# Estruturas de Dados

## *B-Trees*

## *Splay Trees*

**2025/2026**

# Árvores e Memória Persistente

- As árvores equilibradas são úteis quando se pretende manusear informação armazenada em memória persistente?
  - Se sim, como?

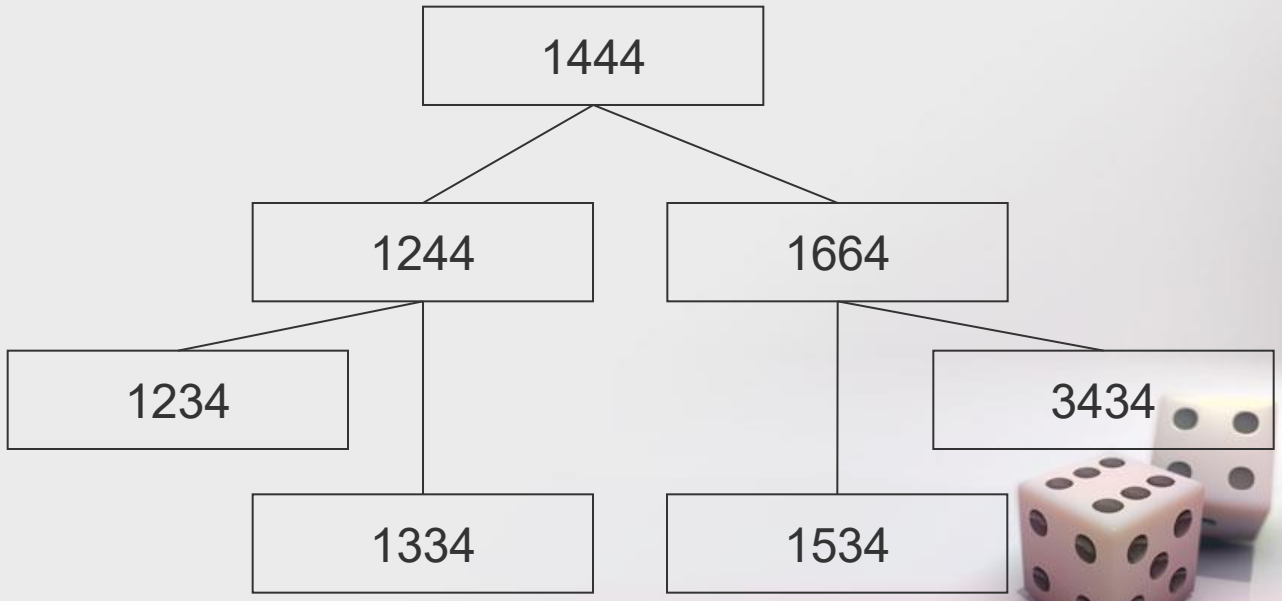
# Árvores e Memória Persistente

- A ideia é utilizar árvores para organizar a informação em disco, com benefícios análogos:

Ficheiro Monolítico

| 1234 |
| --- |
| 1244 |
| 1334 |
| 1444 |
| 1534 |
| 1664 |
| 3434 |

Estrutura em árvore



# Árvores e Memória Persistente

- Estas solução são aplicadas através do uso de motores de bases de dados adequados.
- Será que as árvores binárias equilibradas tradicionais são adequadas para este propósito?

# Árvores e Memória Persistente

- Estas solução são aplicadas através do uso de motores de bases de dados adequados.
- Será que as árvores binárias equilibradas tradicionais são adequadas para este propósito?
  - Não! O acesso a disco é demasiado lento.
    - Para 10000000 itens de dados, uma árvore binária **ideal** resulta em cerca de 23 acessos.
    - Cada  acesso a disco é muito lento (tempos de acesso na ordem dos 10 ms)
    - Precisamos de reduzir o número de acessos a dados, mesmo ao custo de maior complexidade computacional, para além do que aquilo que uma árvore binária oferece.

# Árvore M-ária

- Vamos guardar não dois, mas sim vários (M) items de dados em cada nodo
  - Com o objectivo de reduzir o número total de acessos (em pesquisa) para 2 ou 3…
- Um tipo de árvore M-ária usado frequentemente são as *B-trees*.
  - Há várias variantes, vamos analisar a mais comum, a B+-tree.

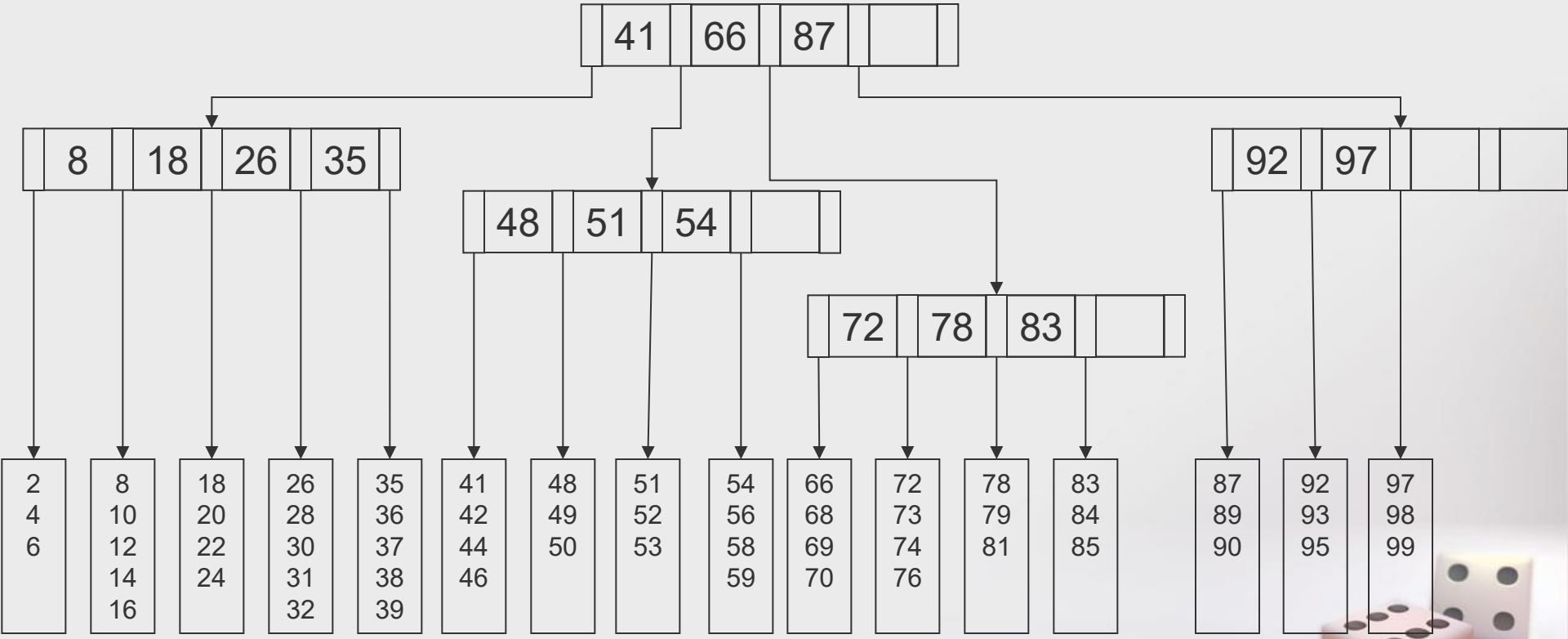
# B-Tree

- Um B-tree de ordem M é uma árvore M-ária que verifica as seguintes propriedades:
  1. Os dados são guardados nas folhas
  2. Cada nodo não-folha guarda até M-1 chaves de pesquisa:
     - Cada chave i representa o menor indice na sub-árvore i+1.
  3. A raiz é uma folha ou tem entre 2 e M descendentes.*
  4. Cada nodo não-folha tem entre M/2 (arredondado para cima) e M descendentes.
  5. Todas as folhas têm entre L/2 (arredondado para cima) e L items de dados*

  * Ignorada para as primeiras inserções.

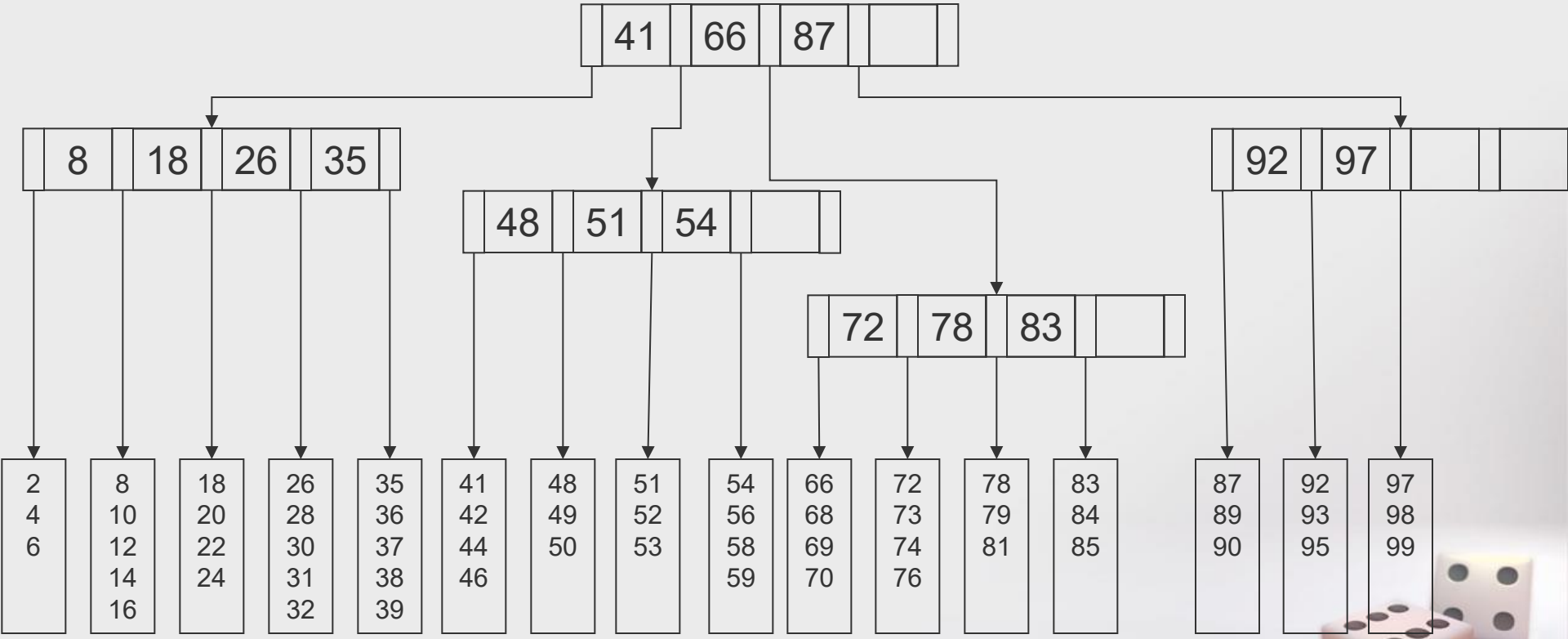
# BTree

Exemplo:



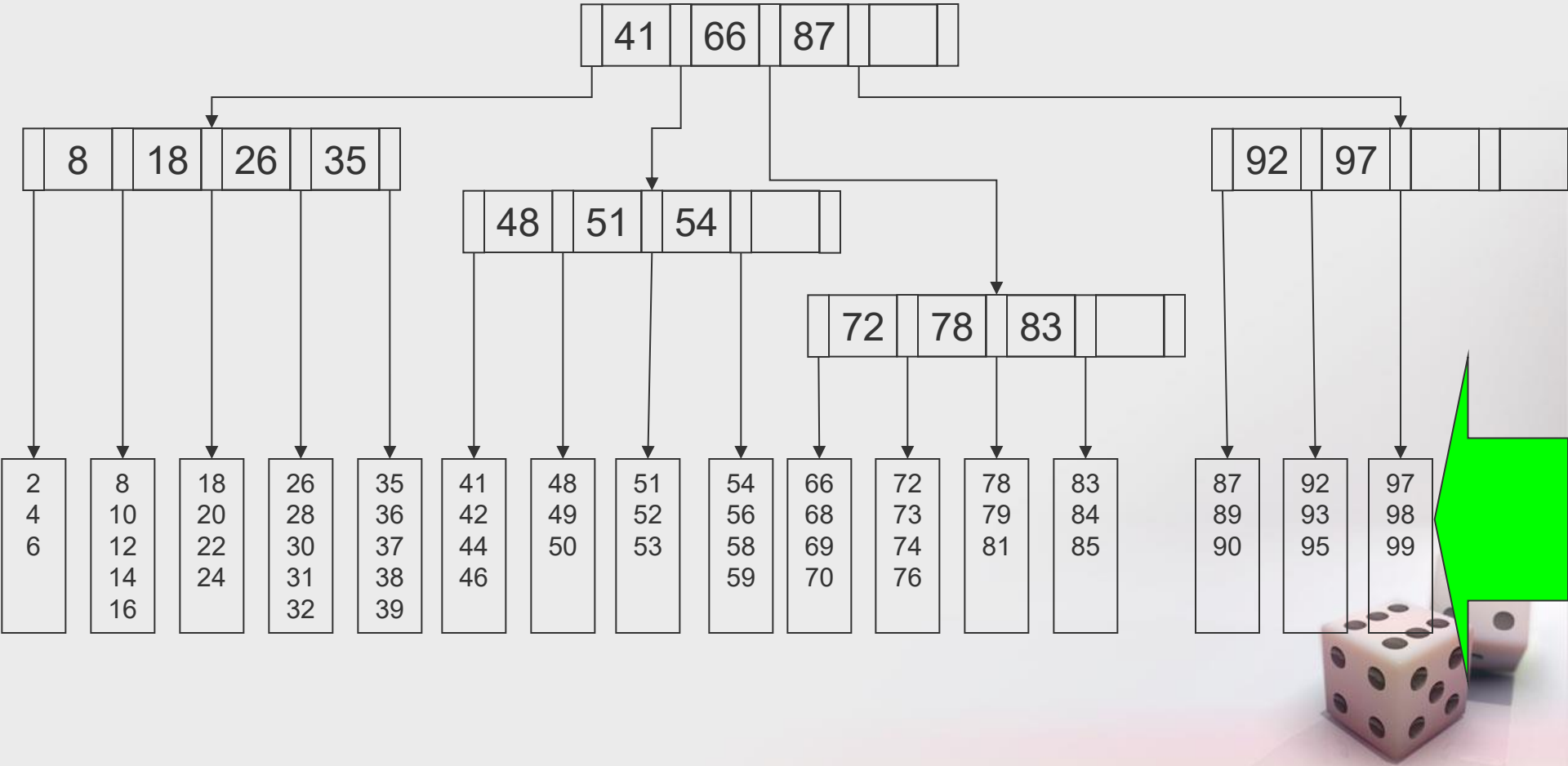
# BTree

Propriedade 1: Os dados são guardados nas folhas



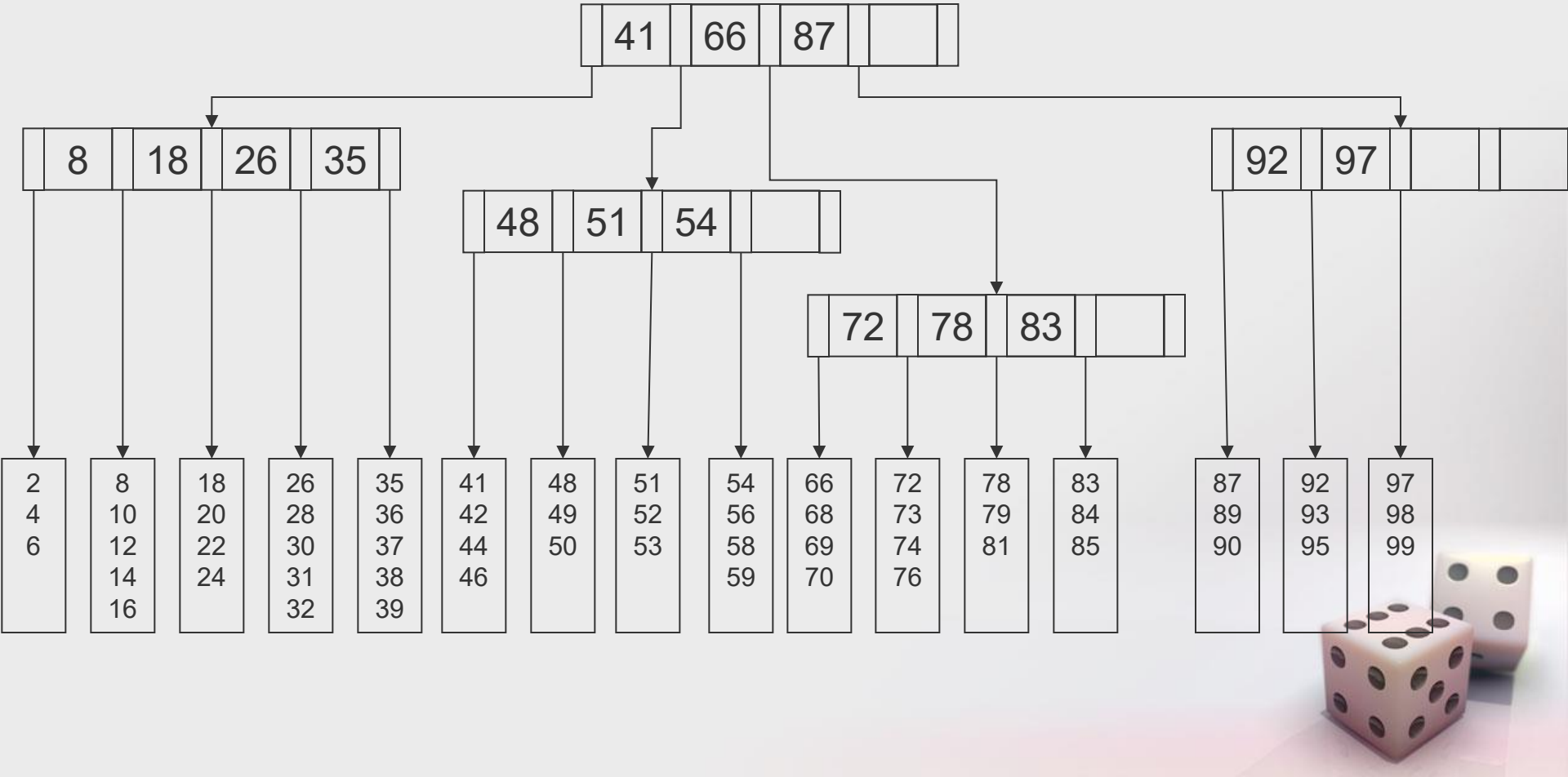
# BTree

Propriedade 1: Os dados são guardados nas folhas



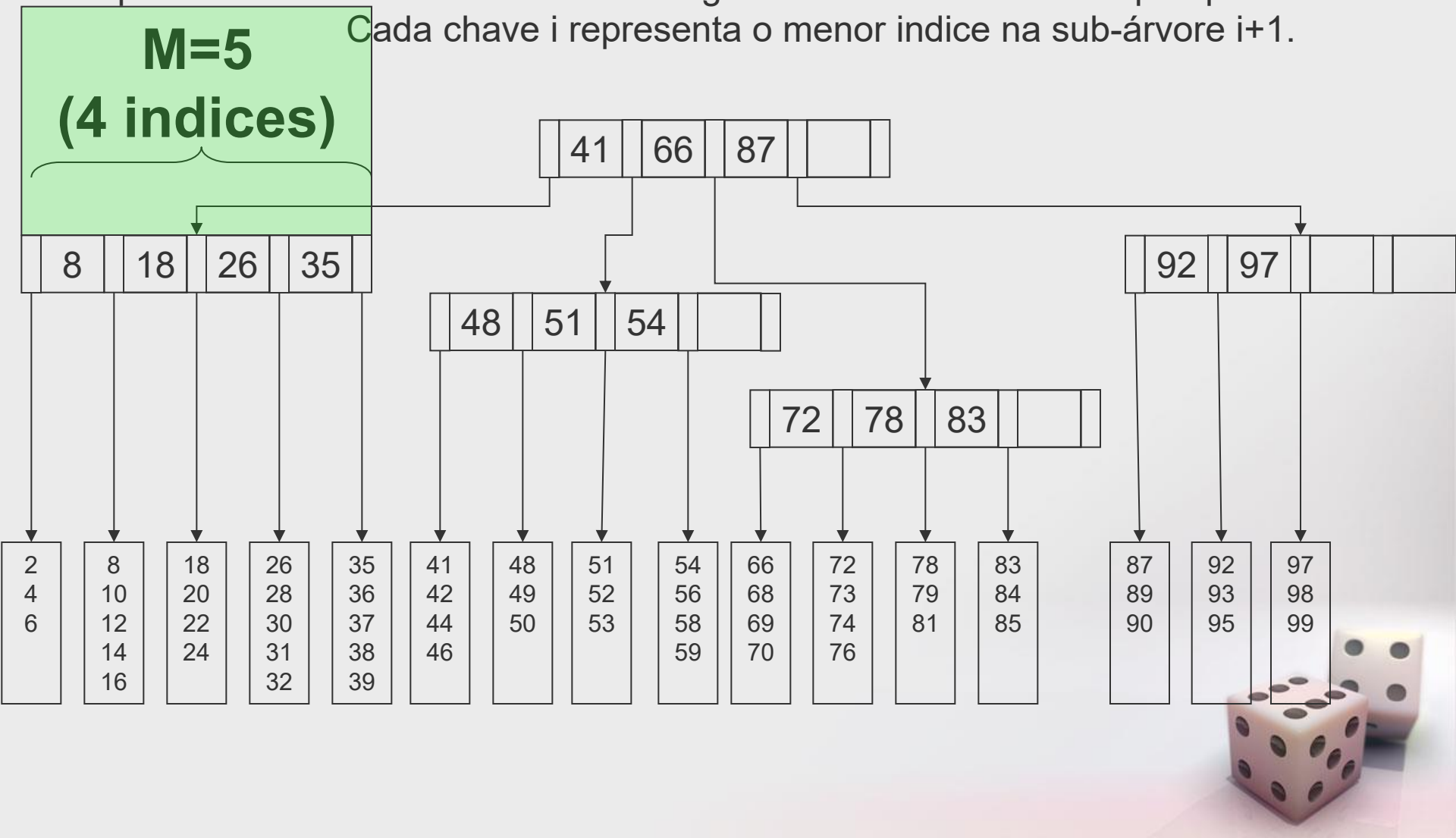
# BTree

Propriedade 2: Cada nodo não-folha guarda até M-1 chaves de pesquisa:
Cada chave i representa o menor indice na sub-árvore i+1.



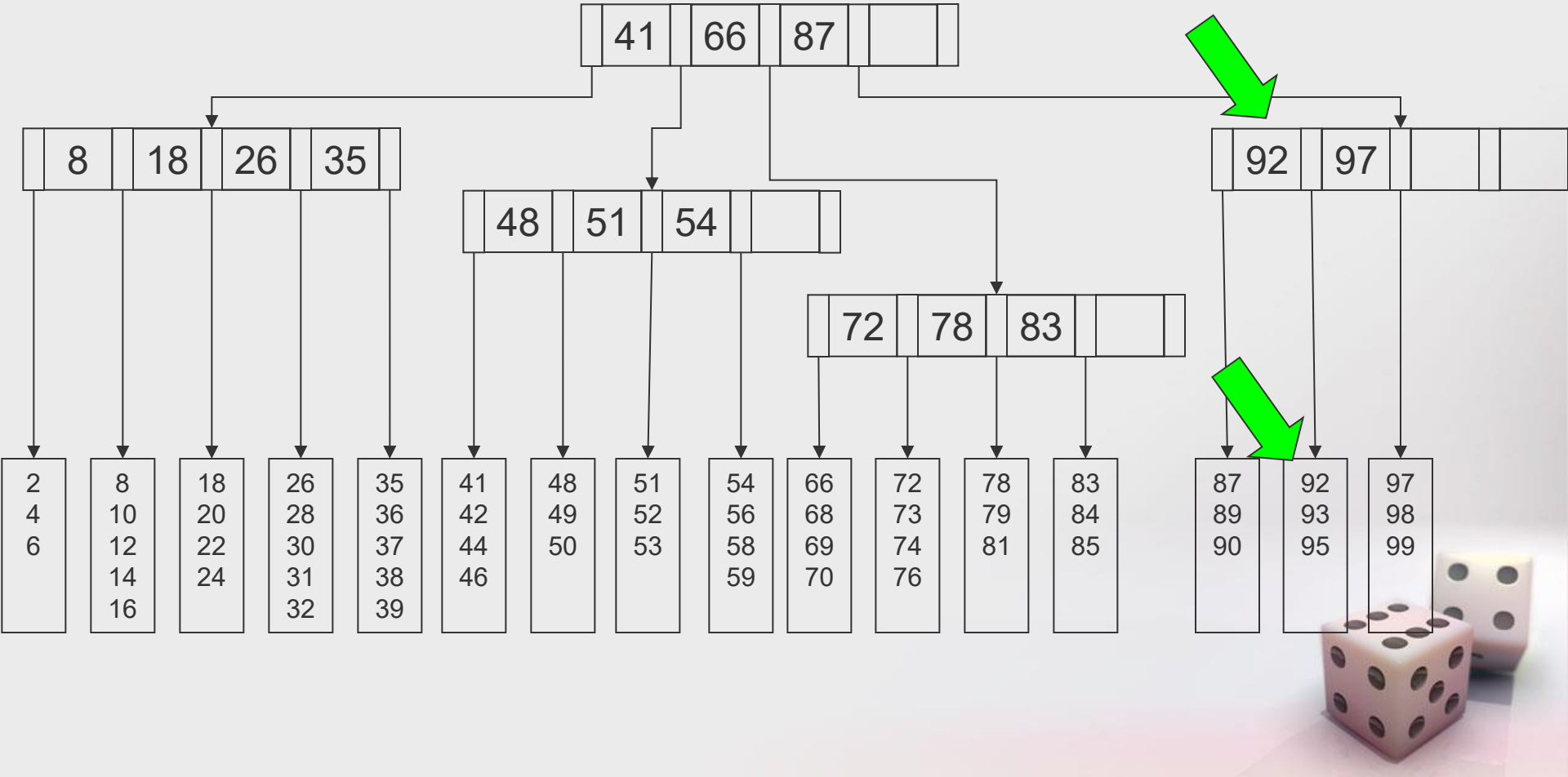
# BTree

Propriedade 2: Cada nodo não-folha guarda até M-1 chaves de pesquisa:
Cada chave i representa o menor indice na sub-árvore i+1.



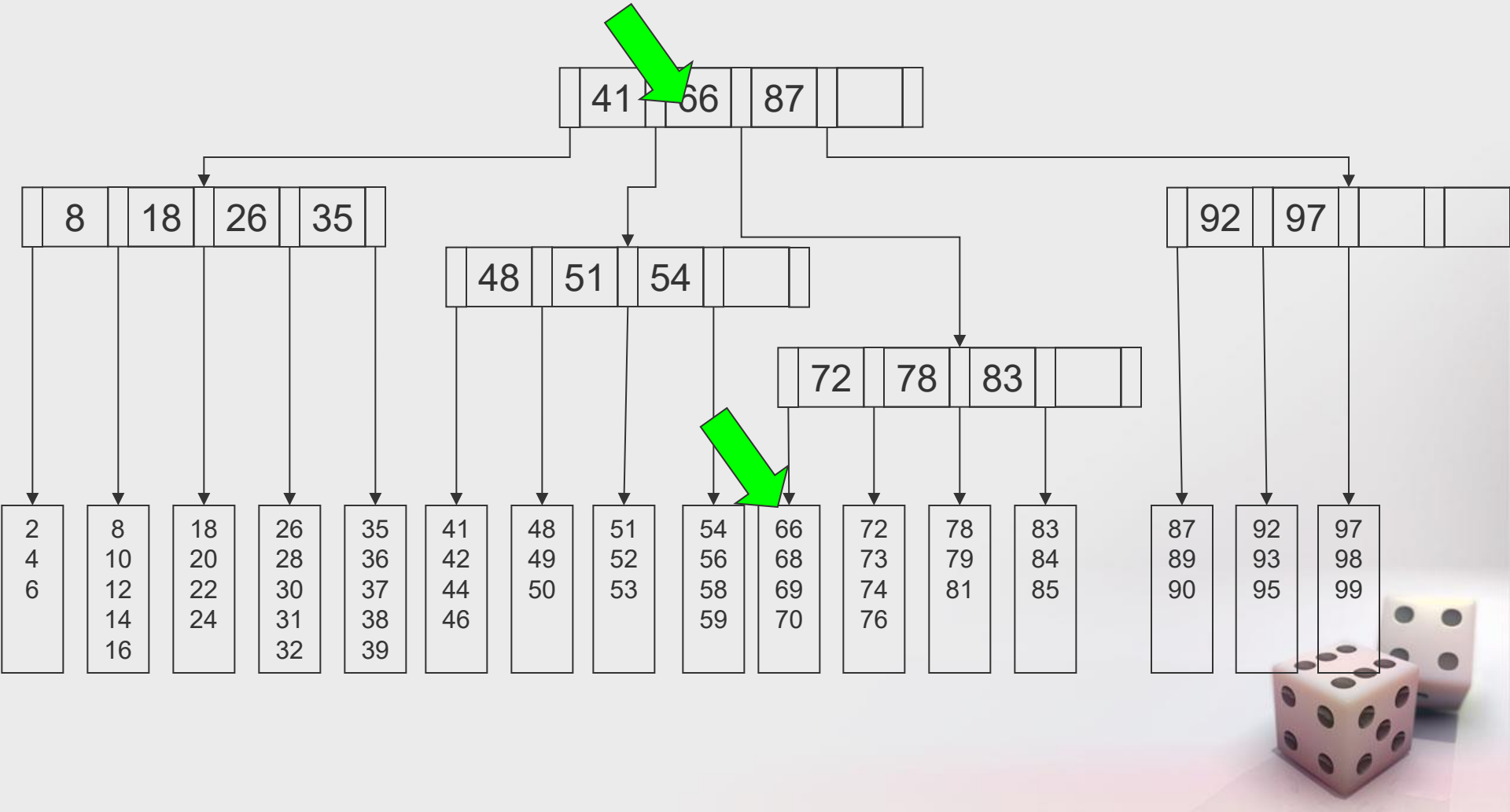
# BTree

Propriedade 2: Cada nodo não-folha guarda até M-1 chaves de pesquisa:
Cada chave i representa o menor indice na sub-árvore i+1.



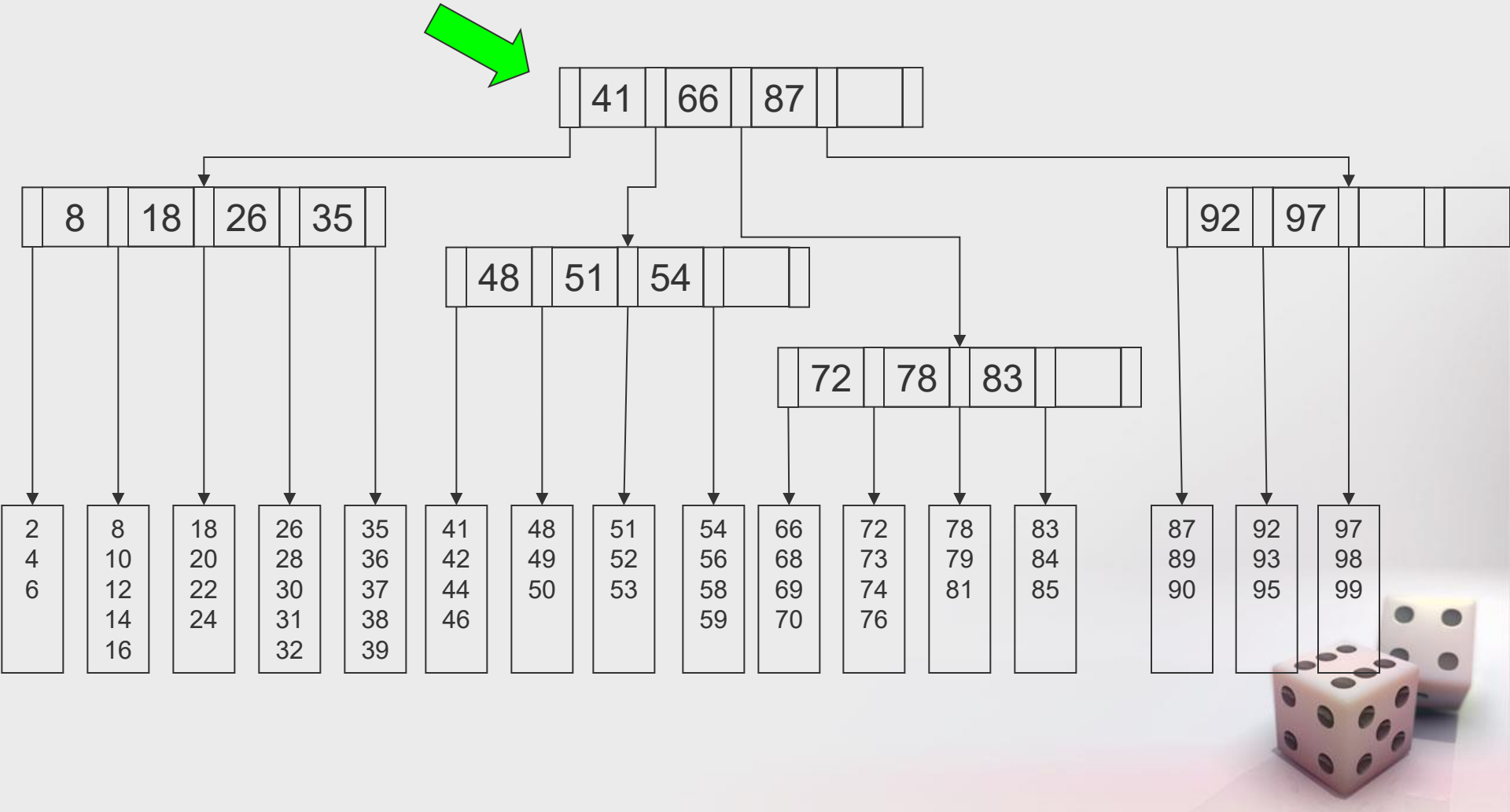
# BTree

Propriedade 2: Cada nodo não-folha guarda até M-1 chaves de pesquisa:
Cada chave i representa o menor indice na sub-árvore i+1.



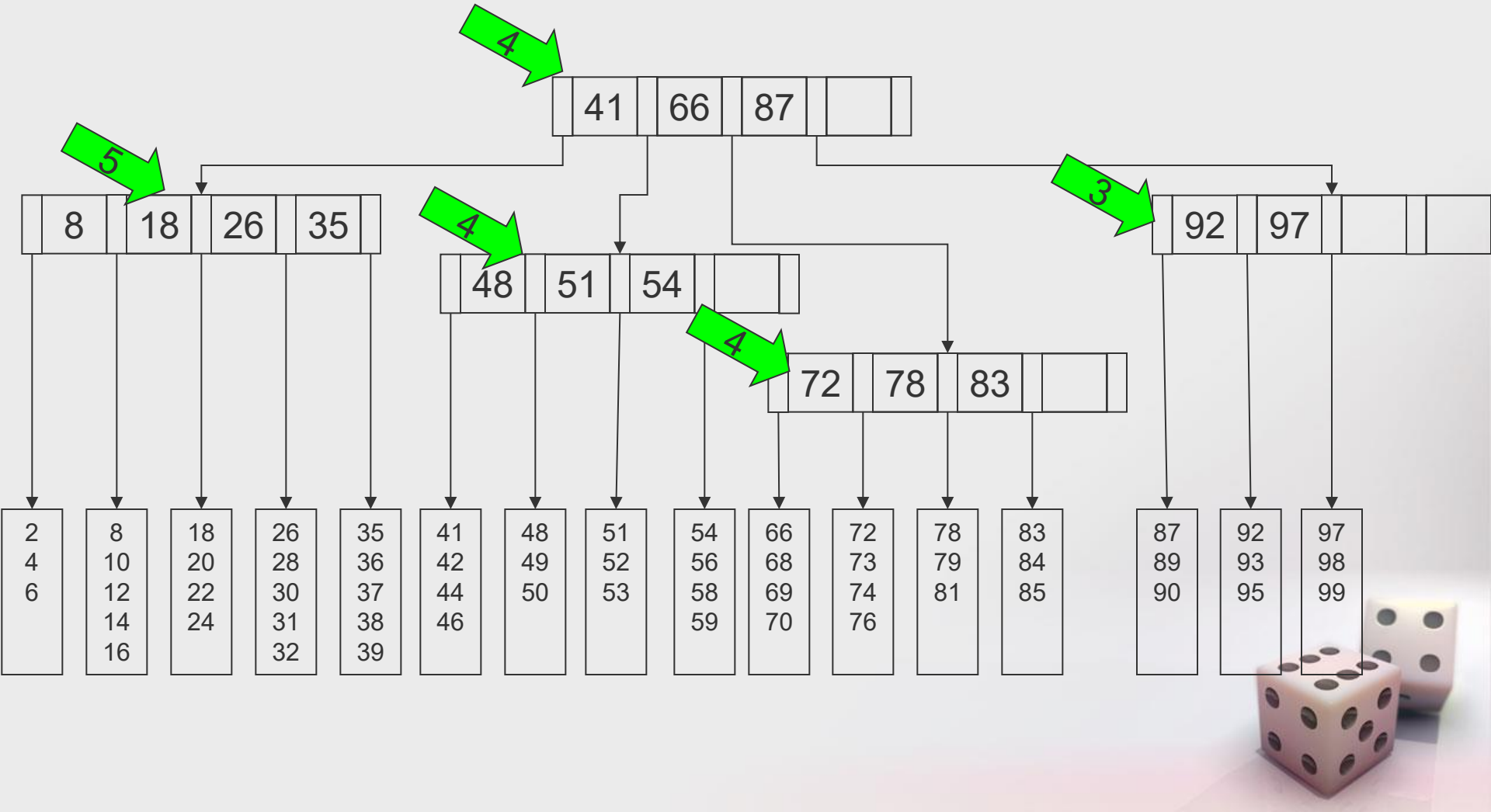
# BTree

Propriedade 3: A raiz é uma folha ou tem entre 2 e M descendentes.



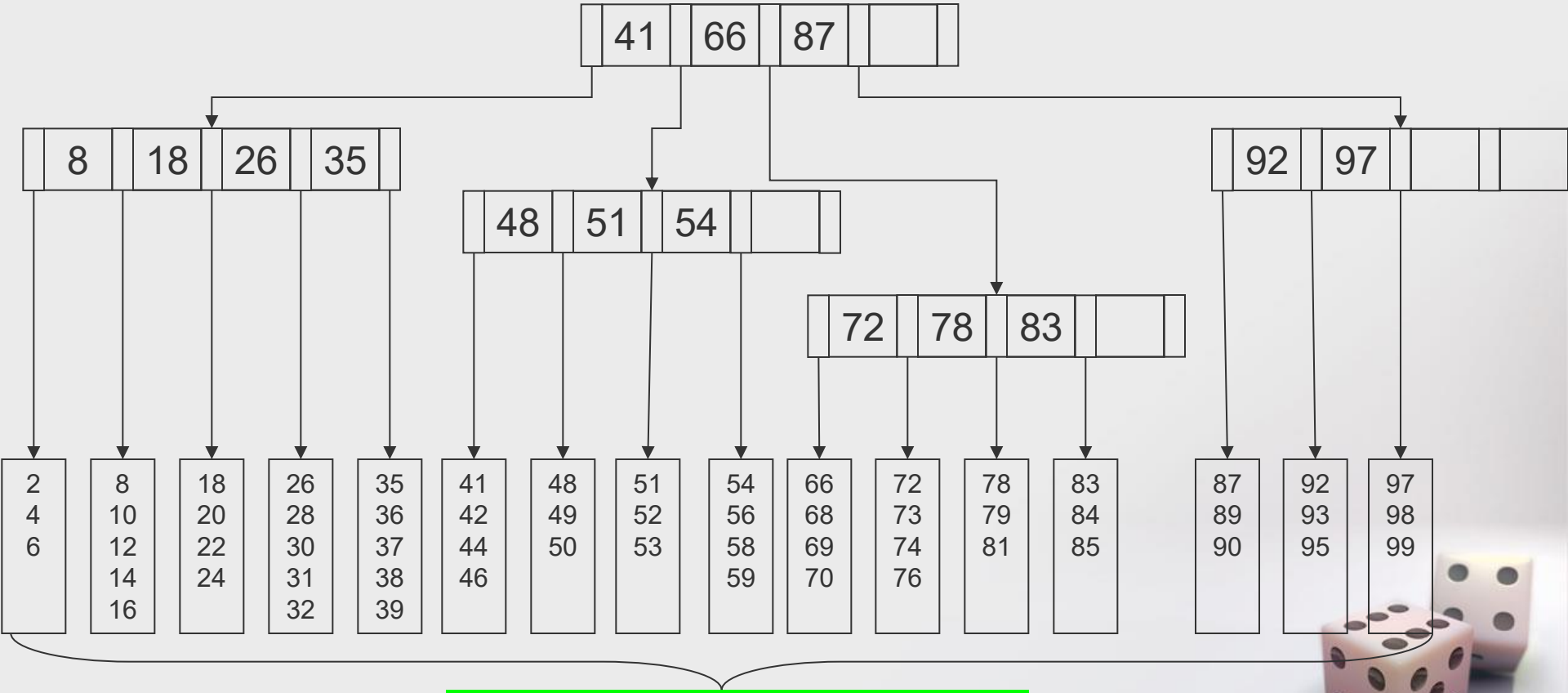
# BTree

Propriedade 4: Cada nodo não-folha tem entre M/2 e M descendentes.



# BTree

Propriedade 5: Todas as folhas têm entre L/2 e L items de dados




**L=5 : 3 a 5 items de dados**

# B-Tree

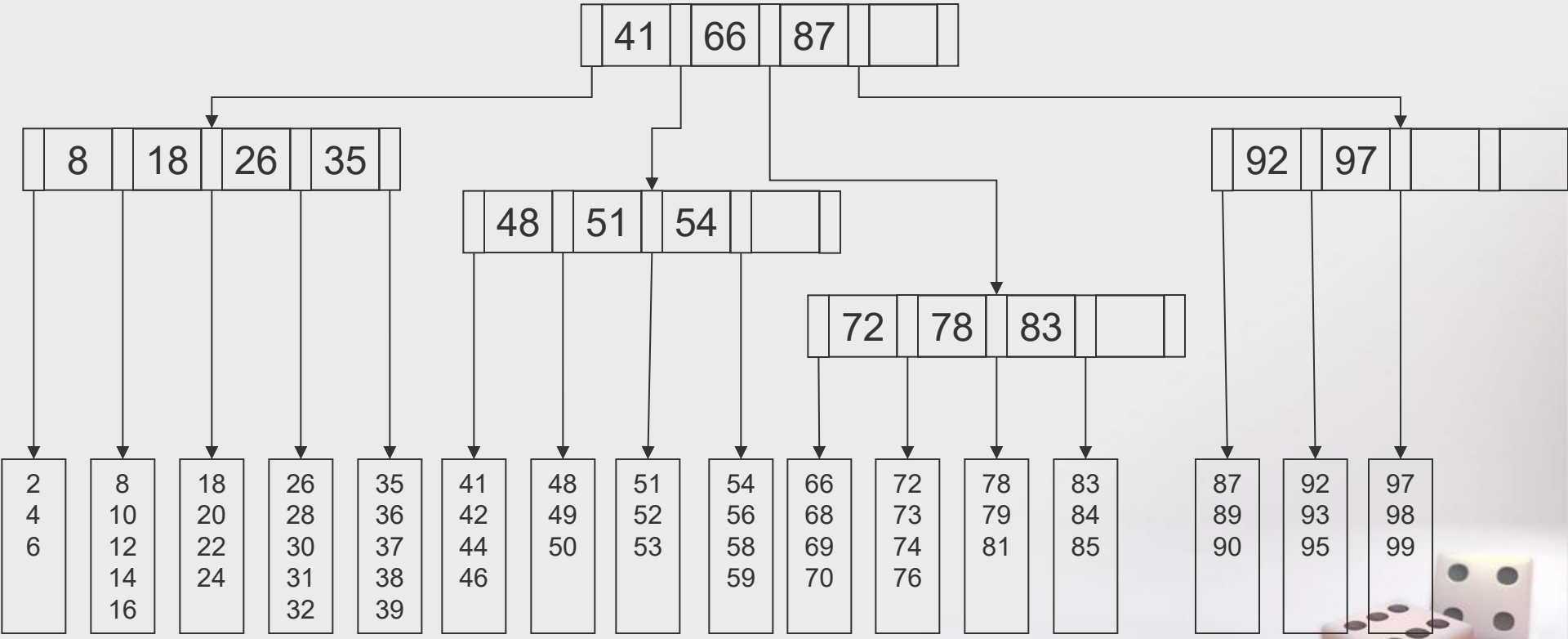
- O tamanho de M e L é escolhido de forma a maximizar a utilidade de cada bloco do disco:
  - Todos os acessos aos discos são feitos bloco-a-bloco, não existindo vantagem em fazer acessos mais pequenos:
  - Por exemplo, se os sectores forem de 8k e cada registo de dados ocupar 256 bytes, então L=8k/256=32 registos por bloco.
- O maior número de acessos é dado aprox. por:

$$\log_{M/2} N$$

P.ex: para 10000000 registos de dados e blocos de 8k, M=228 e a profundidade máxima é 4.

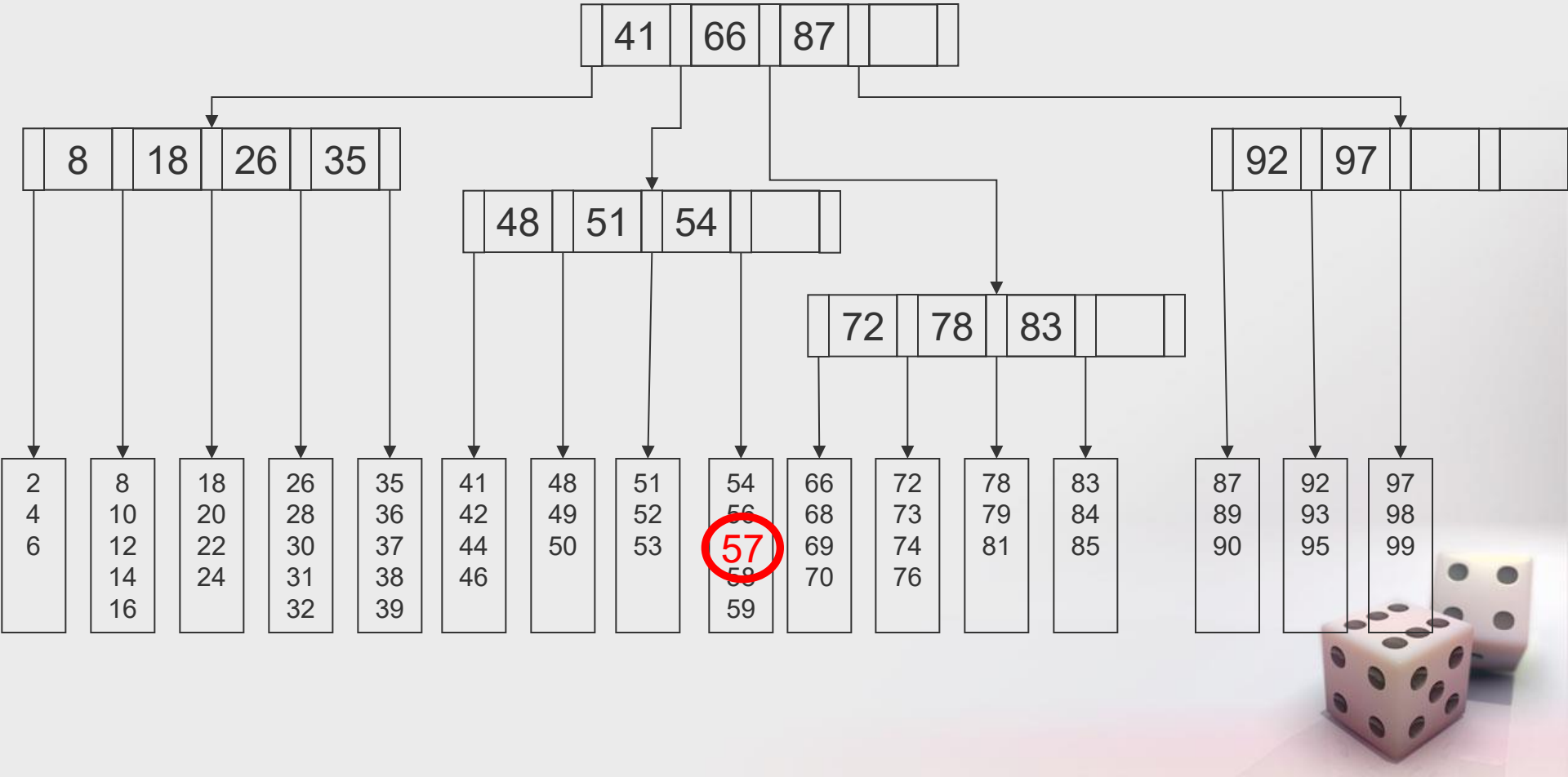
# B-Tree

- Inserção: Inserir registo com chave 57



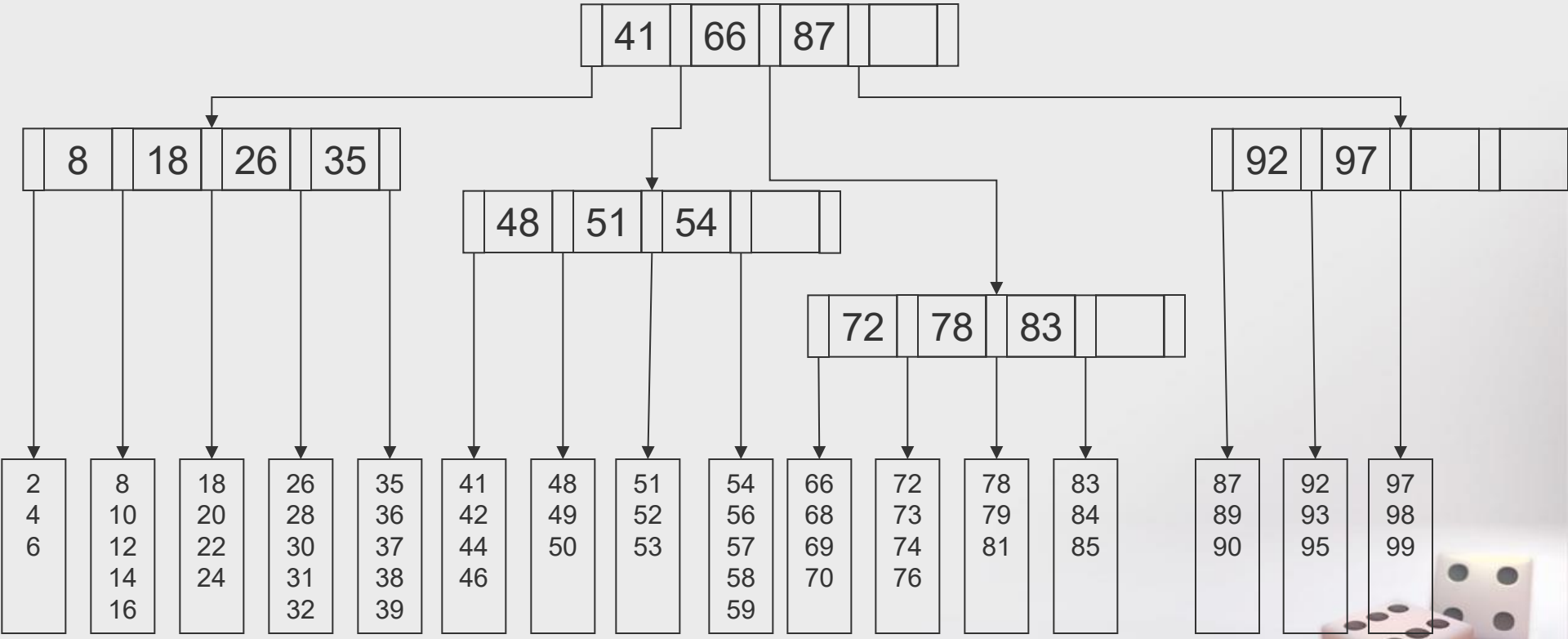
# B-Tree

- Inserção: Inserir registo com chave 57



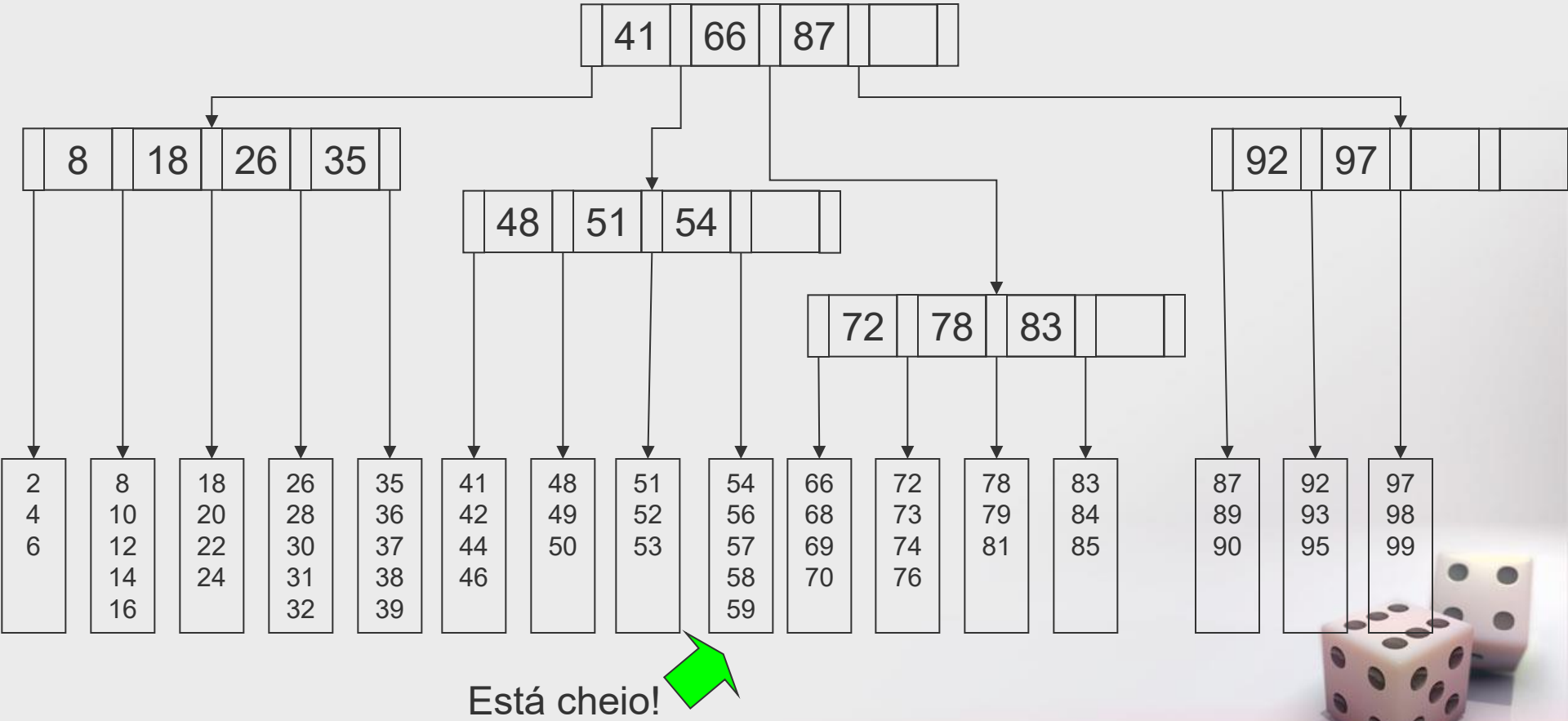
# B-Tree

- Inserção: Inserir registo com chave 55



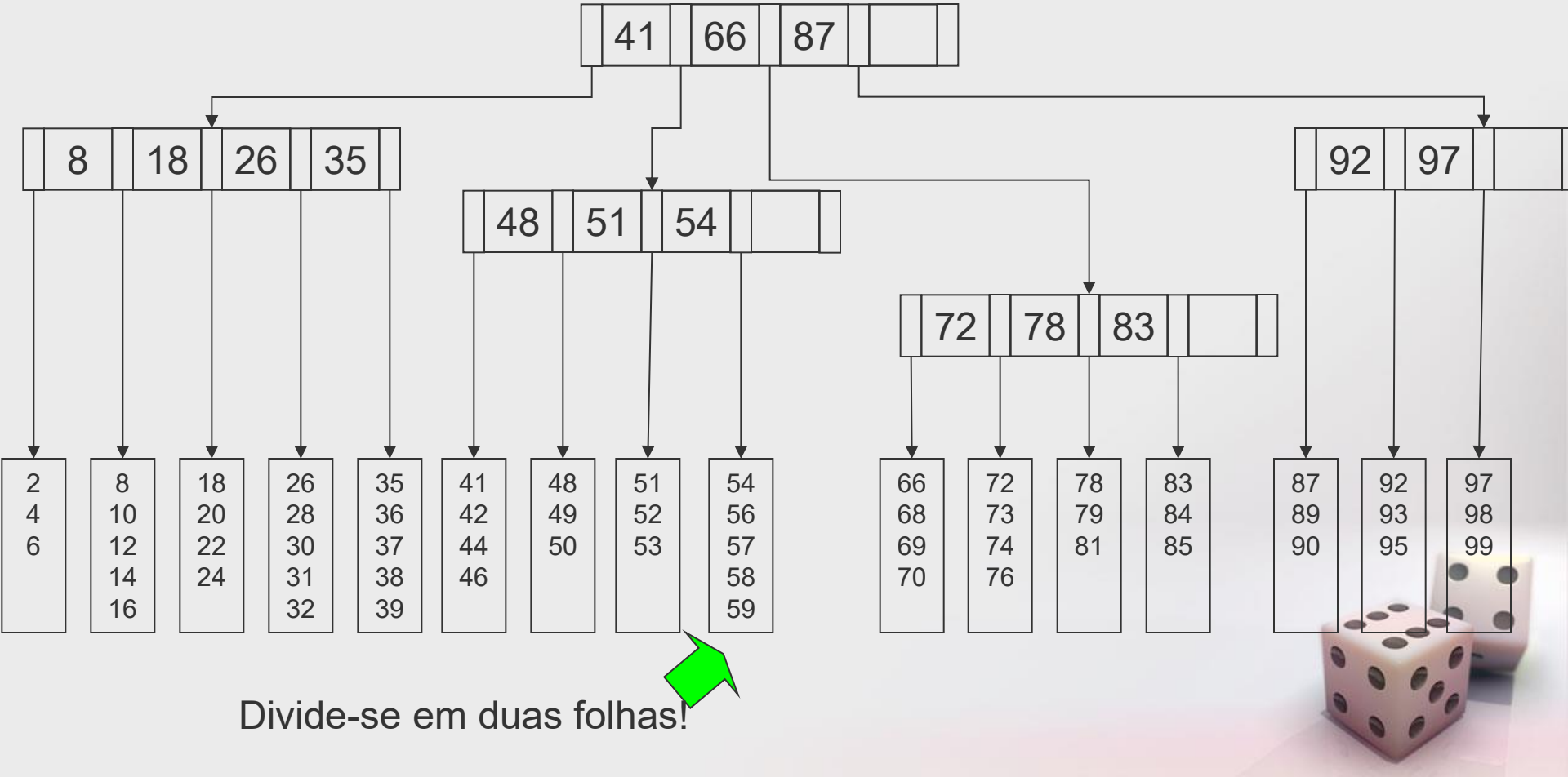
# B-Tree

- Inserção: Inserir registo com chave 55



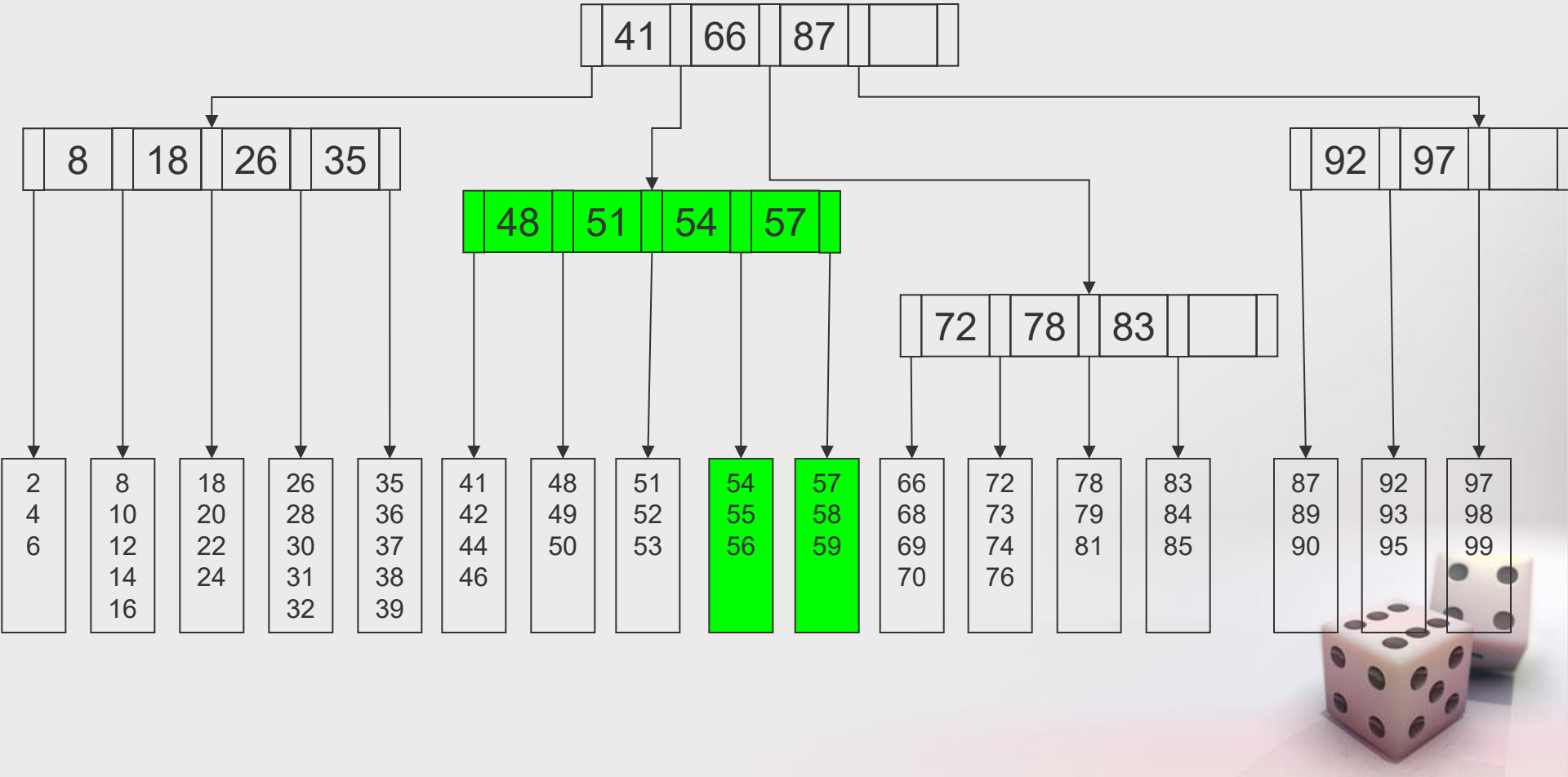
# B-Tree

- Inserção: Inserir registo com chave 55



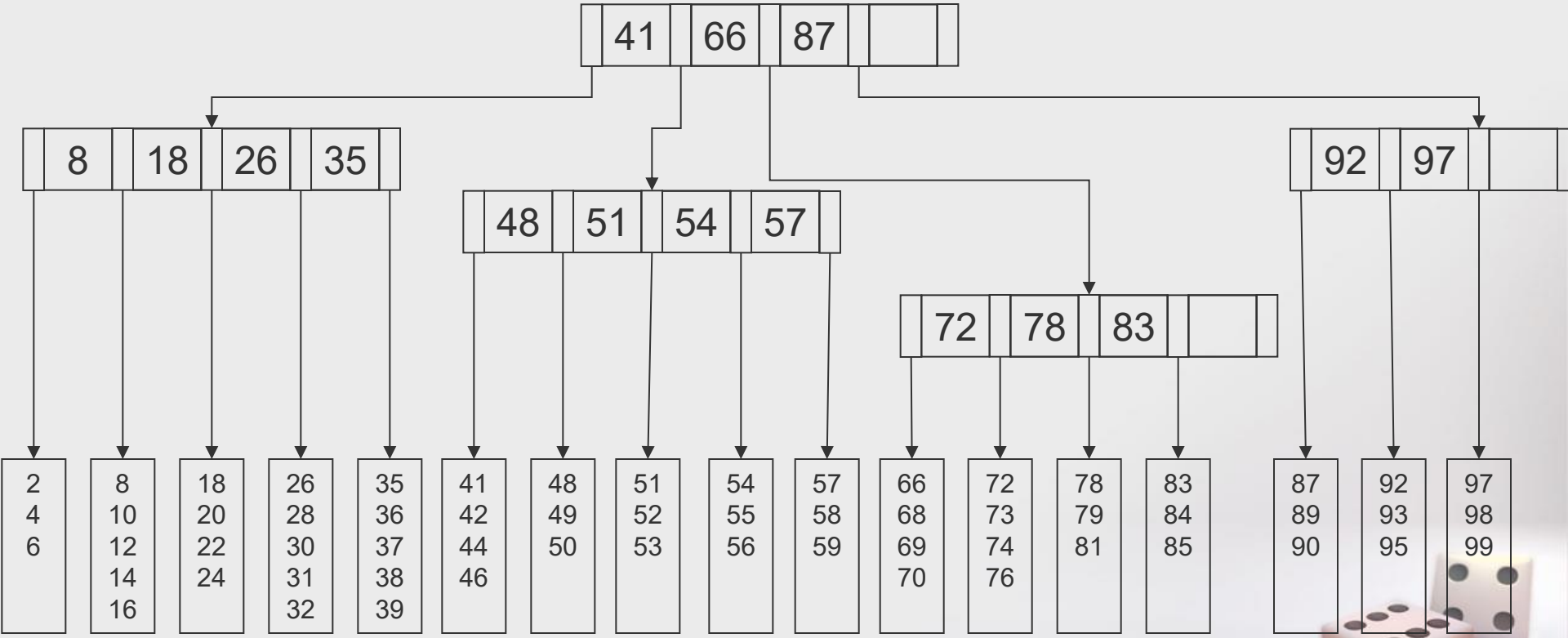
# B-Tree

- Inserção: Inserir registo com chave 55



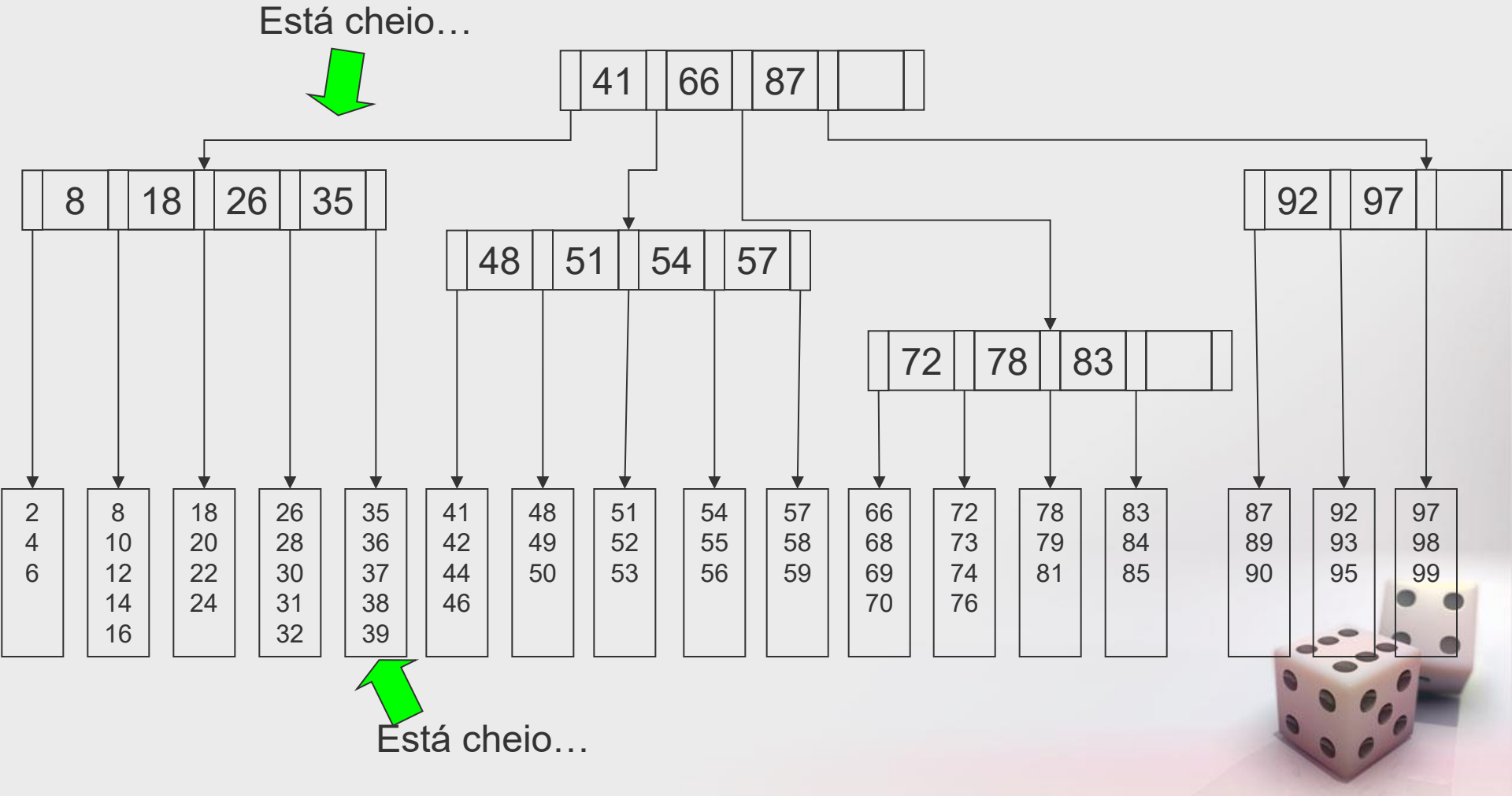
# B-Tree

- Inserção: Inserir registo com chave 40



# B-Tree

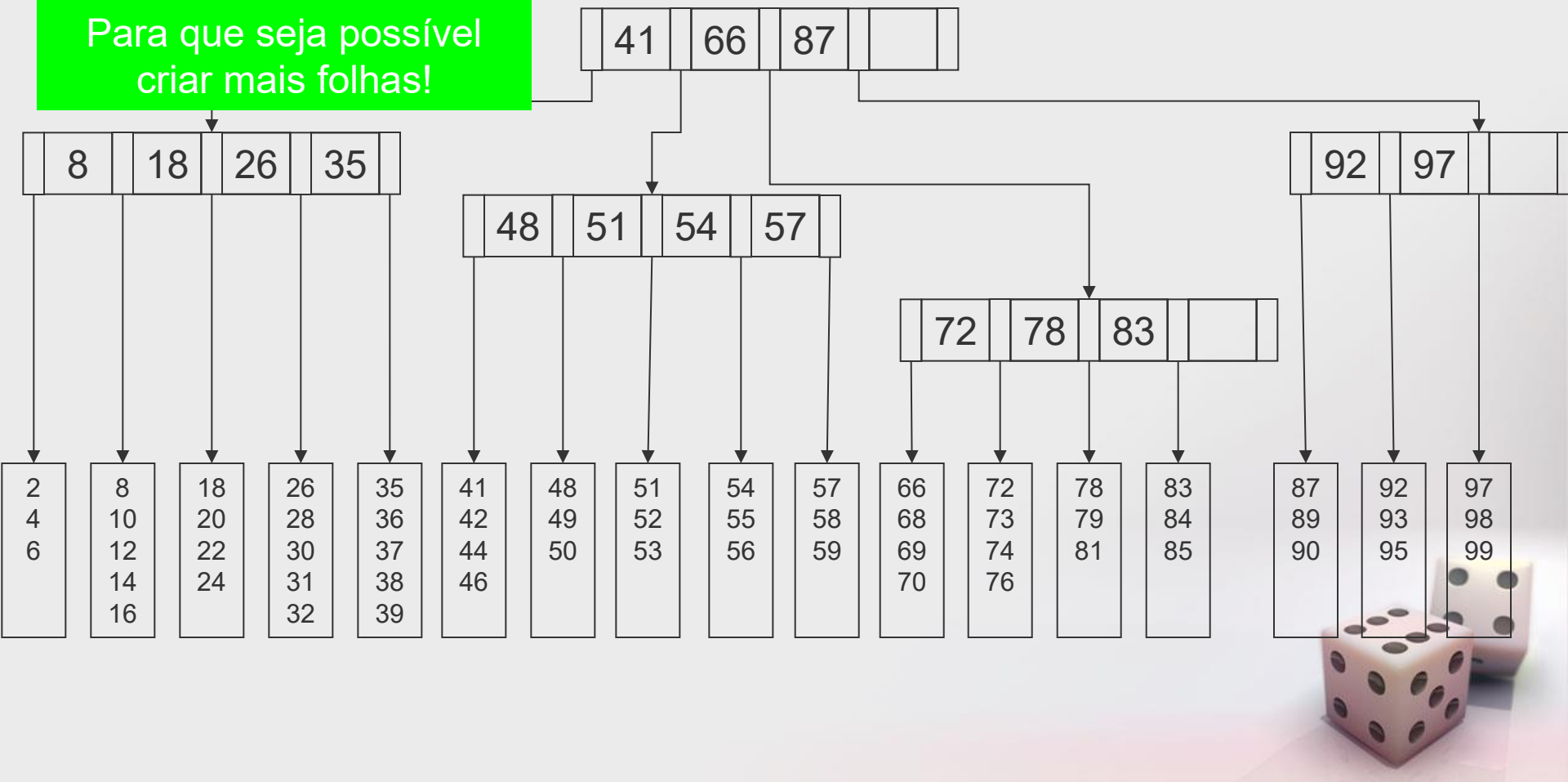
- Inserção: Inserir registo com chave 40



# B-Tree

- Inserção: Inserir registo com chave 40

Vamos dividir o antecessor
Para que seja possível criar mais folhas!

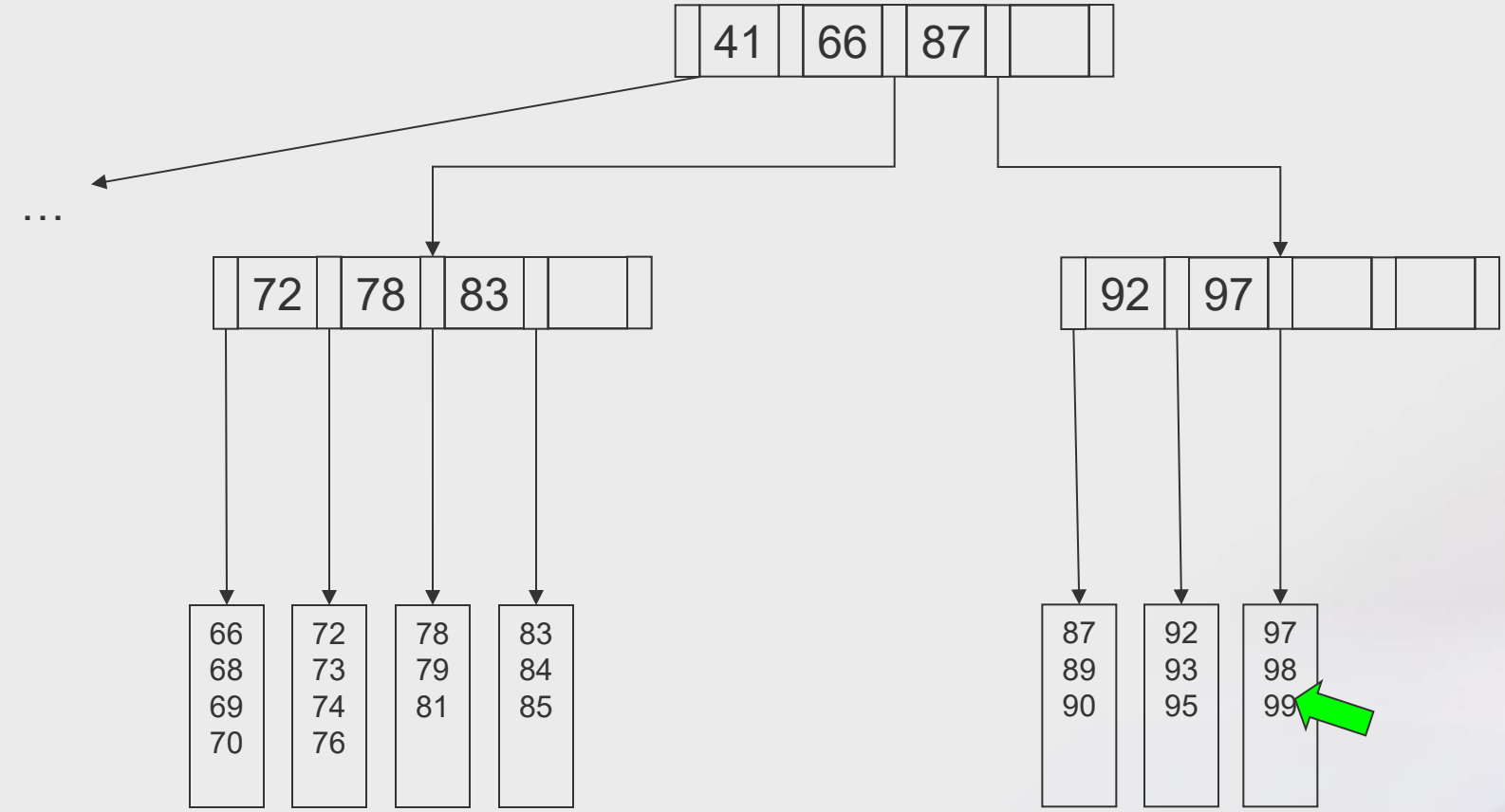


# B-Tree

- Inserção: Inserir registo com chave 40

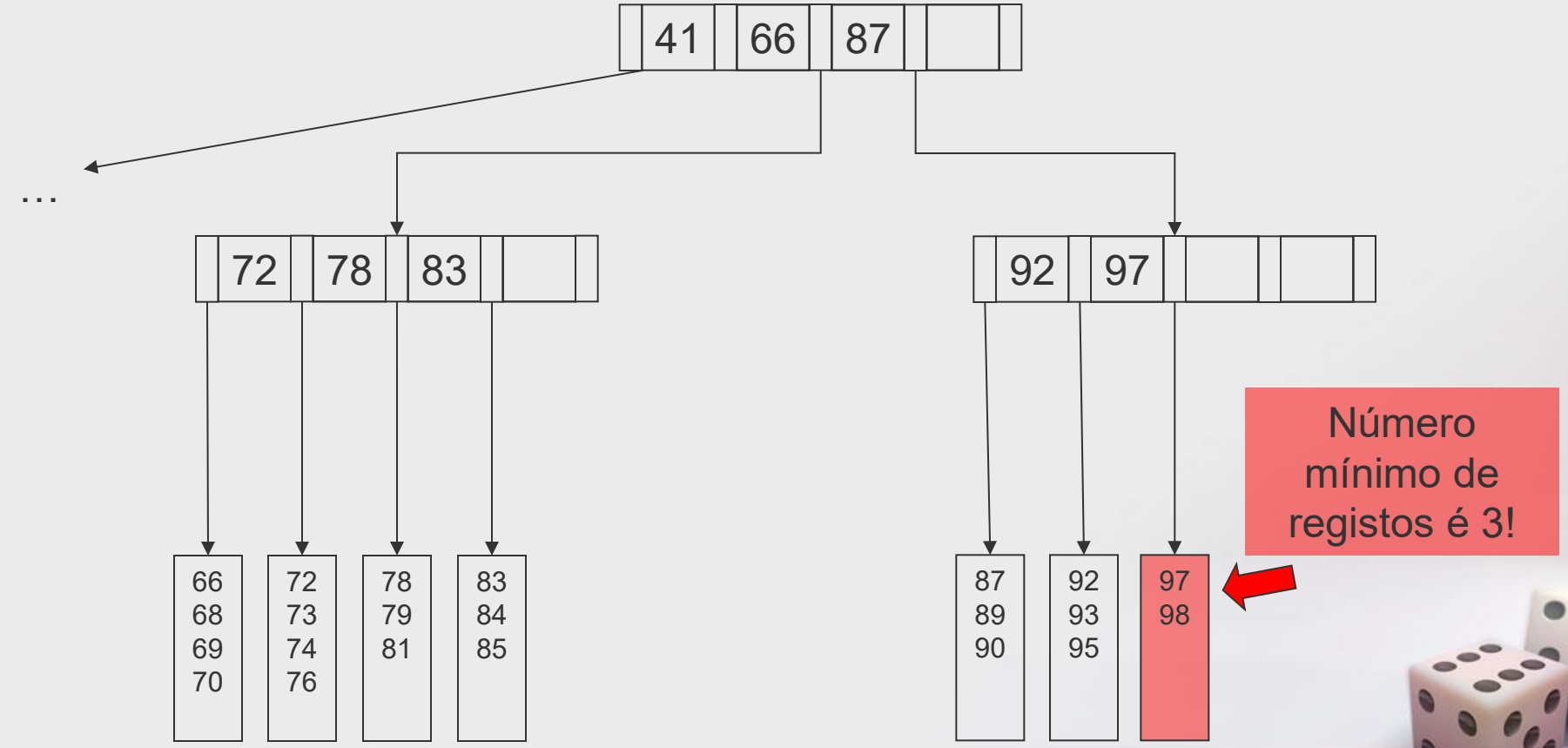
# B-Tree

- Remoção: Apagar 99



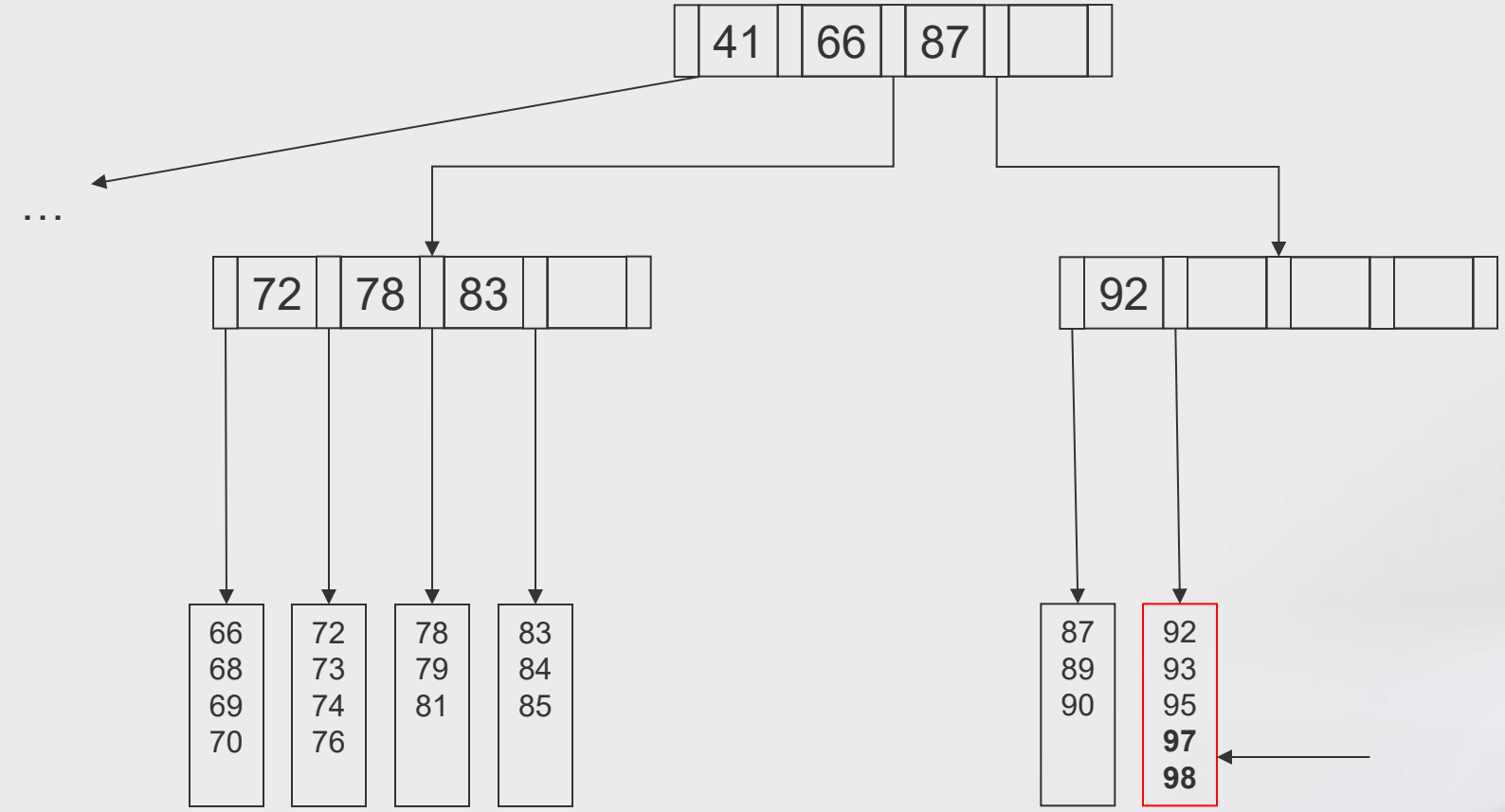
# B-Tree

- Remoção: Apagar 99



# B-Tree

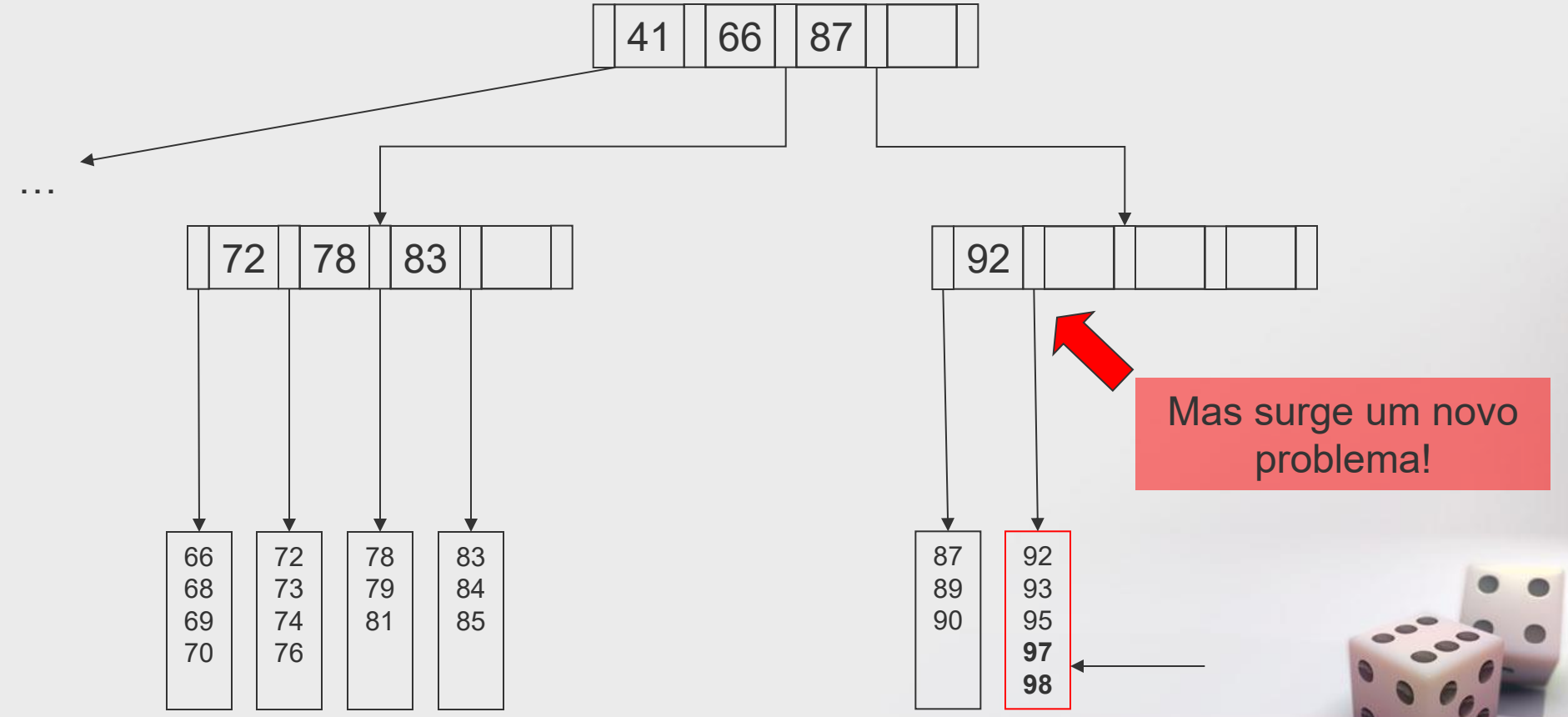
- Remoção: Apagar 99




É possível combinar com um vizinho…

# B-Tree

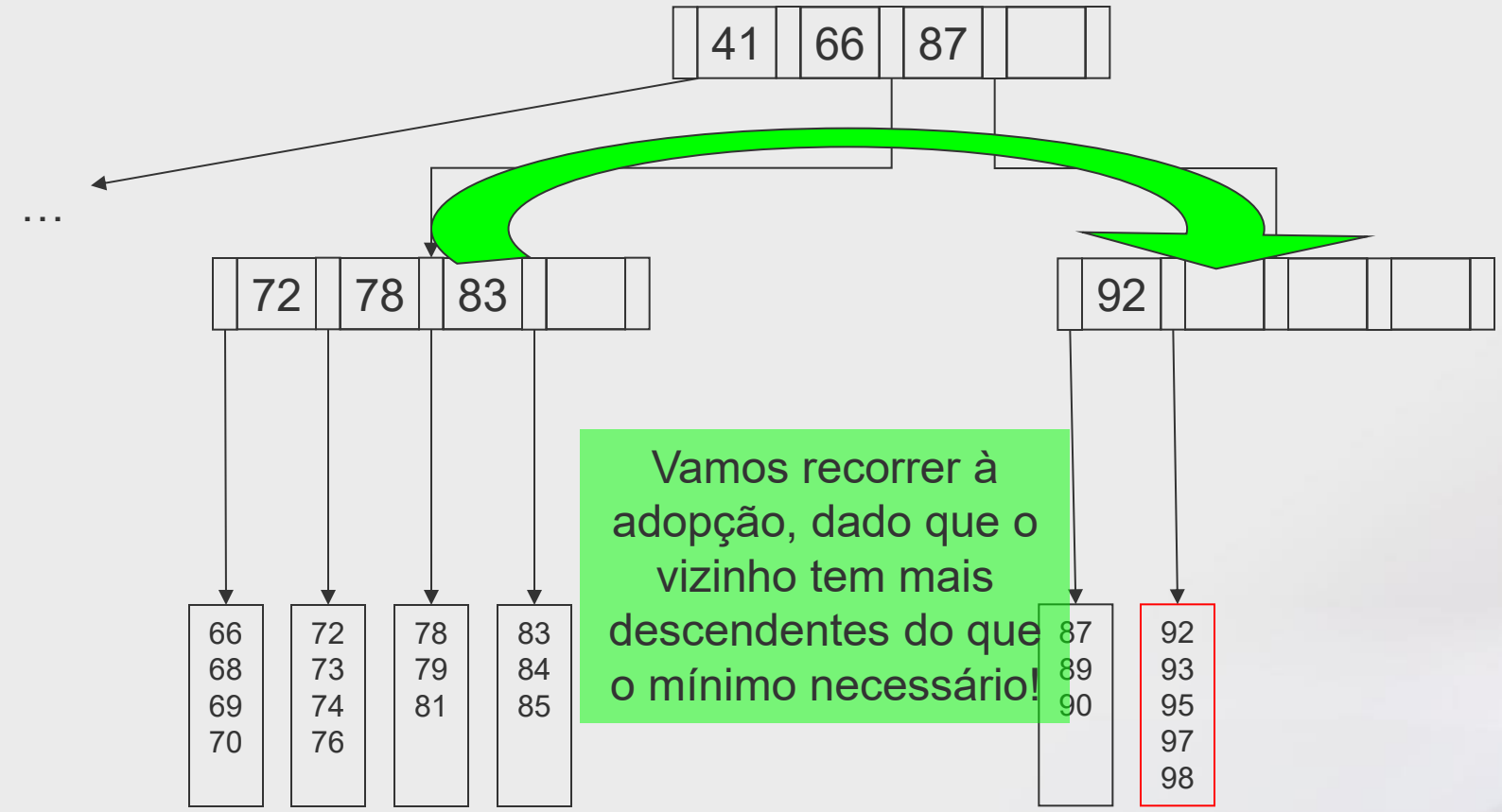
- Remoção: Apagar 99




É possível combinar com um vizinho…

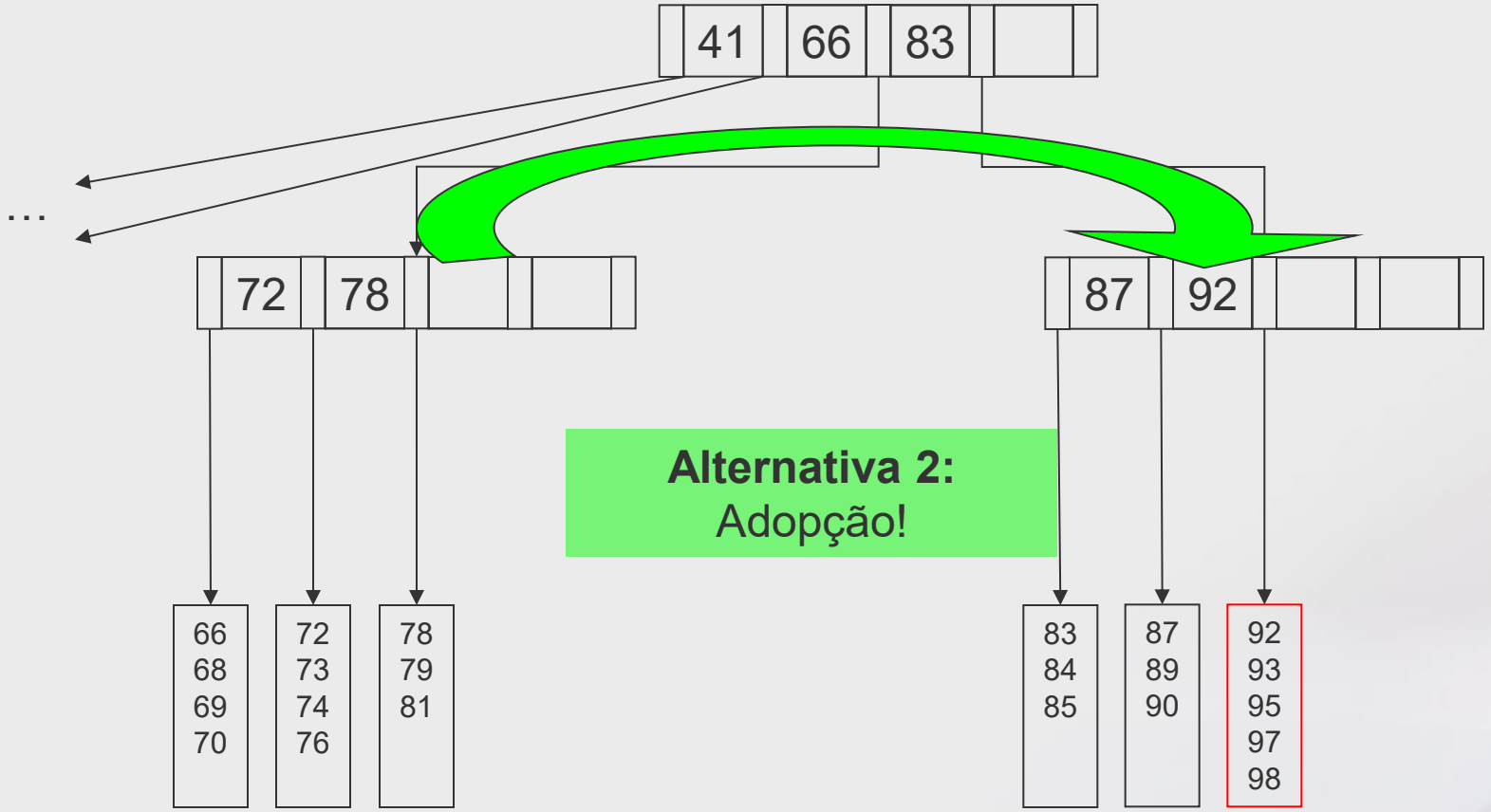
# B-Tree

- Remoção: Apagar 99



# B-Tree

- Remoção: Apagar 99



# Estruturas de Dados Estáticas

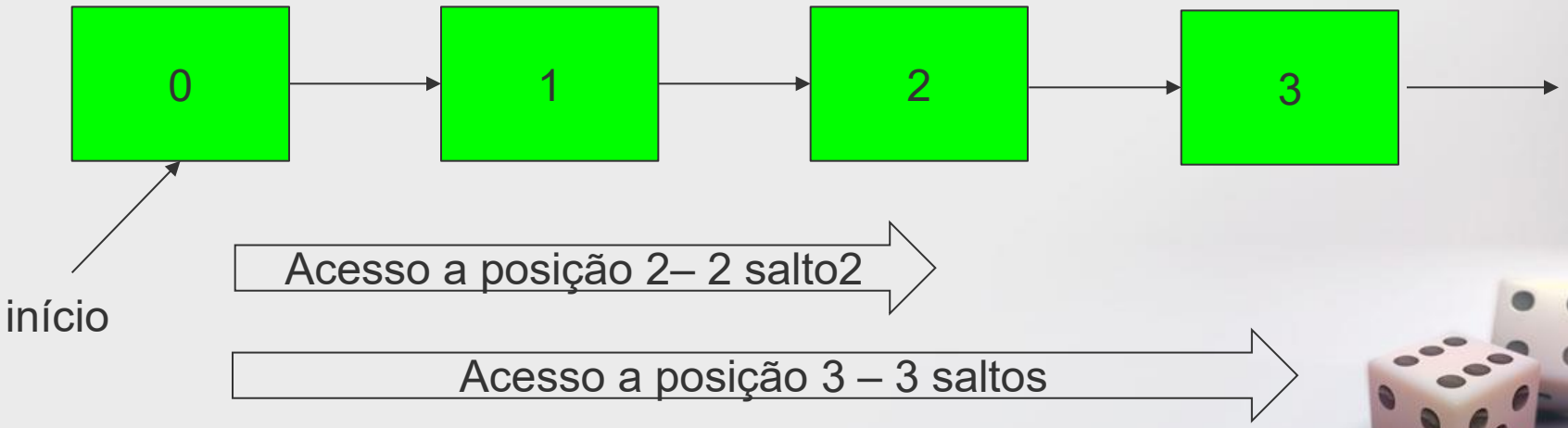
- As árvores de pesquisa equilibradas que estudámos são estáticas:
  - O tempo de pesquisa de um determinado elemento depende da sua profundidade na árvore.
  - A profundidade é estática, ou seja, é determinada no momento de inserção e não varia nunca mais.
  - **Não oferecem vantagens para padrões de acesso comuns:**
    - **Regra 90-10 : 90% dos acessos são a 10% dos dados**
    - **Acesso sequencial**

# Estruturas de Dados Dinâmicas

- As estruturas de dados dinâmicas (ou auto-organizadas) pretendem dar resposta a este problema:
  - A organização interna dos dados é modificada de acordo com os padrões de pesquisa para maximizar o desempenho.
  - Vamos analisar duas estruturas de dados dinâmicas:
    - Listas auto-organizadas – *Usadas para implementar conjuntos.*
    - Splay Trees

# Lista Auto-Organizada

- O acesso aos elementos de uma lista próximos das localizações conhecidas (início, fim e último acesso) é mais eficiente do que o acesso a elementos mais remotos.



# Lista Auto-Organizada

- Idealmente, os nodos deveriam estar organizados em ordem de probabilidade de acesso
- Várias estratégias podem ser adoptadas para atingir este fim:
  - ***MTF*** – *Move to Front* – Copia cada elemento acedido para a cabeça da lista.
  - **Transposição** – Troca um elemento acedido com o elemento anterior.
  - ***Contagem*** – O número de acessos é armazenado em cada nodo, e são efectuadas trocas quando tal é necessário.

Mais estável

Mais reactivo

# Lista Auto-Organizada

- Desempenho:
  - Se p(i) é a probabilidade de aceder ao nodo na posição i, então o número médio de saltos pode ser calculado por

$$S = p(0)*0+p(1)*1+p(2)*2+...$$

# Lista Auto-Organizada

- Desempenho:

  Se *p(0)=0.1 p(1)=0.3 p(2)=0.5 e p(3)=0.1*, então

  **S=1.6**

  Caso os nodos estivessem na ordem ideal, então *p(0)=0.5 p(1)=0.3 p(2)=0.1 e p(3)=0.1, e*

  **S=0.8**

  Neste caso, o benefício máximo que se poderia esperar através da utilização de uma lista auto organizada seria de 50%.

# Splay Tree

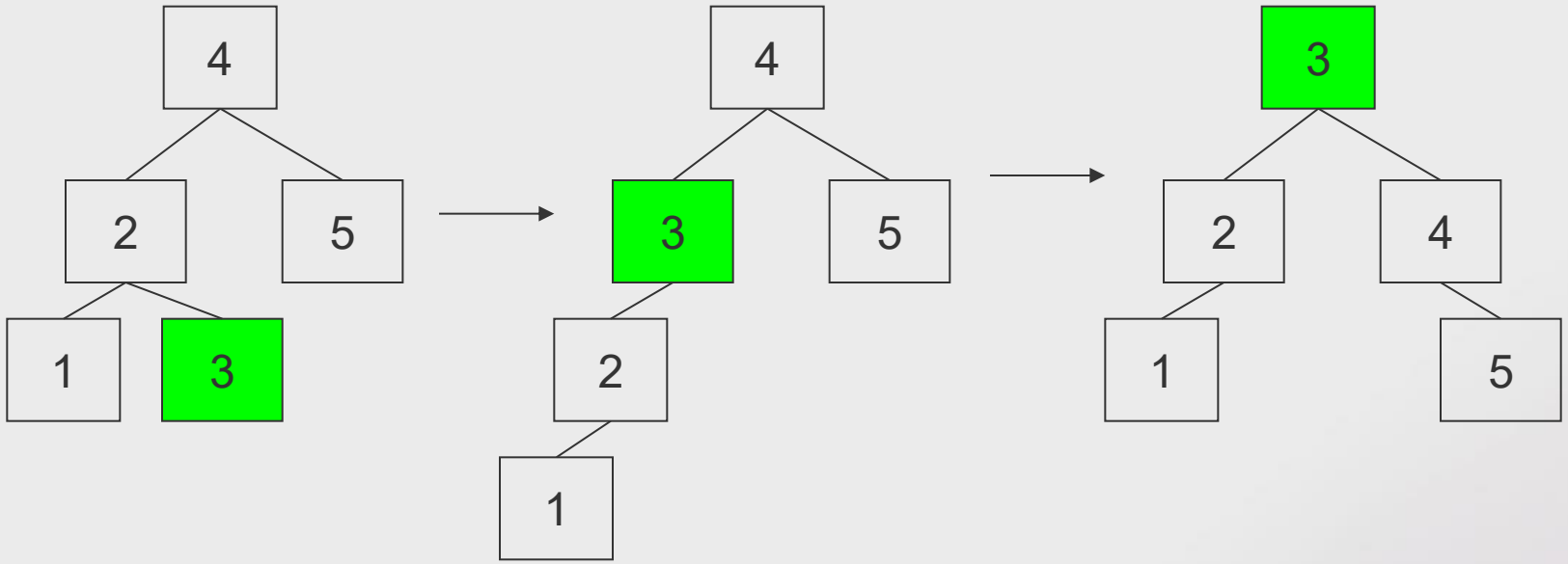
- As Splay Trees são árvores com comportamento dinâmico
  - Qual é o desempenho?
    - Garantia de desempenho O(N) no pior caso.
    - Mas têm-se garantia de desempenho O(log N) *amortizado*.

# Splay Tree

- Ideia Básica: Ajuste dinâmico da estrutura da árvore:
  - Tentativa: Mover nodos acedidos para raiz raiz.
    - *rotação-para-raiz*

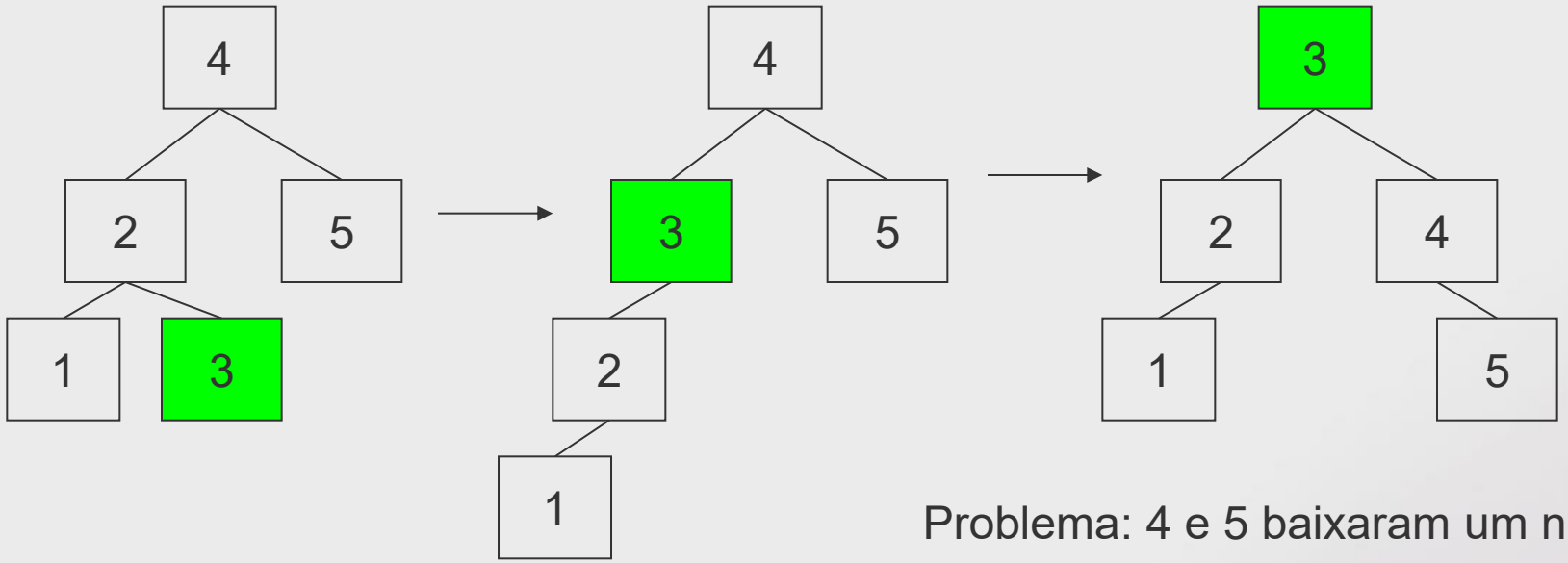
# Splay Tree

- Rotação para a raiz: Acesso a 3



# Splay Tree

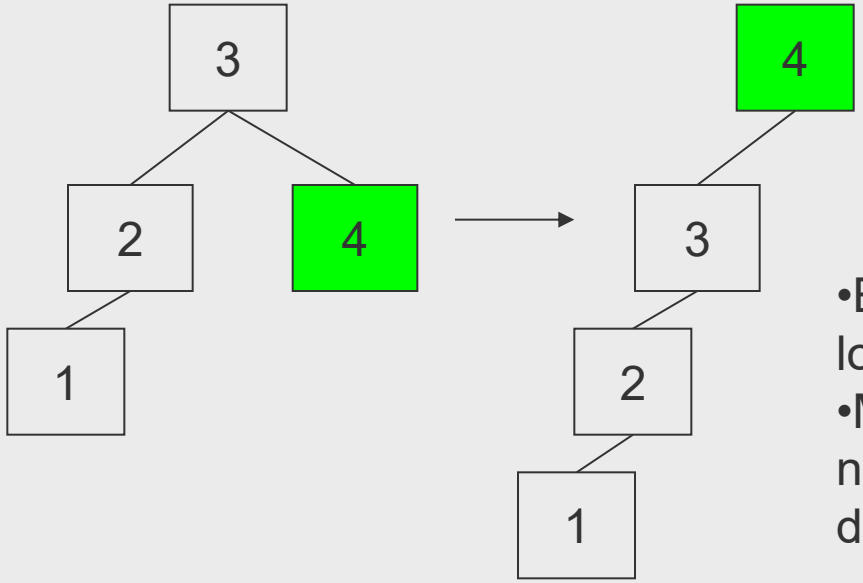
- Rotação para a raiz: Acesso a 3




Problema: 4 e 5 baixaram um nível

# Splay Tree

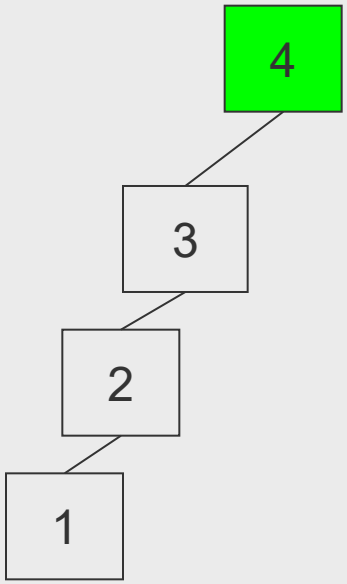
- Rotação para a raiz: Inserção de 4




•Esta estratégia funciona bem quando a localidade é elevada (90/10 verifica-se)
•Mas pode funcionar muito mal se isso não se verificar (i.e: não oferece garantias de custo amortizado)

# Splay Tree

- Rotação para a raiz: Acesso 1,2,3,4




•Complexidade quadrática!! $O(N^2)$

# Splay Trees

- O processo básico de rotação para raiz pode ser modificado ligeiramente com uma técnica chamada *splaying* para assegurar garantias de desempenho amortizado!

*splay: espalhar* ou *dispersar*

# Splay Trees

- Mantém-se as rotações desde o nodo acedido até à raiz, mas observam-se os seguintes casos:
  - Se X é um dos nodos no percurso até raiz e é descendente de raiz, então roda-se normalmente:

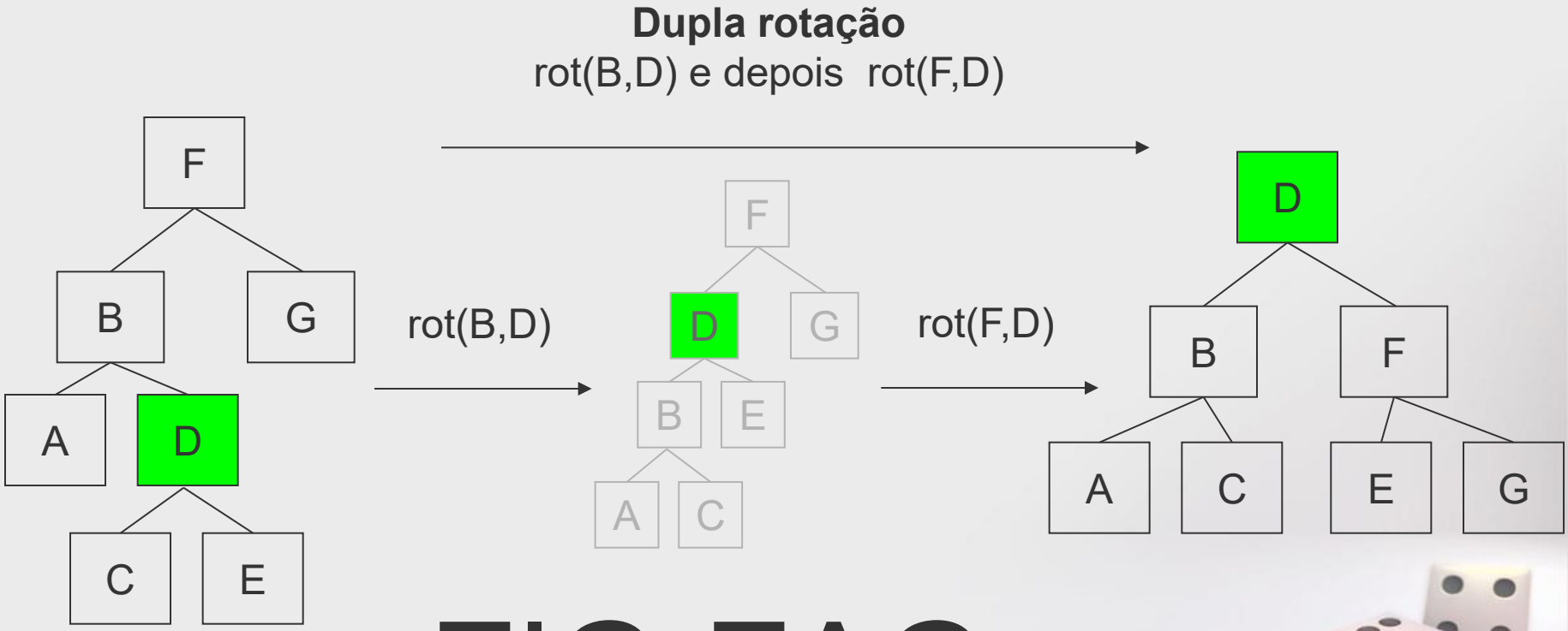
# Splay Tree

- Caso 1: Acesso a Descendente da Raiz


ZIG!

# Splay Tree

- Caso 2: Nodo que não é descendente da raiz:
  - Nodo interior




ZIG-ZAG

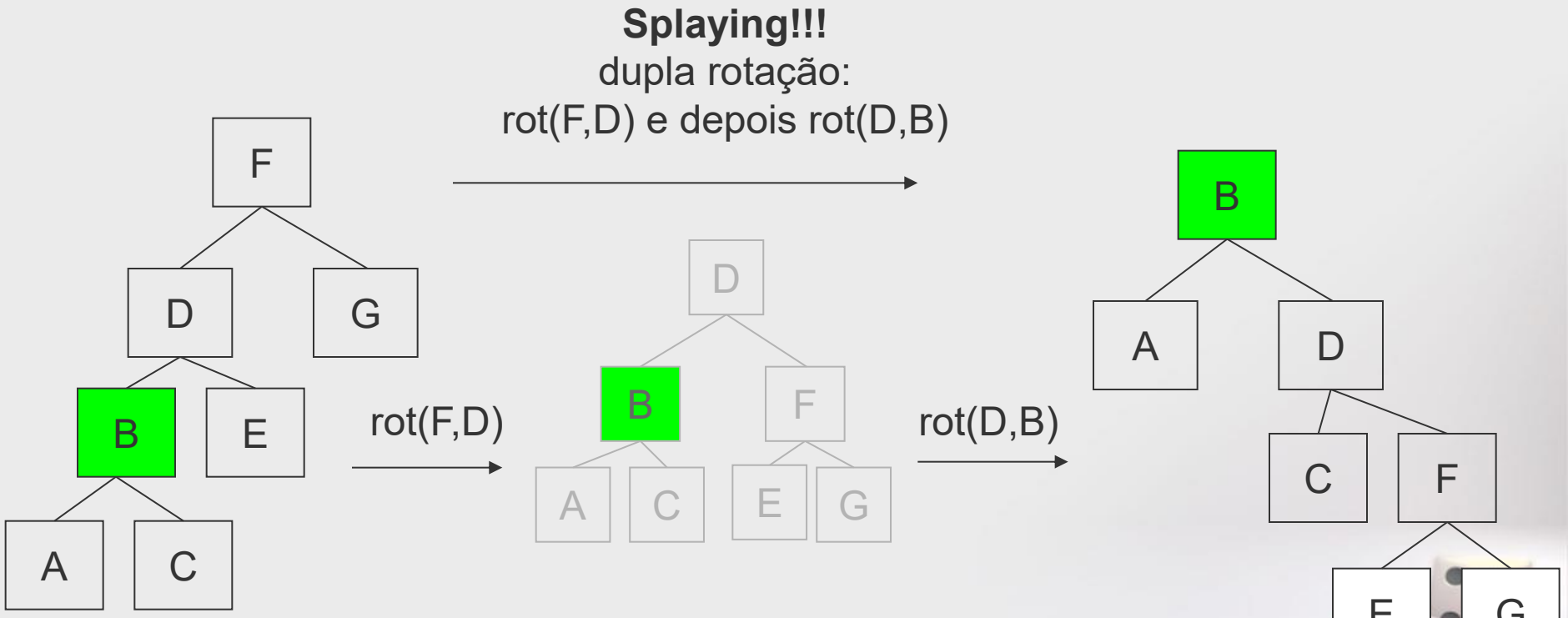
# Splay Trees

- Tanto o caso Zig como o Zig-Zag são idênticos ao que acontece com uma rotação para a raiz tradicional…

  Felizmente, ainda não vimos os casos todos!

# Splay Tree

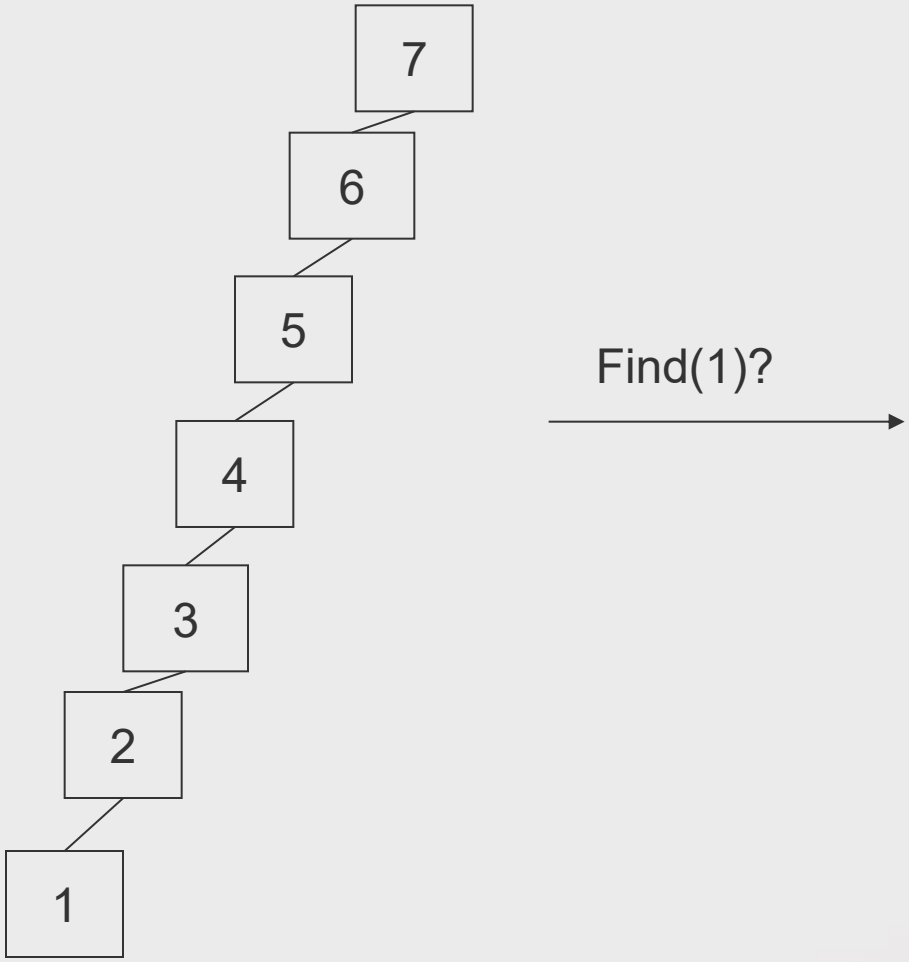
- Caso 2: Nodo não é descendente da raiz:
  - Nodo exterior




ZIG-ZIG!

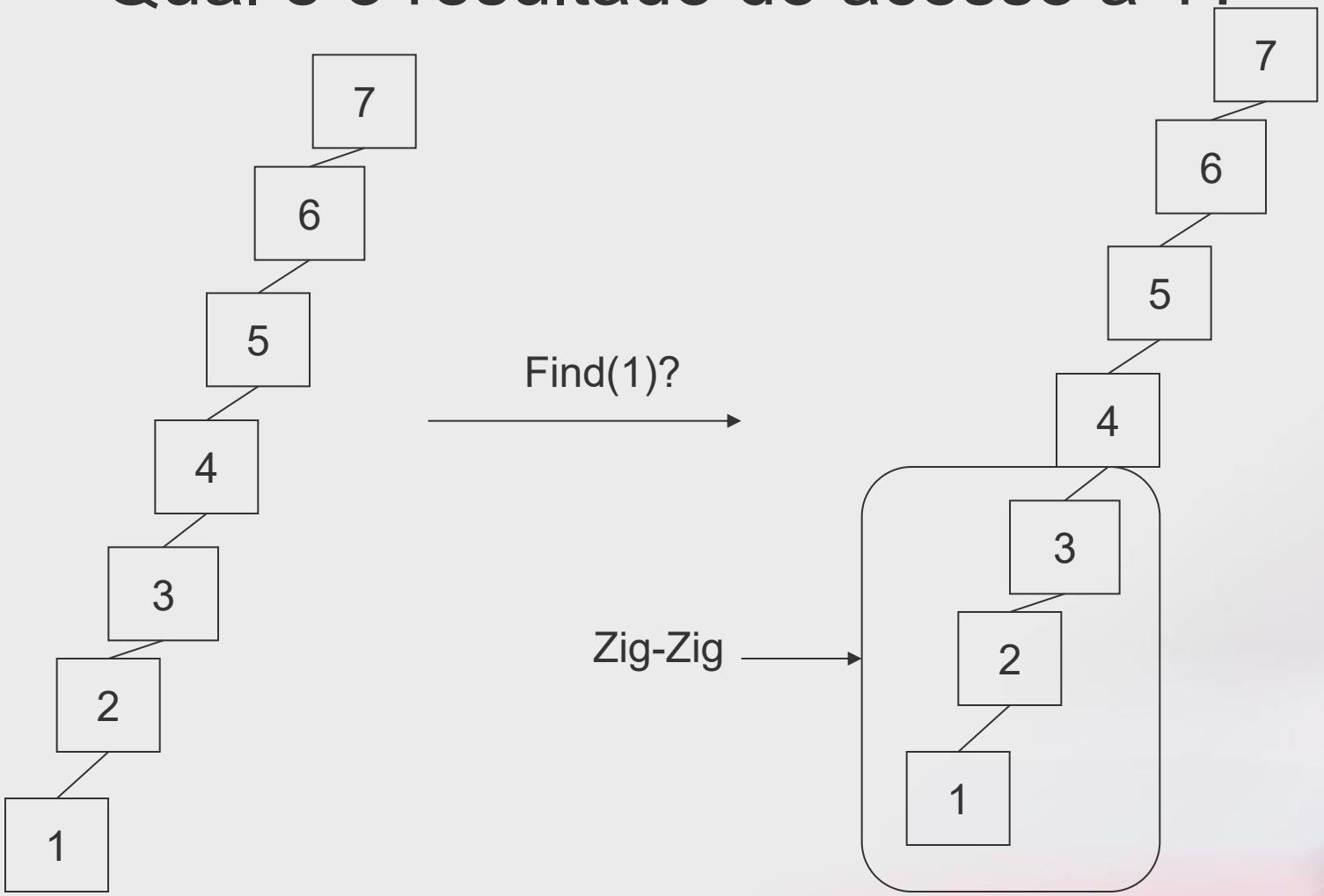
# Splay Tree

- Qual é o resultado do acesso a 1?



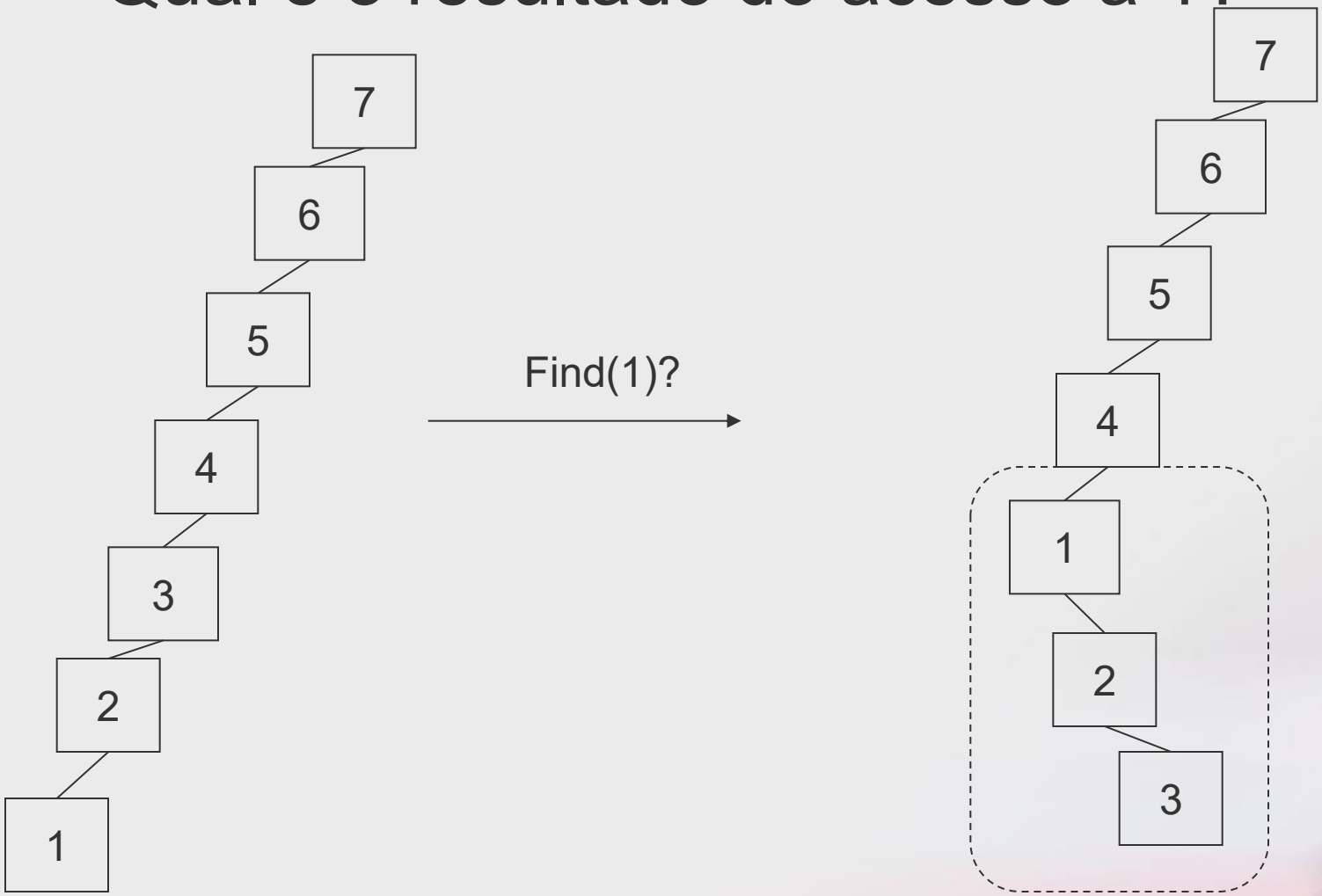
# Splay Tree

- Qual é o resultado do acesso a 1?



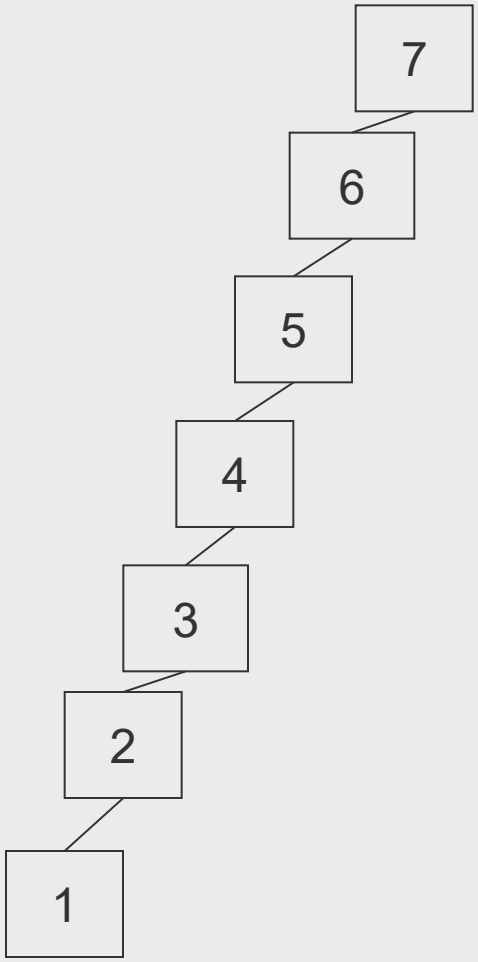
# Splay Tree

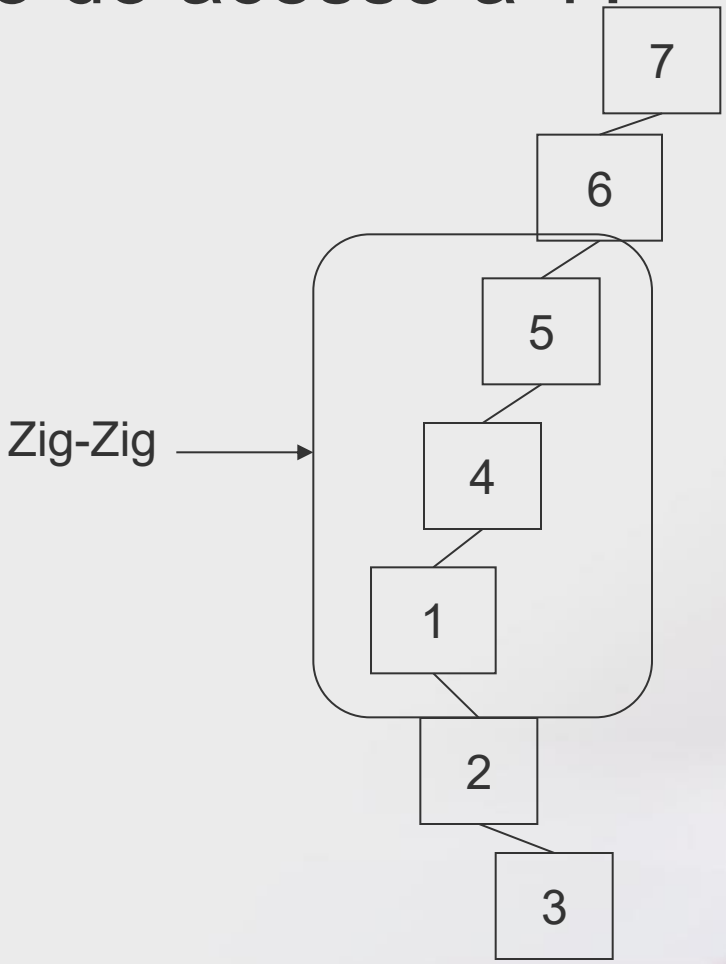
- Qual é o resultado do acesso a 1?



# Splay Tree

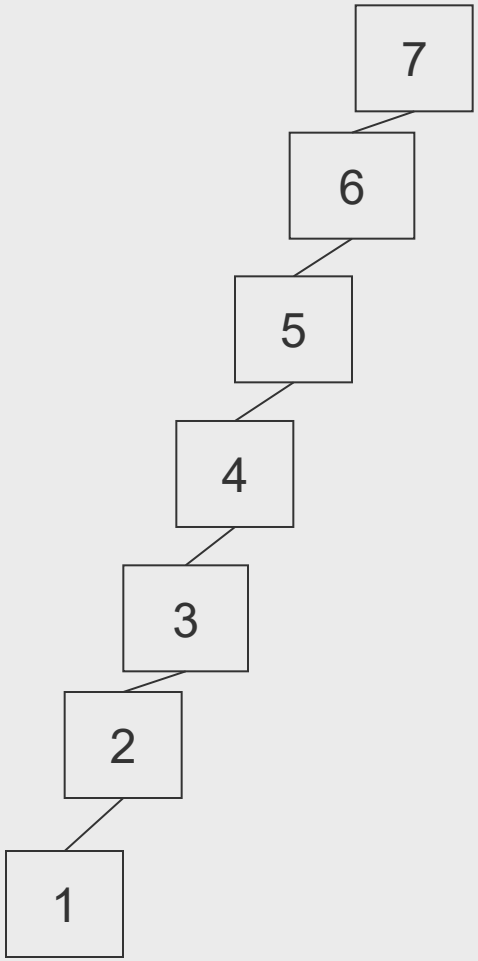
- Qual é o resultado do acesso a 1?

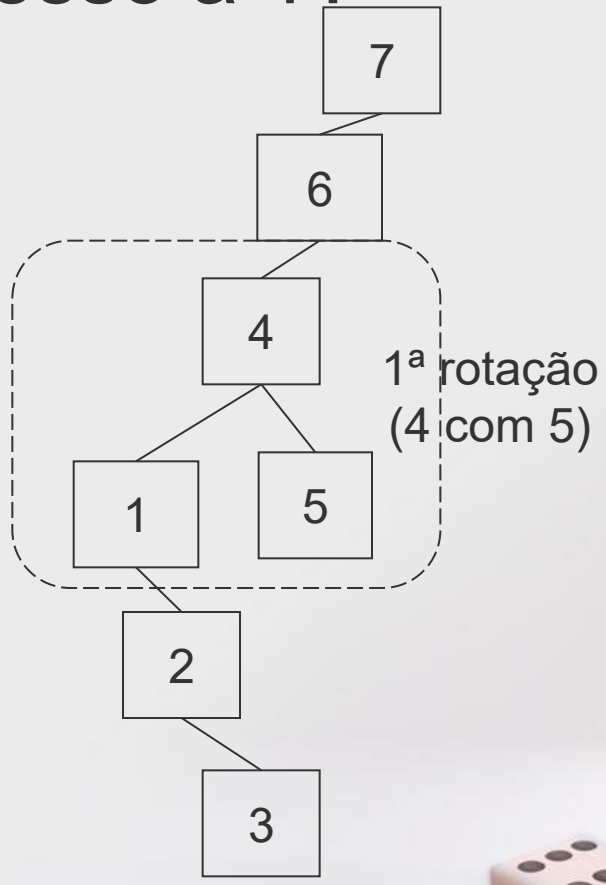



# Splay Tree

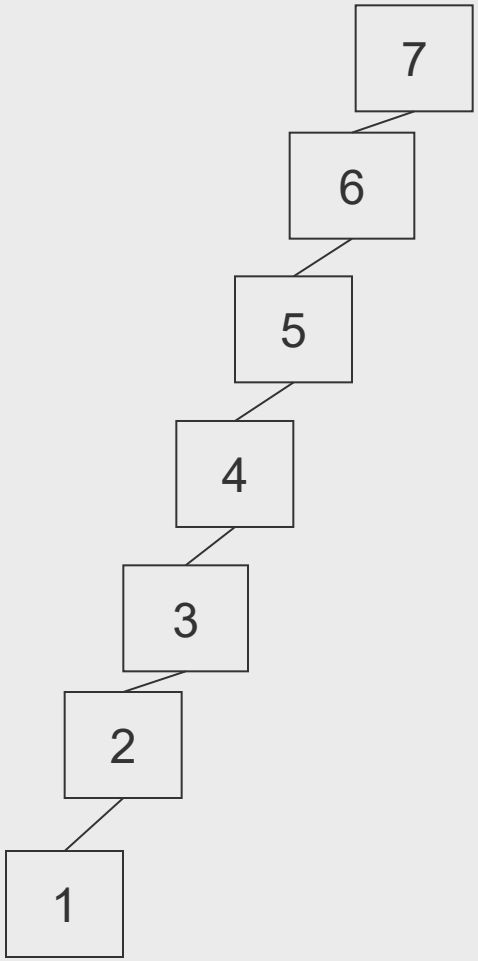
- Qual é o resultado do acesso a 1?

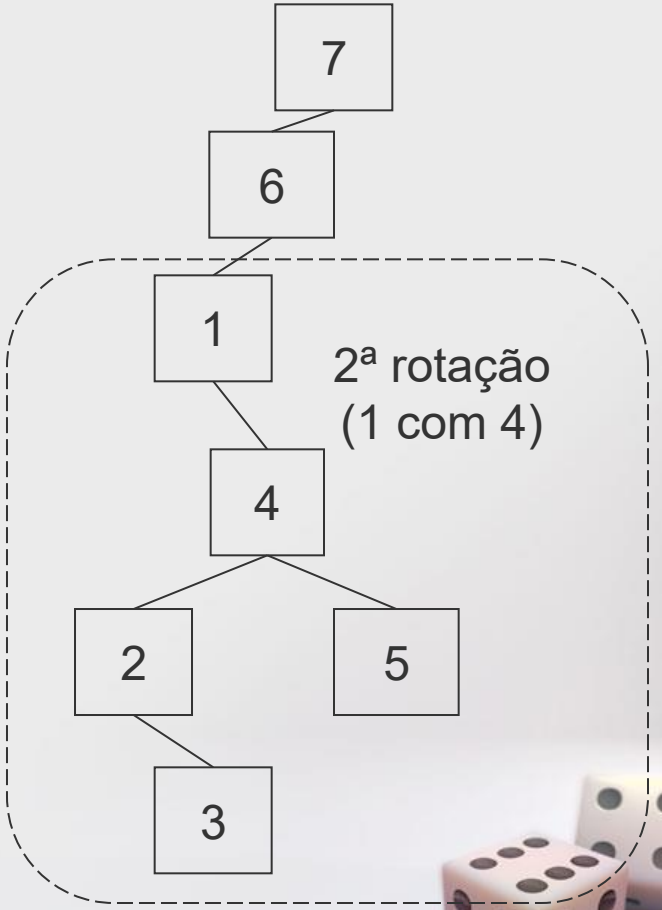


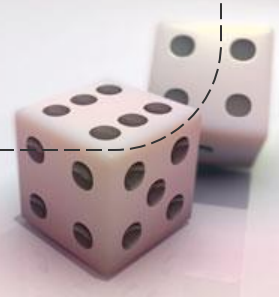




# Splay Tree

- Qual é o resultado do acesso a 1?

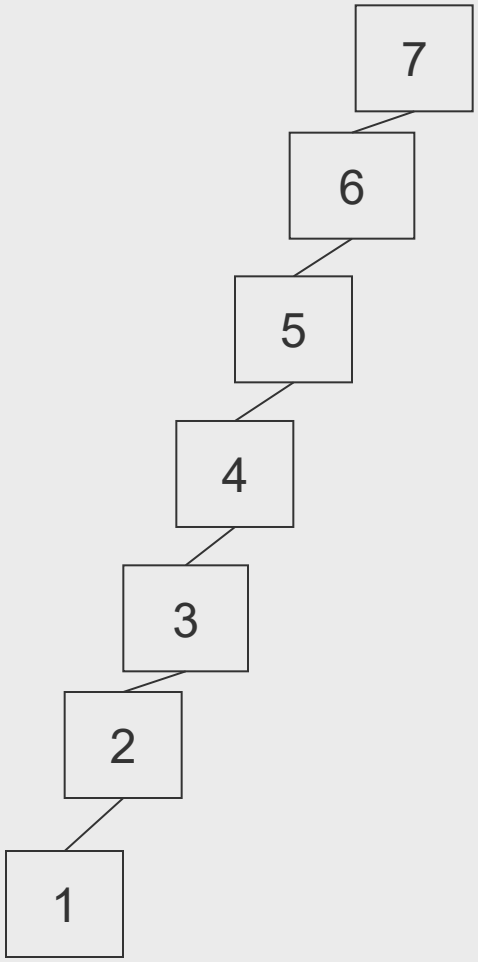


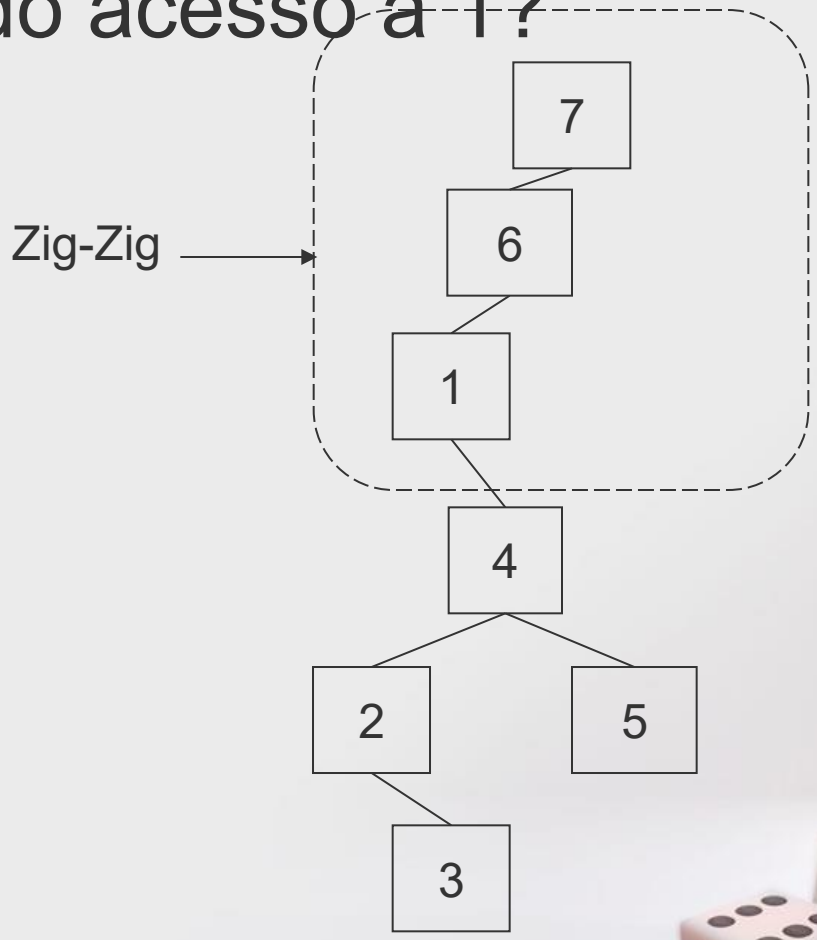



# Splay Tree

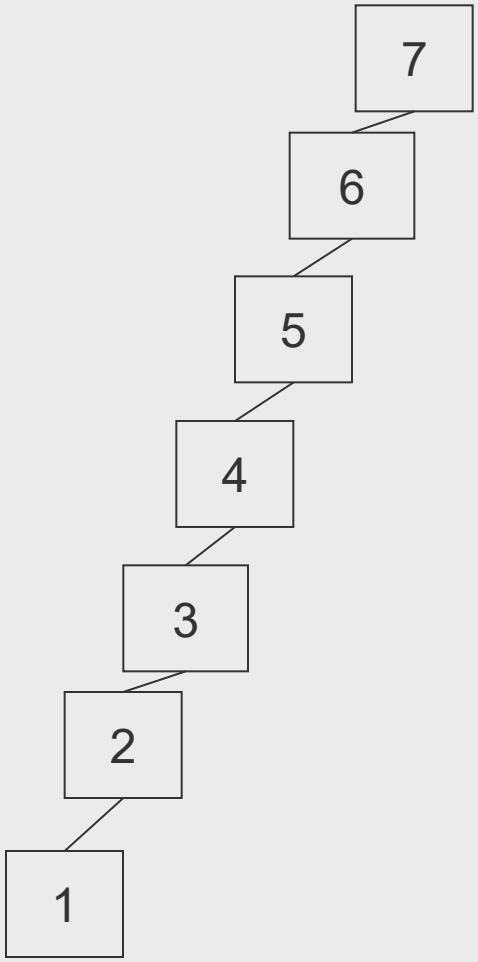
- Qual é o resultado do acesso a 1?

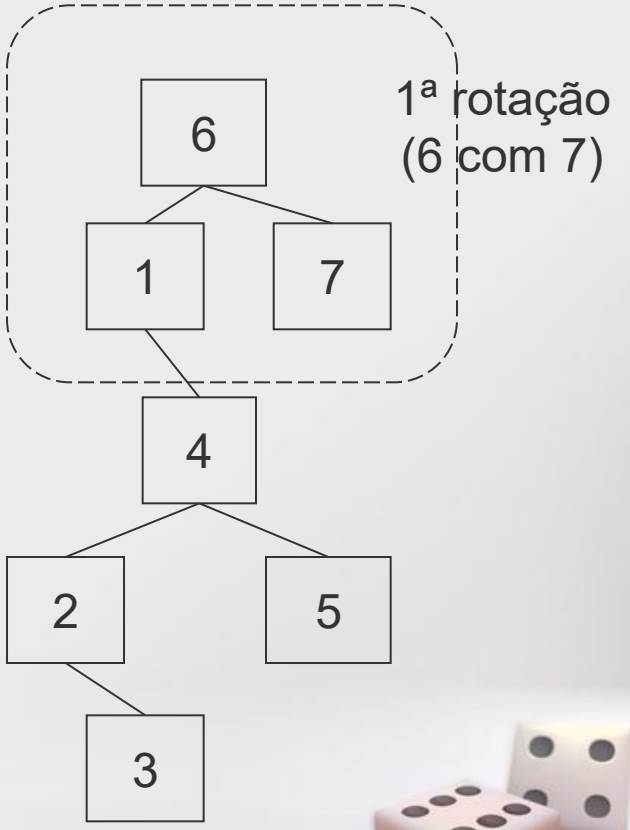



# Splay Tree

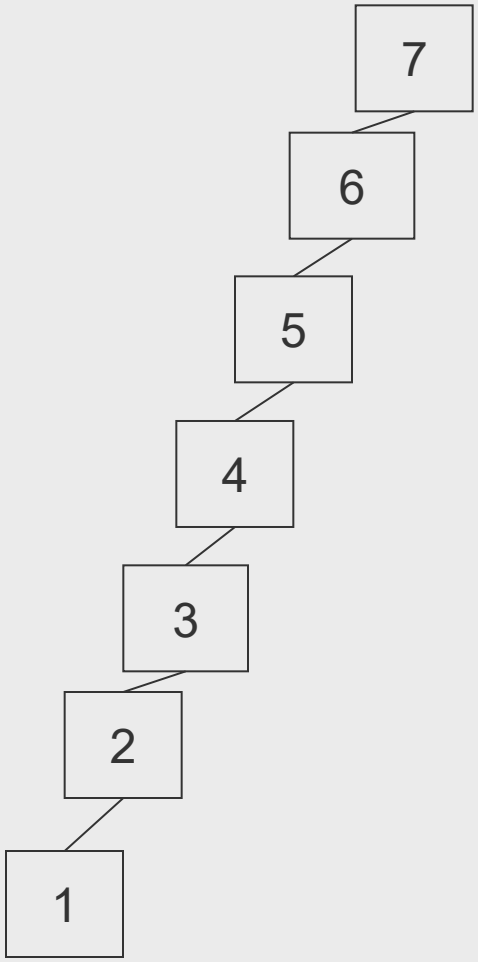
- Qual é o resultado do acesso a 1?

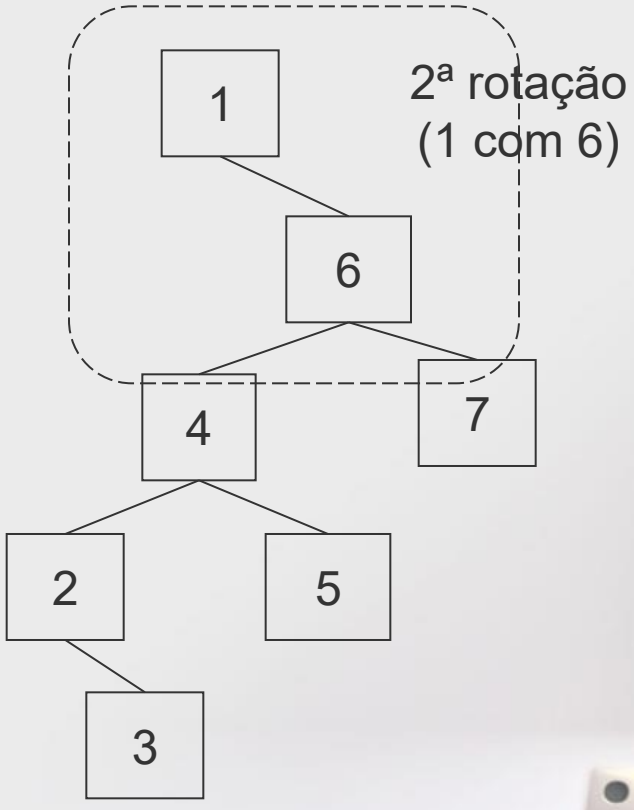



# Splay Tree

- Qual é o resultado do acesso a 1?




# Splay Tree

- Remoção?
  1. Acesso a nodo a remover (é colocado na raiz)
  2. É removido, sobrando as duas árvores descendentes desconexas (L e R).
  3. Procura-se o maior elemento da sub-árvore esquerda (é colocado na raiz)
  4. A sub-árvore direita é colocada como descendente direito da nova raiz.

# Splay Trees

- Propriedades
  - Consegue-se demonstrar, após análises laboriosas, que o custo amortizado de acesso é O(log N):
    - Há que ter em atenção que um desempenho amortizado pode não ser suficiente em muitas situações.
    - Mas em muitos casos, é perfeitamente adequado