posM = j
        Troca ( L, i, j )
```

Mas, esse algoritmo é uma porcaria.

Porque?

Bom, porque ele executa em tempo $O(n^2)$.

Porque?

Bom, porque ele percorre a lista n vezes.

Sim, mas porque isso é uma porcaria?

Bom, porque isso acaba levando a gente a fazer a mesma coisa várias vezes.

Por exemplo, imagine que a lista começa assim

onde o 17 e o 30 não são os maiores elementos da lista.

Então, a cada vez que o algoritmo percorrer a lista, ele vai comparar esses dois elementos para descobrir que o 30 é maior que o 17.

E se a lista tiver tamanho 1.000.000, o algoritmo vai fazer essa comparação 1.000.000 de vezes.

Isso não faz sentido, não é?

Hoje nós vamos resolver isso — (*três vezes ...*)

2 Anotando as coisas

A ideia é bem natural.

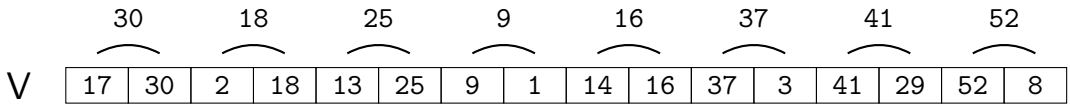
Quer dizer, basta o computador lembrar que já fez aquela comparação.

Mas, como é que o computador lembra de alguma coisa?

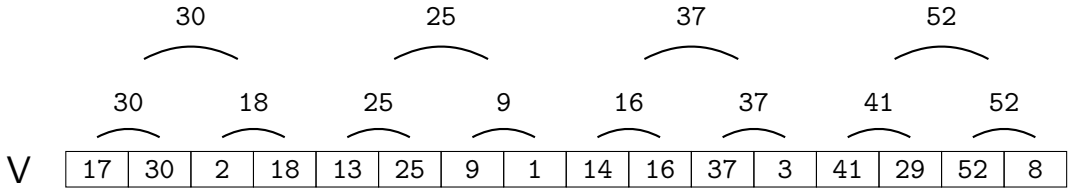
Bom, da mesma maneira que a gente: fazendo uma anotação em algum lugar.

Então a ideia é fazer a comparação, e anotar em algum lugar quem é o maior elemento.

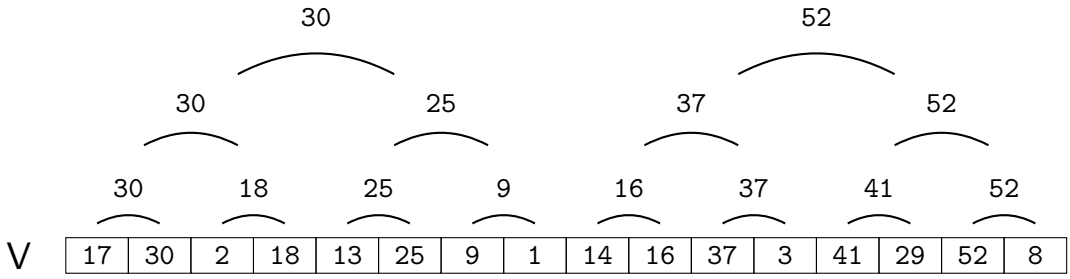
E as esperteza é fazer isso de dois em dois



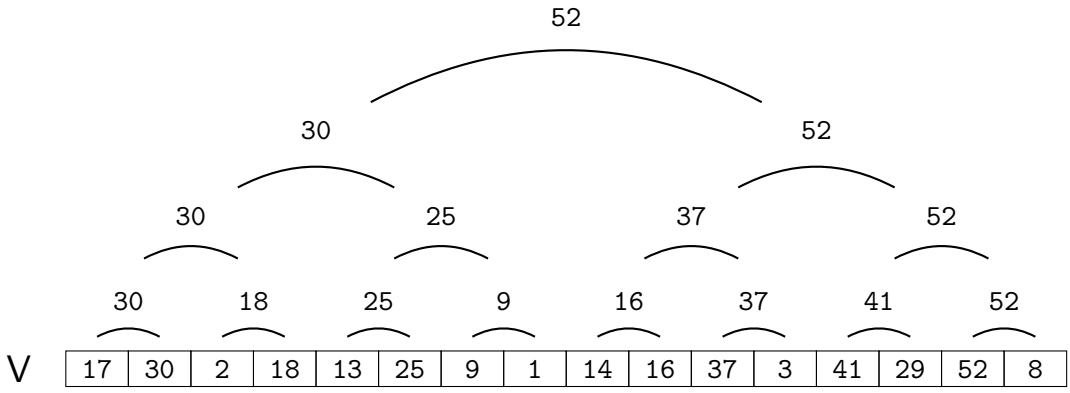
Agora, não é difícil pensar em comparar os maiores de cada par



Daí, a gente faz a mesma coisa outra vez



E faz a mesma coisa outra vez



Pronto!

Agora a gente sabe quem é o maior.

E ainda lembra de um monte de coisas.

Então agora, deve ser fácil encontrar o segundo maior.

Onde é que ele está?

Bom, se ele não está lá em cima, é porque ele encontrou alguém maior do que ele no meio do caminho.

Ora, esse alguém só pode ter sido o maior.

Logo, o segundo maior deve estar em algum lugar no caminho do maior.

E esse caminho não é muito comprido: ele tem $\log_2 n$ passos — (*porque?*)

Legal!

Aqui a gente já consegue ver a ideia funcionando.

Mas, nós ainda temos um pequeno problema.

Quer dizer, imagine agora que nós queremos encontrar o terceiro maior.

Onde é que ele está?

Bom, se ele não está lá em cima, é porque ele encontrou alguém maior do que ele no meio do caminho.

Só que dessa vez pode ser o maior ou o segundo maior.

Então, se a gente quiser encontrar o terceiro maior, é preciso descer pelo caminho do maior, e pelo caminho do segundo maior.

E se a gente quiser encontrar o quarto maior, é preciso descer pelo caminho do maior, pelo caminho do segundo maior, e pelo caminho do terceiro maior.

Pois é, a ideia já não está mais funcionando tão bem ...

2.1 Fazendo mais anotações

E agora?

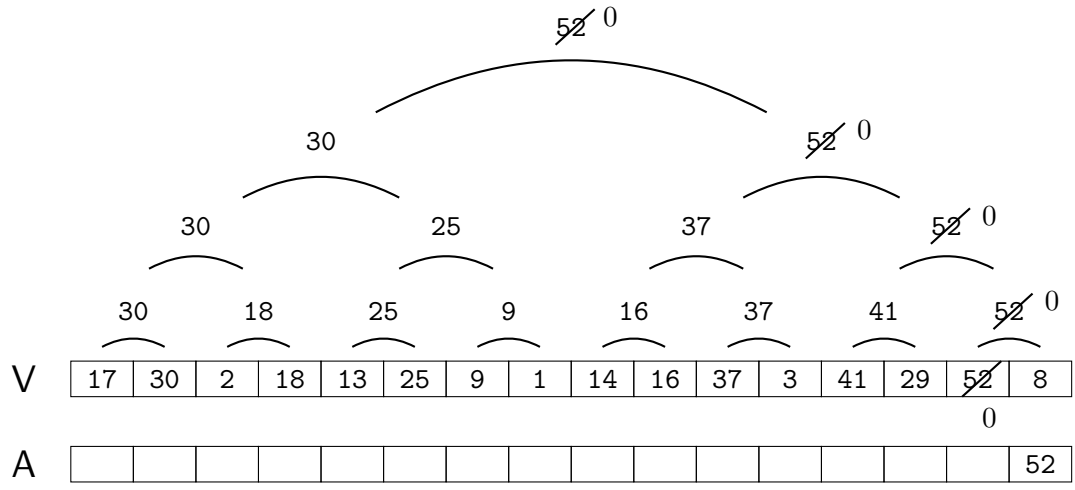
Bom, agora a gente usa a mesma ideia outra vez.

Quer dizer, depois que a gente descobre quem é o maior, a gente pode esquecer dele.

Contanto que ele fique anotado em algum lugar.

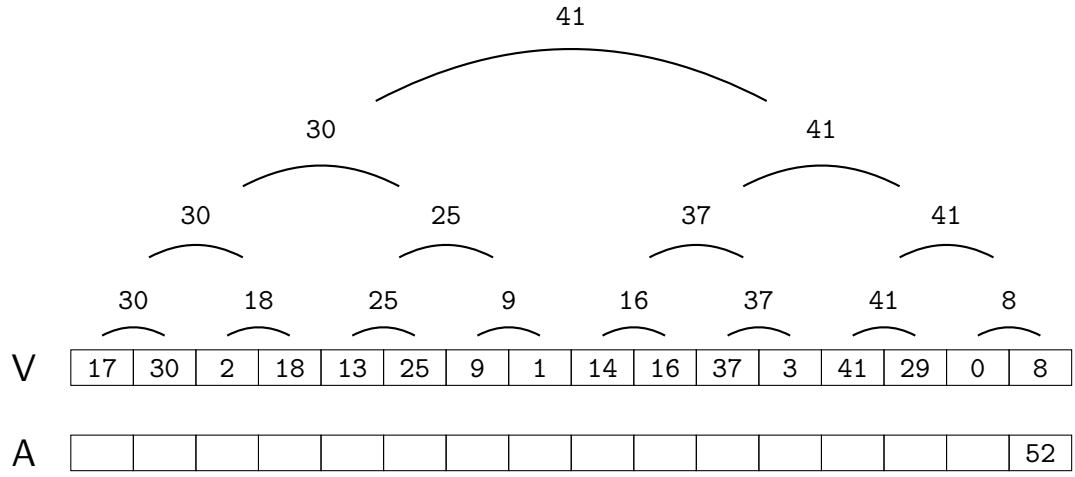
Então agora, a gente vai ter uma lista auxiliar — (*para guardar as anotações*)

E para esquecer o maior, a gente muda o seu valor para 0 — (*em todos os lugares onde ele aparece*)



Daí, como o zero é menor do que todo mundo, ele perde todas as comparações

E a coisa é reconstruída assim

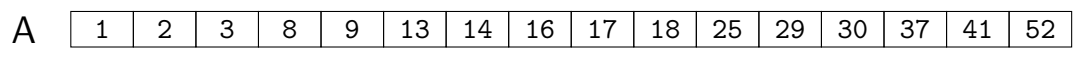


Pronto! agora nós temos a mesma coisa outra vez.

Quer dizer, o maior elemento está no topo — (*dentre aqueles que sobraram*)

E nós podemos fazer a mesma coisa para colocá-lo na lista auxiliar, e restaurar a situação no mesmo formato.

Daí que, fazendo isso repetidamente, a gente obtém a lista completamente ordenada.



Legal.

Análise

Mas, quanto tempo leva isso?

Bom, a coisa funciona em duas etapas

1. a construção da estrutura de árvore
2. a construção da lista auxiliar ordenada

E a gente pode analisar o tempo de cada etapa em separado.

A gente estima o tempo da primeira etapa contando o número de comparações que são realizadas ali

- primeiro, a gente compara os pares de elementos consecutivos

$$\frac{n}{2} \qquad \text{comparações}$$

- depois, a gente compara os ganhadores dessas comparações

$$\frac{n}{4} \qquad \text{comparações}$$

- depois, a gente compara os ganhadores dessas comparações

$$\frac{n}{8} \qquad \text{comparações}$$

- e por aí vai ...

Daí, a gente soma tudo para obter

$$\frac{n}{2} \quad + \quad \frac{n}{4} \quad + \quad \frac{n}{8} \quad + \quad \ldots \quad + \quad 2 \quad + \quad 1 \quad = \quad O(n)$$

Certo.

Agora a gente considera a segunda etapa.

Na segunda etapa, a gente

- copia o maior elemento para a lista auxiliar
- esquece o maior elemento — (*substituindo o seu valor por 0*)
- e refaz as comparações ao longo desse caminho

Daí, a gente nota que o caminho do topo até embaixo tem tamanho $\log_2 n$ — (*porque?*)

Logo, a coisa toda leva tempo

$$O(1) \quad + \quad O(\log n) \quad + \quad O(\log n) \quad = \quad O(\log n)$$

Mas, isso precisa ser feito uma vez para cada elemento.

Logo, a coisa toda leva tempo

$$O(n \log n)$$

Finalmente, é preciso somar os tempos das duas etapas

$$O(n) \quad + \quad O(n \log n) \quad = \quad O(n \log n)$$

Legal, não é?

3 Trocando as coisas de lugar

Sim, mas esse algoritmo faz anotações demais.

Quer dizer, se a lista tem tamanho 1.000.000, nós precisamos reservar espaço para 2.000.000 anotações.

Agora, a gente vai tentar fazer melhor do que isso.

E a ideia é usar um outro truque para lembrar daquilo que aconteceu.

O truque é: mudar as coisas de lugar.

Por exemplo, imagine que as coisas estão assim

Daí, a gente compara o 17 e o 30, descobre que o 30 é o maior, e troca os dois de lugar

Quer dizer, agora a gente sabe que

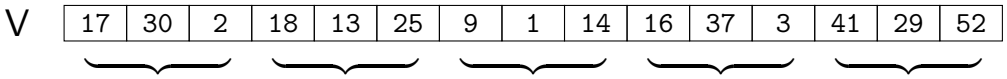
- o maior dos dois está na 1a posição
- o menor dos dois está na 2a posição

E não precisa fazer essa comparação outra vez.

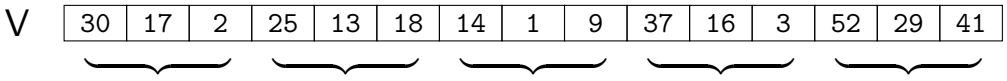
Certo.

Mas aqui a gente vai usar uma outra esperteza.

Quer dizer, a gente vai dividir os elementos em grupinhos de 3

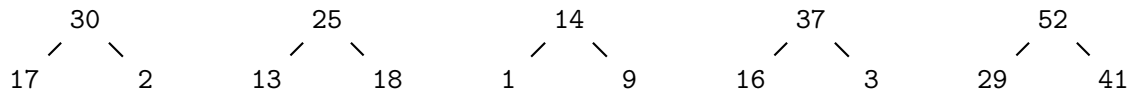


E vai colocar o maior de cada grupinho na primeira posição do grupinho



Porque?

Ora, porque assim a gente pode imaginar uma estrutura de árvore

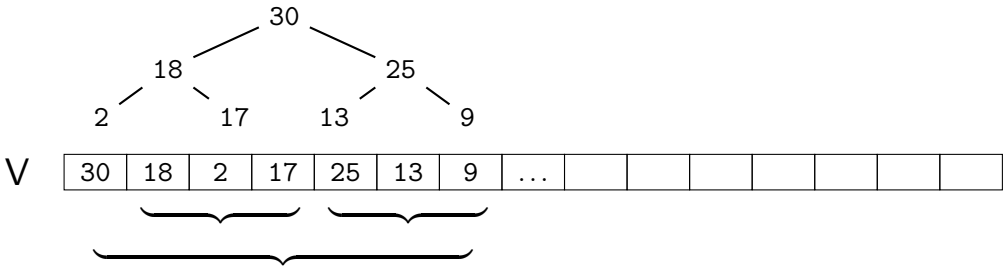


Mas note que isso é apenas a nossa imaginação.

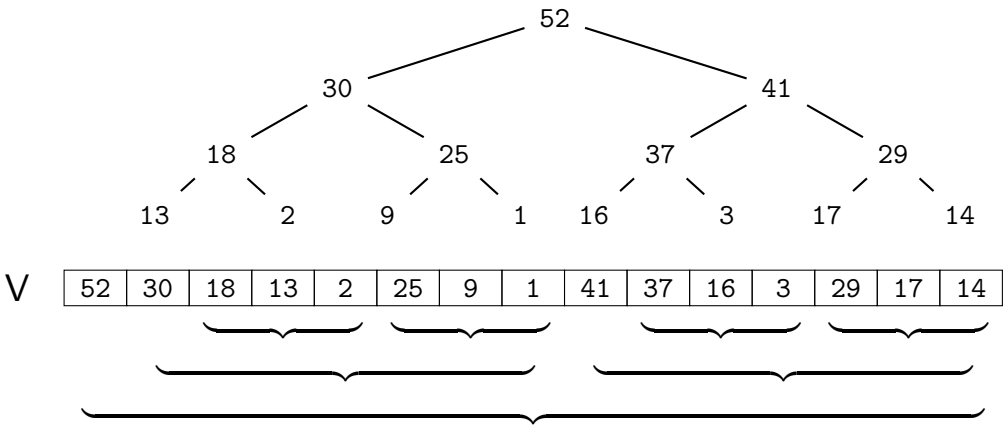
Porque os números continuam armazenados na lista, e nós não estamos fazendo nenhuma anotação.

Certo.

Daí, para aumentar o tamanho da árvore, a gente usa grupinhos de 7 — (*ou 1 mais dois grupinhos de 3*)



E para aumentar o tamanho da árvore outra vez, a gente usa grupinhos de 15 — (*ou 1 mais dois grupinhos de 7*)

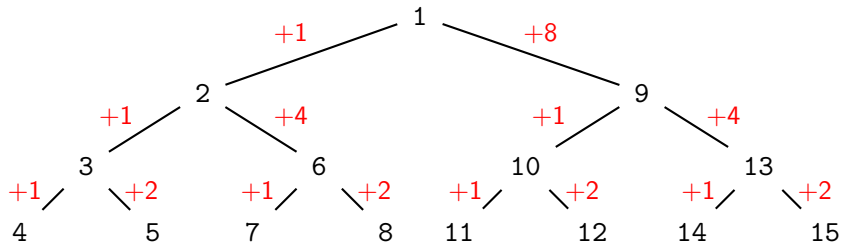


Mas, qual é a regra?

Bom, para descobrir isso, a gente olha para a árvore outra vez.

Só que troca os números pelas posições.

E daí, a gente vê o seguinte padrão



Legal.

Agora, a gente pode usar essa regra para subir e descer na árvore.

E com isso, a gente implementa o esquema de ordenação sem construir a árvore.

Ou melhor, construindo a árvore dentro da lista.

Legal, não é?

4 A bagunça organizada

Sim, mas esse esquema ainda tem dois problemas.

Quer dizer, a gente ainda está utilizando a lista auxliar para produzir a lista ordenada.

E o outro problema é que esse esquema ainda não é muito flexível.

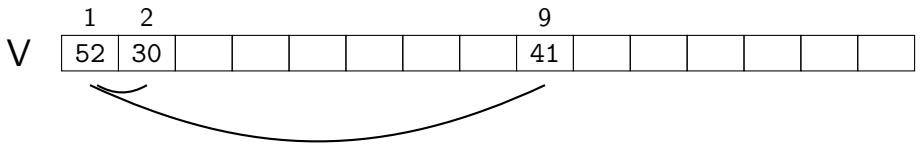
Quer dizer, ele funciona bem quando a lista tem tamanho $2^k - 1$ — (*porque?*)

Isso nem sempre isso é o caso.

Mas a gente sempre pode aumentar o tamanho da lista se for preciso.

O problema maior é o seguinte.

Considere o primeiro elemento (que fica no topo da árvore), e os dois elementos que ficam abaixo dele



O ponto é que o segundo elemento se encontra aproximadamente na metade da lista.

Só que o tamanho da lista pode mudar — (*se ocorrerem inserções ou remoções*)

Mas se o tamanho da lista mudar, nós perdemos a estrutura de árvore que foi construída.

Em resumo, nós construímos uma estrutura de dados dentro de uma estrutura de dados — (*uma árvore dentro de um vetor*)

Mas, a nossa estrutura de dados não suporta as operações de inserção e remoção.

O que significa que isso é uma estrutura de dados muito pouco flexível.

E agora?

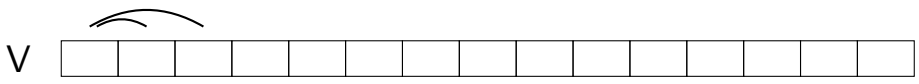
Bom, agora nós temos um problema.

E quando a gente tem um problema, a gente procura uma solução.

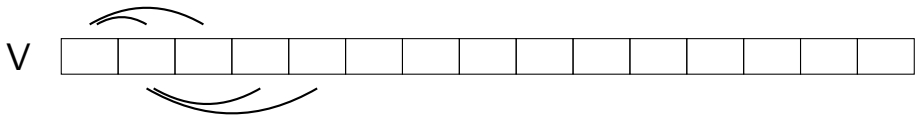
A boa notícia é que nesse caso a solução está ao alcance da mão.

Quer dizer,

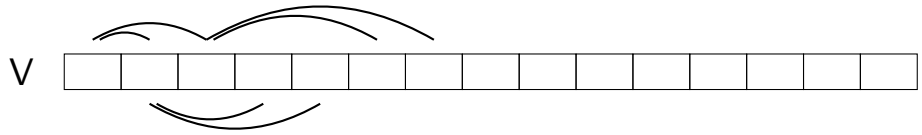
- abaixo do primeiro elemento, nós temos os dois próximos elementos da lista



- abaixo do segundo elemento, nós temos os dois elementos seguintes

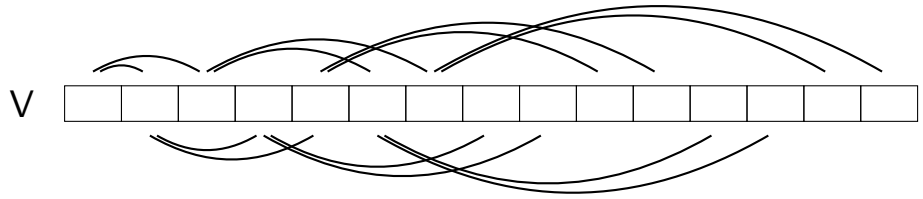


- abaixo do terceiro elemento, nós temos os dois elementos seguintes



- e por aí vai ...

Fazendo isso até o fim, a gente obtém o seguinte



Qual é a regra?

Bom, aqui é bem fácil de ver

→ *abaixo do elemento na posição* `k`

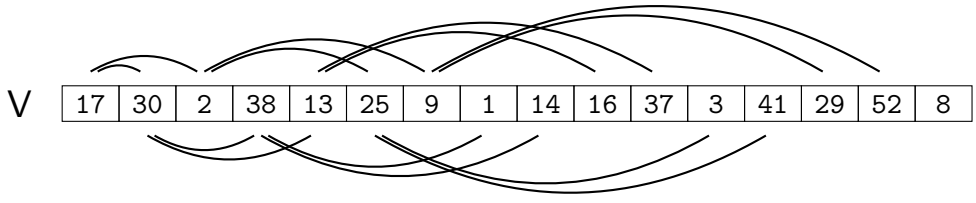
estão os elementos nas posições `2k` *e* `2k + 1`

Legal, não é?

4.1 Construção da árvore

Sim é legal, mas a gente ainda não acabou.

Quer dizer, quando tudo começa a lista está toda desorganizada

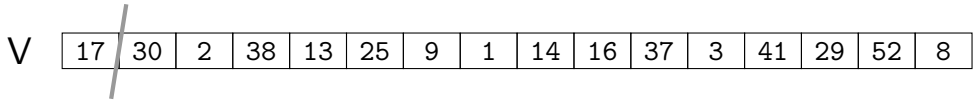


e os elementos não precisam ser maiores do que aqueles que estão abaixo deles na árvore.

E agora?

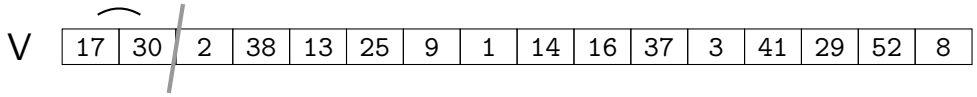
Bom, agora é a hora para uma esperteza.

E a esperteza consiste em imaginar que a lista tem tamanho 1 — (*ignorando os demais elementos*)



Daí, uma lista de tamanho 1 sempre está organizada — (*porque?*)

E o primeiro passo consiste em imaginar que o tamanho da lista vai aumentar para 2

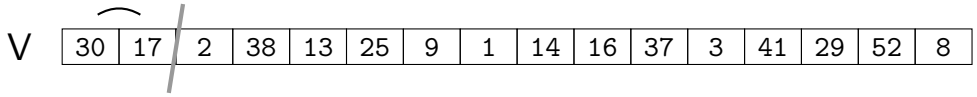


Quer dizer, isso aconteceu porque o 30 entrou na lista.

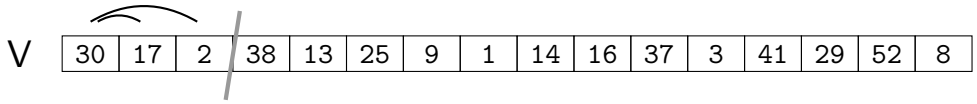
E daí, ele precisa ser menor do que o elemento que está acima dele.

Só que ele não é.

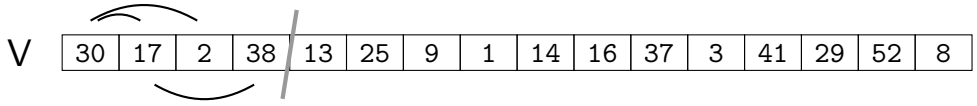
Então, a gente troca os dois de lugar



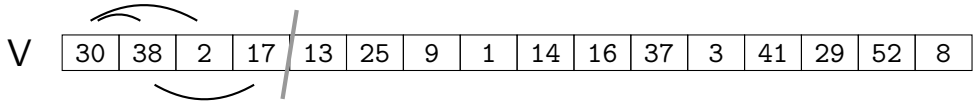
A seguir, quem entra é o 2, e dessa vez não acontece nada



Daí, é a vez do 38

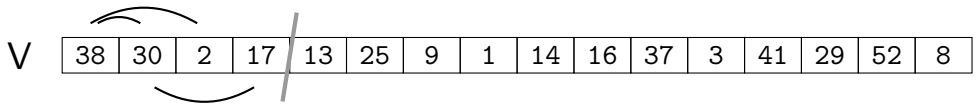


Como o 38 é maior do que o elemento acima dele, os dois trocam de lugar



Só que após a troca, o 38 continua sendo maior do que o elemento acima dele.

Então, ocorre uma nova troca



Quer dizer, um novo elemento que chega na lista pode chegar até o topo — (*por meio de trocas sucessivas*)

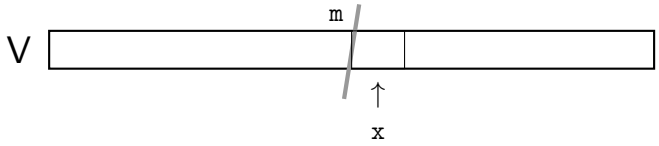
E a construção da árvore pode ser feita por meio de inserções sucessivas.

Certo.

Agora que a gente entendeu, está na hora de escrever o algoritmo.

E a gente começa com a função de inserção

Imagine que a árvore já foi construída até a posição `m`, e que agora nós vamos inserir mais um elemento




```
func  Insercao_HeapMax ( x, V, m )
     m++;    V[m] = x                  // colocando o x na lista
     i = m                             // colocando o dedo sobre ele
     Enquanto ( i > 1 )                // e levando ele para o lugar certo
        j = i/2                        // a posição de cima
        Se ( V[i] > V[j] )
           Troca ( V, i, j );    i = j
        Senão         Quebra
```

E agora que a gente tem a função de inserção, é fácil realizar a construção da árvore

```
func  Construção_Árvore ( V, 1, N )
     m = 1
     Para  i = 2  Até  N
        Inserção_HeapMax ( V[i], V, m )
```

Análise

Como a árvore tem altura $\log_n n$, o laço da função de inserção dá no máximo $\log_2 n$.

Logo, essa função executa em tempo $O(\log n)$.

E agora, é fácil ver que a construção da árvore leva tempo

$$O(n \log n)$$

4.2 Ordenação da lista

(. . .)