Construção e Análise de Algoritmos

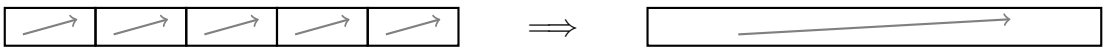
aula 03: Algoritmos eficientes de ordenação

1 Introdução

Na aula passada já apareceu a ideia que vai nos dar os algoritmos eficientes de ordenação.

Só que ainda é preciso trabalhar um pouquinho para chegar lá.

Quer dizer, a ideia consiste em fazer a ordenação por blocos



Mas, quando nós tentamos explorar essa ideia ao máximo (utilizando blocos de tamanho mínimo), nós chegamos a um dilema

1. quando o tamanho dos blocos é grande demais, a etapa de ordenação leva tempo demais
2. quando o tamanho dos blocos é pequeno demais, a etapa de intercalação leva tempo demais

Daí que, a gente foi forçado a escolher o meio termo: usar blocos de tamanho $\sqrt{n}$.

E isso nos deu um algoritmo mais ou menos eficiente — (*que executa em tempo* $O(n\sqrt{n})$

Na aula de hoje, nós vamos ver como escapar desse dilema.

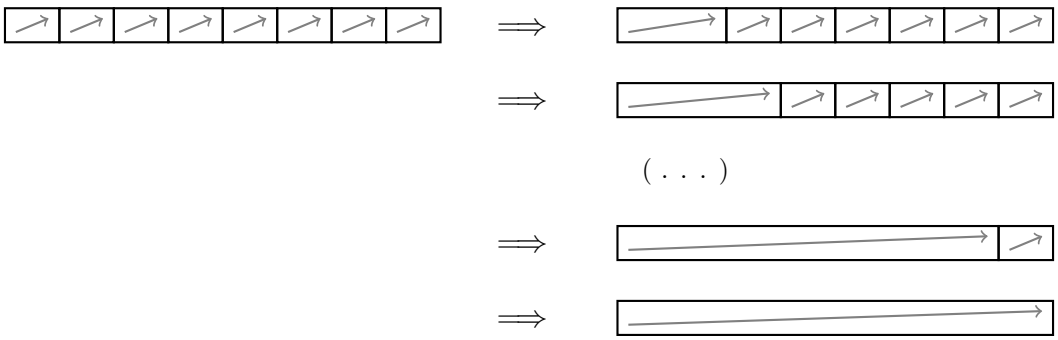
2 Intercalação esperta

Quer dizer, na prática a gente só vai usar a mesma ideia da aula passada outra vez.

Veja só.

Imagine que nós temos uma lista com 8 blocos ordenados

Na aula passada, a gente obtinha a lista completamente ordenada a partir daí, assim



Quanto tempo leva isso?

Bom, a `Intercalação()` leva tempo proporcional ao tamanho da porção da lista que está sendo intercalada — (*i.e.*, $O(n)$)

Daí que,

- a primeira intercalação leva tempo $O(\frac{2n}{8})$
- a segunda intercalação leva tempo $O(\frac{3n}{8})$
- a terceira intercalação leva tempo $O(\frac{4n}{8})$
- ...
- a última intercalação leva tempo $O(\frac{8n}{8})$

Logo, a coisa toda leva tempo

$$\frac{2n}{8} \quad + \quad \frac{3n}{8} \quad + \quad \frac{4n}{8} \quad + \quad \ldots \quad + \quad \frac{8n}{8} \qquad = \qquad \frac{35n}{8}$$

E no caso geral, onde nós temos $\sqrt{n}$ blocos, a coisa dá

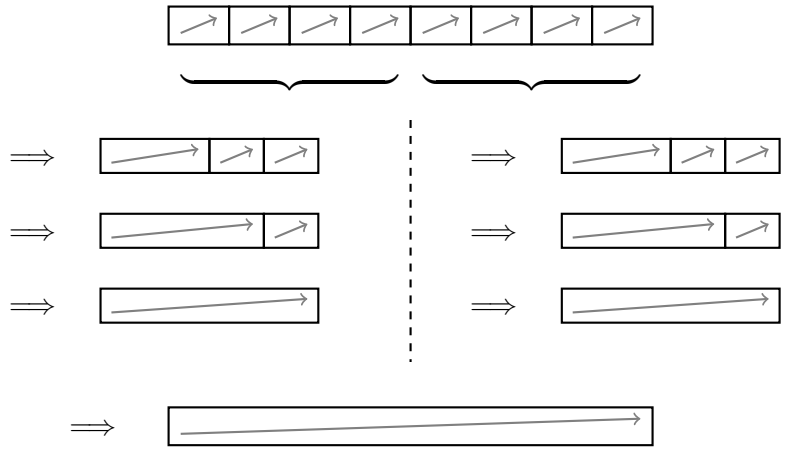
$$\frac{2n}{\sqrt{n}} \quad + \quad \frac{3n}{\sqrt{n}} \quad + \quad \frac{4n}{\sqrt{n}} \quad + \quad \ldots \quad + \quad \frac{\sqrt{n} \cdot n}{\sqrt{n}} \qquad = \qquad \frac{n \cdot \sqrt{n}}{2}$$

Certo.

Mas, a gente não precisa fazer as coisas assim.

Quer dizer, a gente também pode

- fazer a primeira metade das intercalações primeiro
- fazer a primeira metade das intercalações depois
- e daí, fazer a intercalação da coisa toda



Sim! essa é a ideia da aula passada

→ *Fazer duas vezes a metade de uma coisa pode ser mais rápido do que fazer a coisa inteira de uma só vez*

Será?

Bom, vamos fazer as contas para ver.

Quer dizer, a coisa fica assim

$$2 \times \left(\frac{2n}{8} + \frac{3n}{8} + \frac{4n}{8} \right) + n = \frac{18n}{8}$$

E no caso geral

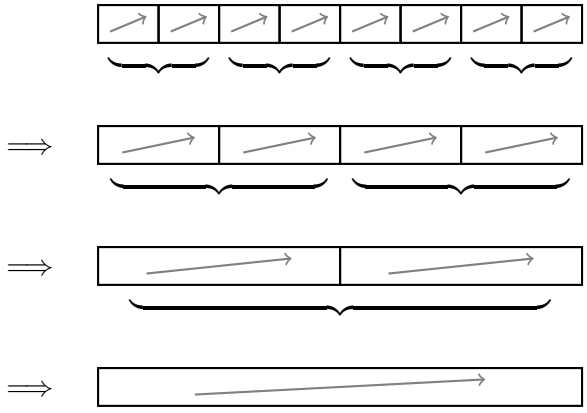
$$2 \times \left(\frac{2n}{\sqrt{n}} + \frac{3n}{\sqrt{n}} + \ldots + \frac{\left(\sqrt{n}/2\right) \cdot n}{\sqrt{n}} \right) + n = \frac{n \cdot \sqrt{n}}{4}$$

Deu certo! — (*o tempo foi reduzido à metade*)

E agora?

Bom, agora é só aplicar a esperteza de novo — (*e de novo, e de novo ...*)

Quer dizer, a gente vai intercalar a lista com 8 blocos assim



Quanto tempo leva isso?

Bom,

- a primeira etapa leva tempo

$$4 \times \frac{n}{4} = O(n)$$

- a segunda etapa leva tempo

$$2 \times \frac{n}{2} = O(n)$$

- e a terceira etapa leva tempo $O(n)$

Sim! é tudo a mesma coisa.

Daí que, a coisa toda leva tempo

$$O(n) + O(n) + O(n) = 3 \cdot O(n)$$

Nesse ponto já não é difícil ver que

- uma lista com 16 blocos é intercalada em tempo $4 \cdot O(n)$
- uma lista com 32 blocos é intercalada em tempo $5 \cdot O(n)$

E, em geral

- uma lista com 2^k blocos é intercalada em tempo $k \cdot O(n)$

A esperteza agora é escrever

$$k \cdot O(n) \quad = \quad \log_2\left(2^k\right) \cdot O(n)$$

E assim, a gente vê que a lista com $\sqrt{n}$ blocos é intercalada em tempo

$$\log_2\left(\sqrt{n}\right) \cdot O(n) \quad = \quad O(n \cdot \log n)$$

— (*porque?*)

Legal, não é?

3 Ordenação por intercalação

Sim.

Não só é legal, como vai nos permitir escapar do dilema.

Quer dizer, utilizando a intercalação esperta, a situação que nós temos com $\sqrt{n}$ blocos é

$$\begin{array}{rrcl} \texttt{ordenação:} & \sqrt{n} \cdot \left(\sqrt{n}\right)^2 & = & n \cdot \sqrt{n} \\ \texttt{intercação:} & \log\left(\sqrt{n}\right) \cdot n & = & \frac{1}{2}\, n \cdot \log n \end{array}$$

Quer dizer, os tempos de ordenação e intercalação não são mais iguais.

E isso significa que a gente pode continuar diminuindo o tamanho dos blocos.

Mas, até quando a gente pode diminuir?

Bom, agora vai acontecer uma coisa engraçada ...

Quer dizer, imagine que a lista é dividida em blocos de tamanho $\sqrt[4]{n} \;=\; n^{1/4}$

De modo que o número total de blocos é $n^{3/4}$ — (*porque?*)

Daí, a situação fica assim

$$\begin{array}{rrcl} \texttt{ordenação:} & n^{3/4} \cdot \left(n^{1/4}\right)^2 & = & n \cdot \sqrt[4]{n} \\ \texttt{intercação:} & \log\left(n^{3/4}\right) \cdot n & = & \frac{3}{4}\, n \cdot \log n \end{array}$$

Mas,

$$\frac{3}{4} \log n \quad < \quad \sqrt[4]{n}$$

— (*porque?*)

Quer dizer, o tempo de ordenação continua maior do que o tempo de intercalação.

Logo, a gente pode continuar diminuindo o tamanho dos blocos.

Imagine agora que nós dividimos a lista em blocos de tamanho $\sqrt[8]{n} = n^{1/8}$
— (*com um total de* $n^{7/8}$ *blocos*)

Daí, a situação fica assim

$$\begin{aligned} \texttt{ordenação:} \quad & n^{7/8} \cdot \left(n^{1/8}\right)^2 = n \cdot \sqrt[8]{n} \\ \texttt{intercação:} \quad & \log\left(n^{7/8}\right) \cdot n = \tfrac{7}{8}\, n \cdot \log n \end{aligned}$$

Mas,

$$\frac{7}{8} \log n < \sqrt[8]{n}$$

E o tempo de ordenação continua maior do que o tempo de intercalação.

Você já percebeu o que está acontecendo?

Sim! nós podemos diminuir o tamanho dos blocos para sempre.

E fazendo isso, a gente chega em blocos de tamanho 1.

Só que um bloco de tamanho 1 não precisa ser ordenado.

Quer dizer, não vai mais ter uma etapa de ordenação.

Só o que vai ter é intercalação.

Em outras palavras, nós acabamos de obter o algoritmo de *ordenação por intercalação*.

Esse algoritmo divide a lista em blocos de tamanho 1.

E o seu tempo de execução é $O(n \log n)$.

4 O algoritmo Mergesort

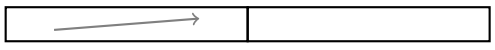
Mas, a gente pode pensar a mesma coisa de um jeito diferente.

Quer dizer, a gente pode organizar a computação de maneira recursiva.

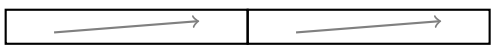
A ideia é que, para ordenar uma lista de tamanho n

Para ordenar a lista com 4 blocos

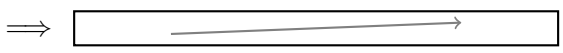
1. primeiro a gente ordena a primeira metade



2. depois a gente a gente ordena a segunda metade



3. e daí, a gente faz a intercalação

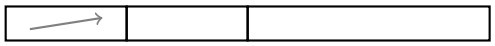


Só que a coisa é recursiva.

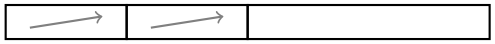
Quer dizer, a ordenação das metades acontece da mesma maneira.

Por exemplo, para ordenar a primeira metade da lista

1. primeiro a gente ordena a primeira metade da metade



2. depois a gente a gente ordena a segunda metade da metade



3. e daí, a gente faz a intercalação

E a recursão continua dividindo a lista, até chegar aos blocos de tamanho 1.

Daí que, no final das contas, só o que existe é intercalação.

A implementação recursiva da ordenação por intercalação é conhecida como o *algoritmo mergesort*

```
void  Mergesort ( L, inicio, fim )
 {
    Se ( inicio = fim )   Retorna            // blocos de tamanho 1

    meio  <--  ( inicio + fim ) / 2          // divide no meio

    Mergesort ( L, incio, meio )             // ordena a primeira metade

    Mergesort ( L, meio+1, fim )             // ordena a segunda metade

    Intercalação ( L, inicio, meio, fim )    // intercala
 }
```

Como esse é essencialmente o algoritmo que nós vimos na seção anterior, o seu tempo de execução é $O(n \log n)$.

5 Ordenação por partição

Legal.

Mas, quando a gente entende uma coisa, a gente sempre pode pensar em outro jeito de fazer a mesma coisa.

Quer dizer, a gente acabou de entender que

→ `Para ordenar uma lista, basta`

`- ordenar a primeira metade`

`- ordenar a segunda metade`

`- e depois fazer a intercalação`

Mas, porque é preciso fazer a intercalação?

Bom, isso é necessário porque

- os elementos grandes da 1a metade precisam ir para a 2a metade
- e os elementos pequenos da 2a metade precisam ir para a 1a metade

Mas,

- se a 1a metade só tiver elementos pequenos
- e a 2a metade só tiver elementos grandes

então, não é preciso fazer a intercalação ...

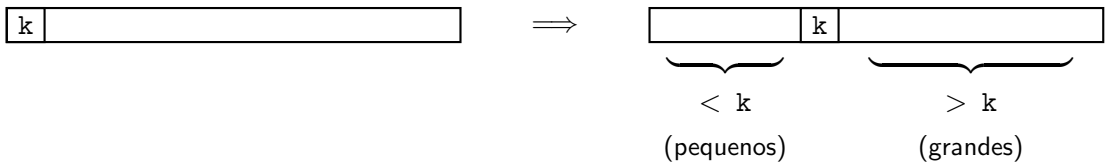
Hmm!

Isso significa que a gente também pode ordenar uma lista assim

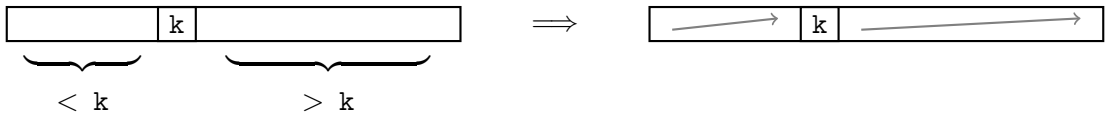
1. `primeiro, a gente separa os elementos pequenos e grandes`
2. `depois, a gente ordena o grupo dos pequenos`
3. `e depois, a gente ordena o grupo dos grandes`

Certo.

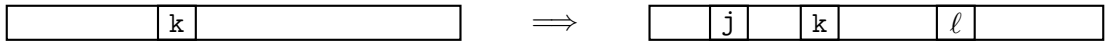
A primeira etapa é realizada pelo procedimento de *partição*



E as etapas 2 e 3 são feitas recursivamente.



Quer dizer, como a ordenação é recursiva, ela é feita por meio de novas chamadas ao procedimento de partição — (*até chegar aos blocos de tamanho 1*)



Daí que, no final das contas, a lista é ordenada apenas por meio de partições.

Quer dizer, agora nós temos um algortitmo de *ordenação por partição.*

A implementação recursiva dessa ideia é conhecida como o *algoritmo quicksort.*

```
void  Quicksort ( L, inicio, fim )
 {
    Se ( inicio = fim )   Retorna              // blocos de tamanho 1

    meio  <--  Partição ( L, inicio, fim )     // separa pequenos e grandes

    Quicksort ( L, inicio, meio-1] )           // ordena os pequenos

    Quicksort ( L, meio+1, fim] )              // ordena os grandes
 }
```

Análise

Mas ainda falta dizer uma coisa.

Quer dizer, em geral o procedimento `Partição()`não divide a lista exatamente na metade

`< figura >`

Daí que, em geral, o Quicksort não executa exatamente como o Mergesort.

Daí que, nós não temos a garantia de tempo $O(n \log n)$.

E agora vem uma má notícia ...

Quer dizer, no pior caso as partições acontecem sempre da pior maneira possível.

Por exemplo,

`< figura >`

— (*i.e., todos os elementos (exceto o pivô) ficam no mesmo lado da partição*)

Nesse caso, vão ocorrer n partições em sequência.

E como a partição leva tempo proporcional ao tamanho da lista, isso vai nos dar um total de

$$n \;+\; (n-1) \;+\; (n-2) \;+\; \ldots \;+\; 2 \;+\; 1 \qquad = \qquad O(n^2)$$

Arrrgh!!

Mas isso é apenas uma má notícia teórica.

Quer dizer, na prática isso nunca acontece.

E na prática, o Quicksort é muito, muito rápido — (*é por isso que ele tem esse nome ...*)

Veja só,

```
quicksort:  .....
mergesort:  .....
```

Legal, não é?

6 Ordenação por árvore

Agora, nós vamos entender melhor as coisas.

E ao fazer isso, nós vamos resolver todos os nossos problemas — (*pelo menos no que diz respeito à ordenação de listas ...*)

Quer dizer, a gente começa perguntando

- *Aonde está o ganho de eficiência do Mergesort?*

Bom, na realidade são duas coisas.

A primeira delas é que o algoritmo nunca compara o mesmo par de elementos duas vezes.

Porque, quando a comparação acontece, os elementos são colocados em uma lista ordenada.

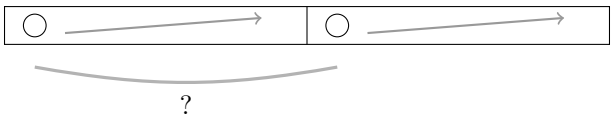
E os elementos dessa lista nunca são comparados outra vez — (*porque não precisa ...*)

Mas, isso não basta para tornar um algoritmo de ordenação eficiente — (*porque?*)

Daí vem a segunda coisa.

E a segunda coisa é a transitividade.

Quer dizer, quando a gente compara os primeiros elementos de duas listas ordenadas, e descobre quem é o menor

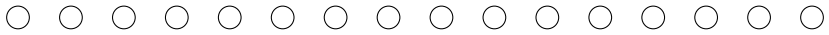


a gente ganha de graça o fato de que ele é menor do que todos os elementos da outra lista — (*e isso é um monte de comparações que a gente não precisa fazer*)

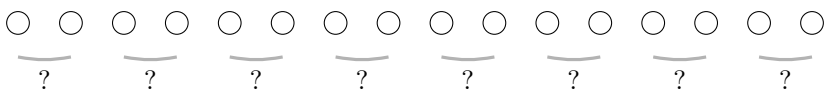
Legal.

Agora que a gente entendeu, nós podemos considerar o problema da ordenação outra vez.

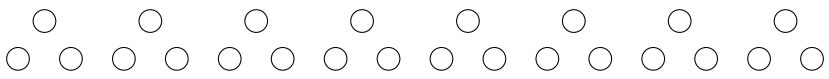
Quer dizer, imagine que os elementos da lista estão um ao lado do outro — (*em uma fila*)



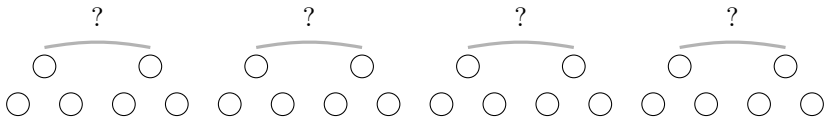
Daí, a gente compara os pares de elementos consecutivos



E daí, como a gente não quer fazer essas comparações outra vez, a gente copia o menor de cada par para cima

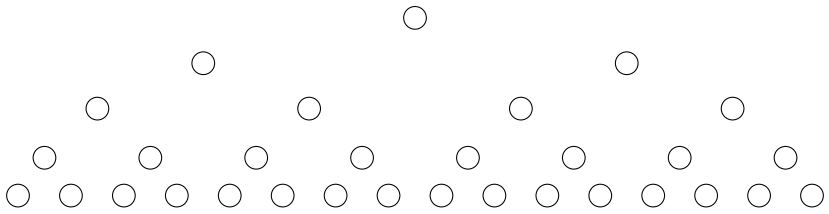


Daí, o próximo passo é comparar os pares de elementos consecutivos que foram copiados para cima



Já deu pra ver o que está acontecendo, não é?

Quer dizer, a gente está construindo uma estrutura de árvore, onde o menor elemento fica no topo

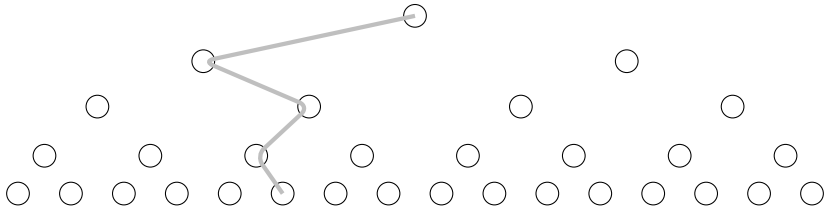


E nesse ponto a gente pergunta

- *Como é que a gente acha o segundo maior ?*

Bom, é fácil

- basta percorrer outra vez o caminho que foi feito pelo menor
- e encontrar o menor elemento que foi comparado com ele



E daí vem uma esperteza.

Quer dizer, a ideia é apagar o caminho do segundo maior — (*fazendo os elementos que foram comparados com ele subir no caminho*)

E fazer o segundo menor percorrer o caminho do maior para chegar ao topo.

Porque daí, a gente está na mesma situação outra vez.

E pode encontrar o terceiro maior, o quarto maior, etc ...

Quer dizer, aqui a gente tem mais um algoritmo de ordenação.

E daí, a gente pergunta

- *Qual é o tempo de execução desse algoritmo ?*

Bom, a primeira etapa leva tempo

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

Na segunda etapa, a gente precisa descer e subir na árvore para obter o próximo menor elemento — (*e ajustar os caminhos*)

Logo, a coisa toda leva tempo

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

E aqui nós temos mais um algoritmo eficiente de ordenação.

Mas, a gente não ganhou nada.

Porque esse algoritmo precisa de memória adicional para armazenar a árvore — (*e a lista auxiliar que recebe os menores elementos, como o Mergesort*)

E agora?

7 O algoritmo Heapsort

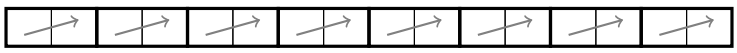
Bom, agora é a hora para mais uma esperteza.

Quer dizer, a ideia é armazenar a árvore dentro da própria lista.

E para isso, a gente relembra a intuição inicial do algoritmo

→ *organizar as coisas na memória para lembrar os resultados das comparações*

Outra maneira de fazer isso é comparar pares de elementos consecutivos, e mover o menor para a esquerda

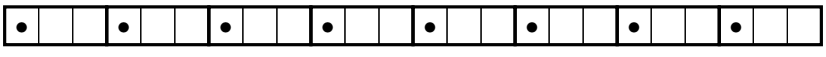


como o Mergesort.

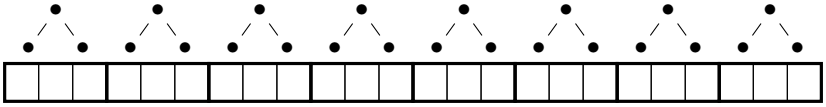
Mas, não é por aí que a gente vai ...

Quer dizer, a ideia é dividir a lista em grupinhos de 3.

Daí, a gente move o menor dos 3 para a primeira posição do grupo

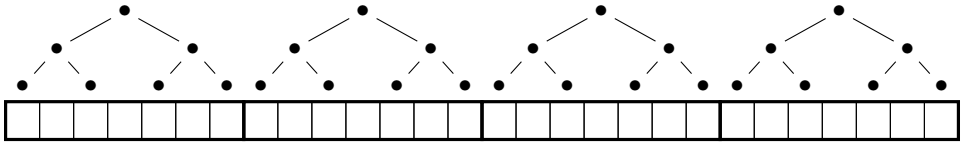


E daí, a gente visualiza (ou imagina) a coisa assim

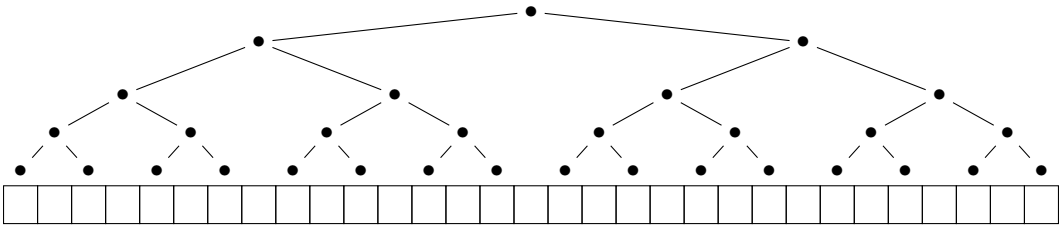


Sim! são arvorezinhas.

Trabalhando com grupos de 7 a gente obtém árvores maiores



E continuando assim, a gente monta uma árvore completa



Ou melhor, isso é um heapMin — (*onde cada elemento é menor do que os seus dois filhos*)

Só que a gente também pode pensar que isso é o heap-Max — (*porque é a mesma coisa*)

Daí, a ideia é fazer a remoção do elemento no topo do heap — (*o maior elemento da lista*)

E colocá-lo na última posição da lista — (*que foi liberada pela remoção*)

Daí, a gente remove o novo elemento do topo — (*o segundo maior elemento da lista*)

E colocá-lo na penúltima posição da lista.

Mas, para a coisa funcionar, é preciso utilizar o esquema tradicional de indexação do heap — (*onde os filhos do elemento na posição k ficam nas posições $2k$ e $2k+1$*)

Dessa maneira, a gente ordena a lista em tempo $O(n \log n)$ — (*no pior caso*)

Sem a necessidade de usar uma lista auxiliar.