Árvore B: Variações

Prof. André Backes | @progdescomplicada

Árvore B

- Uma árvore B é uma árvore *n*-ária usada como estrutura de armazenamento
  - Muito eficiente e flexível
  - Mantém propriedades de balanceamento mesmo após inserções e remoções
  - Provém busca a qualquer chave com poucos acessos a disco

Árvore B

- Problema com acessos a disco
  - Não é porque uma árvore B tem 3 níveis, que toda busca tenha que fazer 3 acessos a disco
  - Encontrar uma maneira de fazer um uso eficiente de índices que são muito grandes para serem armazenados inteiramente em memória principal (i.e., RAM)
- Objetivo
  - Encontrar uma maneira de diminuir o número médio de acessos a disco para pesquisa

Árvore B

- Diferentes variações foram propostas para aumentar a sua eficiência
  - Árvore B Virtual
  - Árvore B*
  - Árvore B+

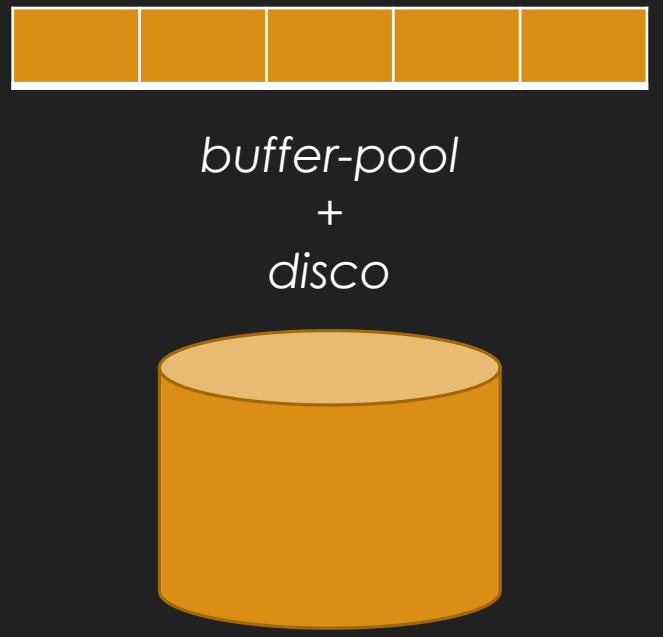
Árvore B Virtual

Árvore B Virtual

- Uma forma de melhorar o desempenho da árvore B
- Busca manter a página raiz da árvore em memória principal
  - Ainda deixa espaço disponível em RAM
  - Diminui o número de acessos a disco em 1 no pior caso

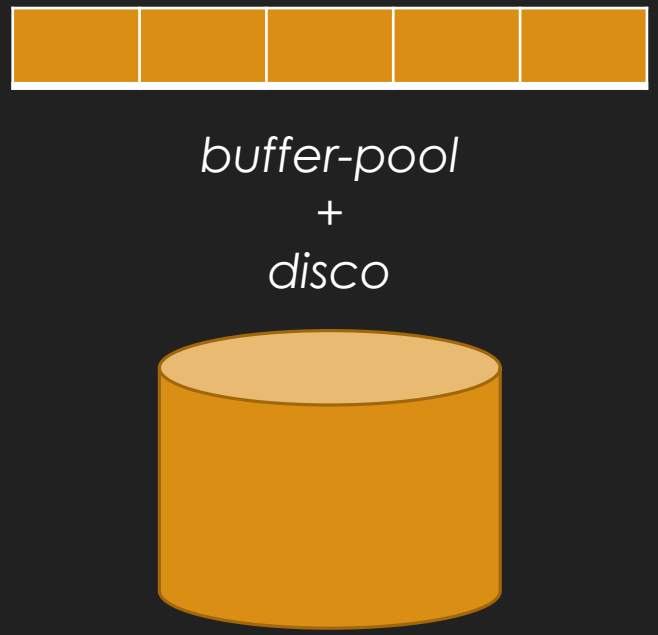
Árvore B Virtual

- Uso de um buffer de páginas (*buffer-pool*) para guardar um certo número de páginas da árvore B
  - Abordagem mais genérica
  - *Buffer-pool* fica em memória principal (i.e., em RAM)



Árvore B Virtual

- Funcionamento da busca
  - Primeiro procura a página no *buffer-pool* para evitar acessos a disco
  - Se a página não está em memória, ela é lida do disco para o *buffer*, substituindo alguma página previamente lida



Árvore B Virtual

- Há necessidade de substituição de páginas
  - ***Page Fault***: *processo de acessar o disco para trazer uma página que não está no buffer-pool*
- Causas
  - A página nunca foi utilizada
  - A página foi substituída no *buffer-pool* por outra página
- Decisão crítica
  - Qual página deve ser substituída no *buffer*, quando este encontra-se cheio?

Árvore B Virtual

- Necessidade de gerenciamento do buffer
  - A decisão mais crítica é qual página substituir quando o *buffer* se encontra cheio
- Quais páginas manter no Buffer?
  - Apenas a raiz
  - Política LRU (*least recently used*)
  - Substituição baseada na altura da página (*page height*)

LRU (*least recently used*)

- Uma estratégia bastante comum é substituir pela página menos recentemente usada
  - Substitui a página que foi acessada menos recentemente, isto é, a página que ficou mais tempo sem ser requisitada para uso
  - Baseia-se no fato de que é mais comum necessitar de uma página que foi recentemente usada do que de uma página que foi usada a mais tempo

LRU (*least recently used*)

- Página menos recentemente usada é diferente de substituir pela página menos recentemente lida
  - O tempo é dado pela último uso da página, e não o momento em que ela foi lida
- Localidade temporal
  - Assume algum tipo de agrupamento no uso das páginas ao longo do tempo
  - hipótese pode não ser sempre válida, mas aplica-se bem nas árvores B

Substituição baseada na altura da página

- Modo mais direto que a estratégia LRU para guiar as decisões de substituição de página no buffer
  - Mantém as páginas que estão nos níveis mais altos da árvore (i.e., próximas à raiz)
  - Utiliza a política LRU para as demais páginas (i.e., páginas mais utilizadas)

Árvore B*

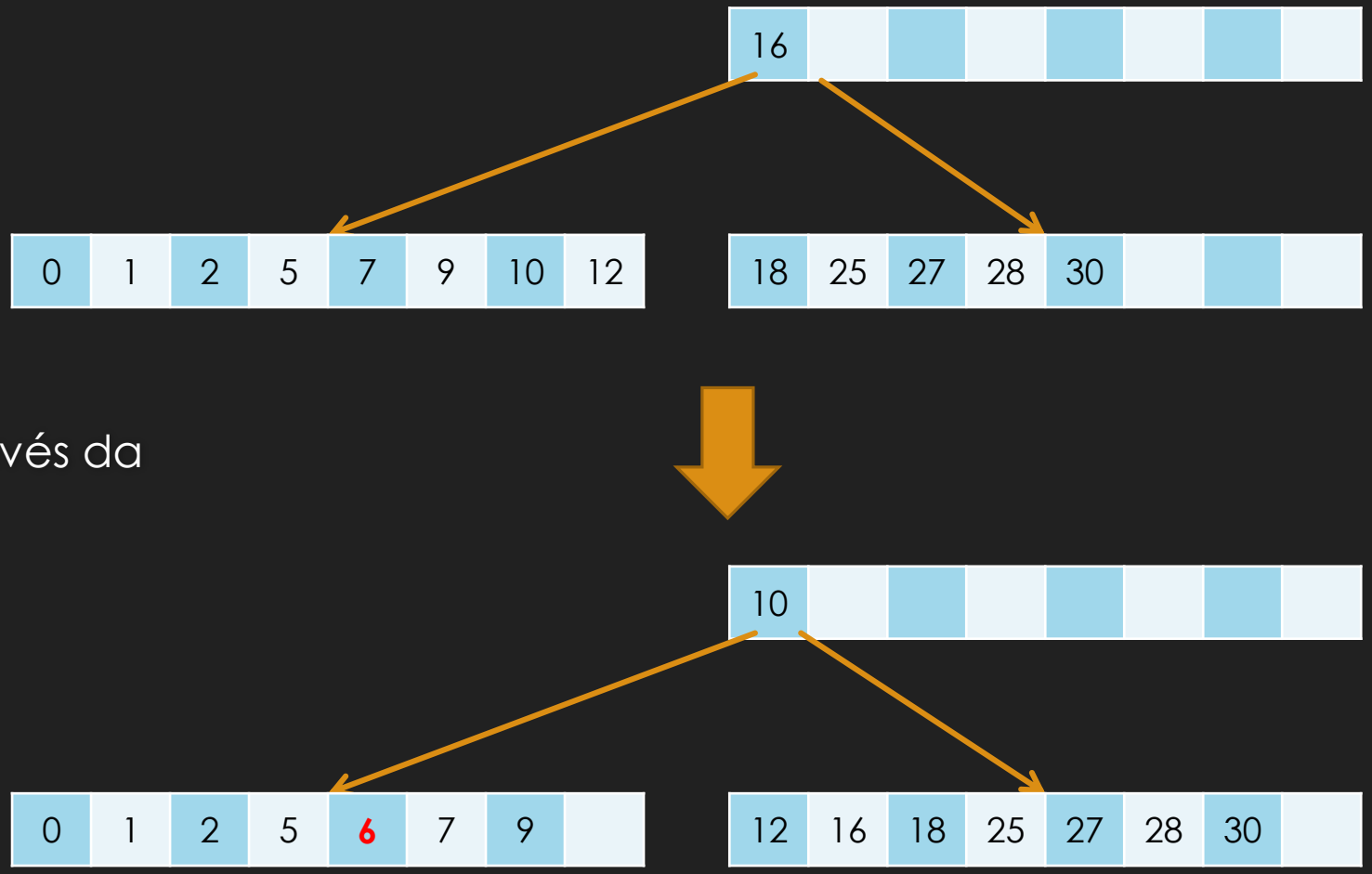
Árvore B*

- Proposta por Knuth em 1973 como uma variação de árvore B
- Característica
  - Cada nó contém, no mínimo, **2/3** do número máximo de chaves
  - Na árvore B de ordem **m**, o número mínimo de chaves é $\lceil m/2 \rceil - 1$
  - Geração da árvore é feita por um processo de redistribuição e subdivisão

Árvore B*

- Postergar divisão de páginas (*split*) através da redistribuição na inserção
  - Estende a noção de redistribuição durante a inserção para incluir novas regras para o particionamento de nós
  - A subdivisão é adiada até que duas páginas irmãs estejam cheias

Árvore B*



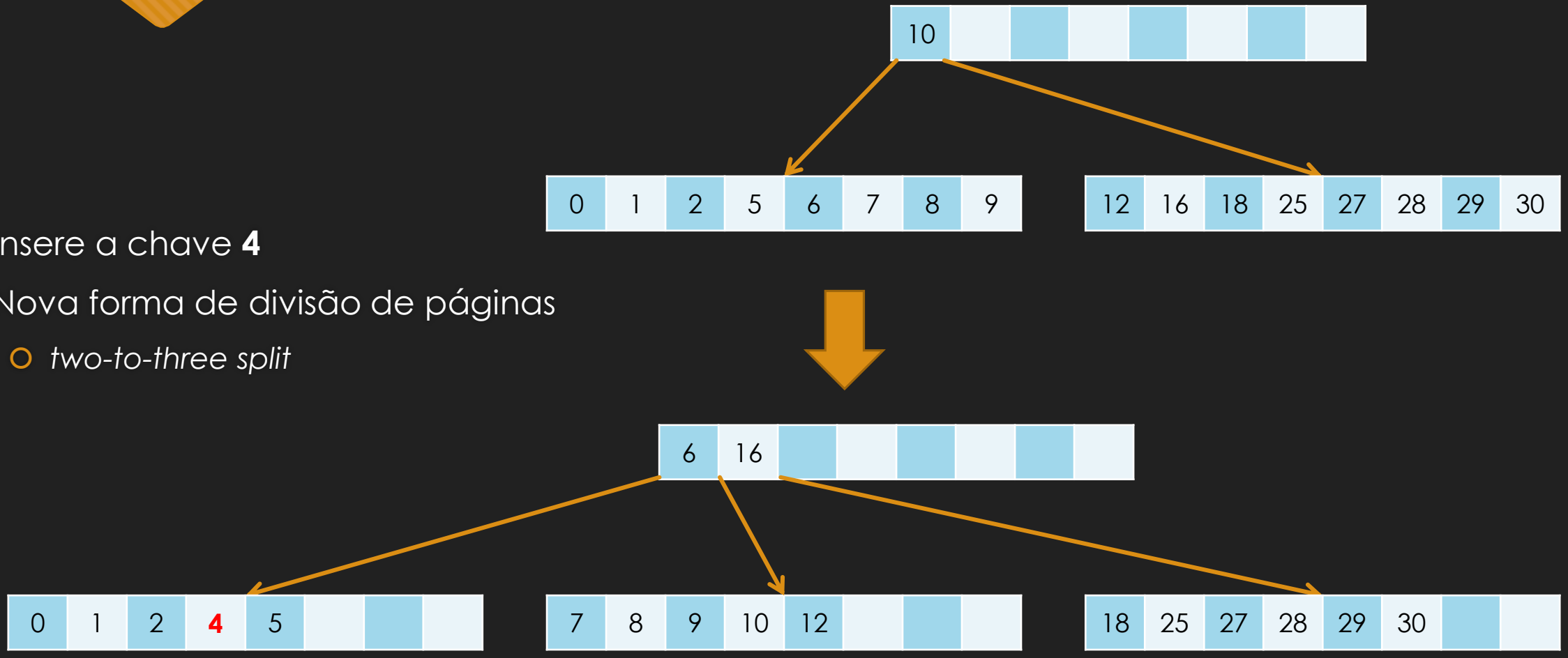
- Insere a chave **6**
- Postergar divisão de páginas através da redistribuição

Árvore B*

- Nova forma de divisão de páginas, de modo a garantir a ocupação mínima
  - Divisão do conteúdo de duas páginas irmãs em três páginas (*two-to-three split*)
- Funcionamento
  - Árvore B: split 1-to-2
  - Árvore B*: split 2-to-3, pelo menos um nó irmão está cheio

Árvore B*

- Insere a chave **4**
- Nova forma de divisão de páginas
  - *two-to-three split*



Árvore B*

- Mudança na taxa de ocupação afeta as rotinas de remoção e redistribuição
- Particionamento da raiz é um problema
  - Raiz não possui nó irmão para fazer divisão 2 para 3 páginas
  - Se dividir, os filhos da raiz não terão a taxa de ocupação mínima de 2/3

Árvore B*

- Soluções possíveis
- Permitir que a raiz seja maior (tamanho de página diferente)
  - Assim, quando dividir, terá 2 filhos com 2/3 de ocupação
- Fazer uma divisão convencional na raiz
  - Dividir a raiz usando a divisão 1-to-2 split
  - Permitir os filhos da raiz com uma taxa de ocupação menor que 2/3, como uma exceção

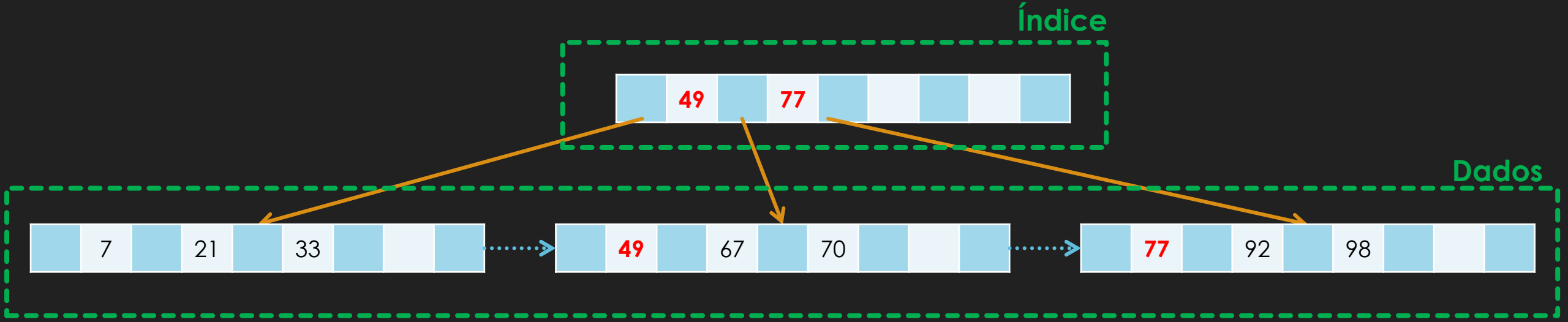
Árvore B+

Árvore B+

- Semelhante à árvore B, exceto por duas características muito importantes:
  - Armazena dados somente nas folhas – os nós internos servem apenas de ponteiros
  - As folhas são encadeadas. Cada página folha aponta para sua próxima, para permitir acesso sequencial

Árvore B+

- Isso permite o armazenamento dos dados em um arquivo, e do índice em outro arquivo separado
  - Diferente do que acontece nas árvores B

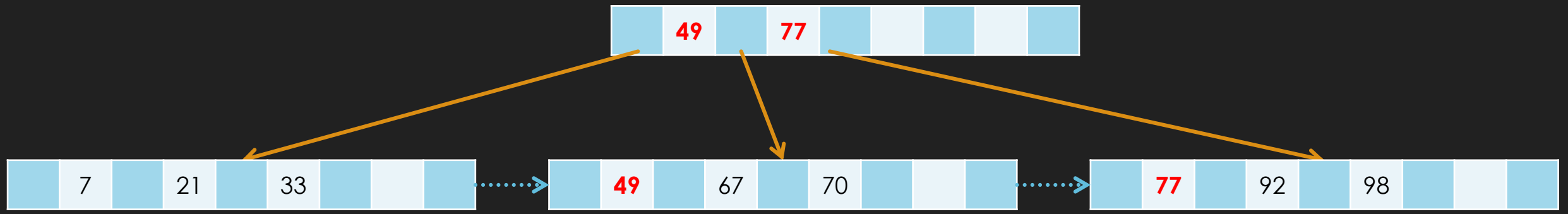


Árvore B+

- São muito importantes por sua eficiência, e muito utilizadas na prática
  - Os sistemas de arquivo NTFS, ReiserFS, NSS, XFS, e JFS utilizam este tipo de árvore para indexação
  - Sistemas de Gerência de Banco de Dados como IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASE, PostgreSQL, Firebird, MySQL e SQLite permitem o uso deste tipo de árvore para indexar tabelas
  - Outros sistemas de gerência de dados como o CouchDB, Tokyo Cabinet e Tokyo Tyrant permitem o uso deste tipo de árvore para acesso a dados

Árvore B+: busca

- Só se pode ter certeza de que o registro foi encontrado quando se chega em uma folha
  - Não existem dados nos nós intermediários, apenas nas folhas
  - Os dados dá chave **77** estão na folha, não no nó raiz
- As comparações agora não são apenas >, mas ≥
  - Índices **repetem valores** de chave que aparecem nas folhas

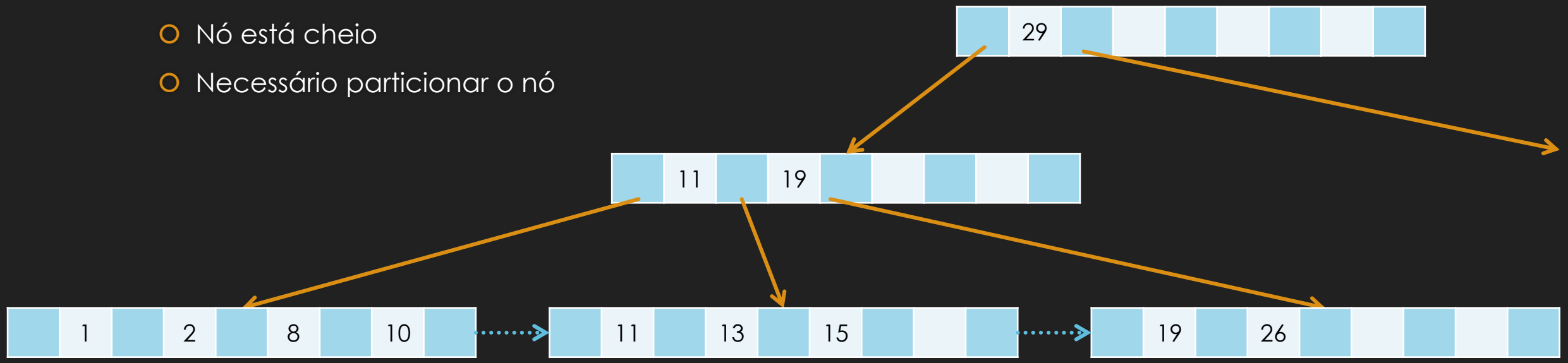


Árvore B+: inserção

- Quando for necessário particionar um nó durante uma inserção, usa-se o mesmo raciocínio da Árvore B
  - A diferença é que sobe somente a chave para o nó pai
  - O registro fica na folha, juntamente com a sua chave
  - **ATENÇÃO**: isso vale apenas se o nó que está sendo particionado for uma folha. Se não for folha, o procedimento é o mesmo utilizado na árvore B

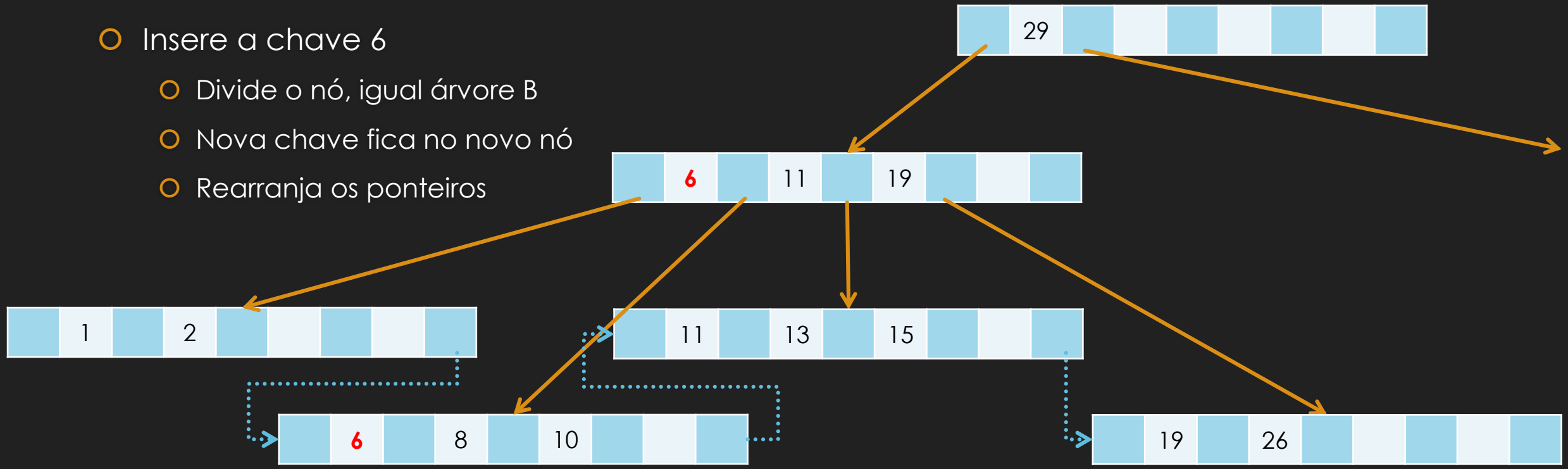
Árvore B+: inserção

- Insere a chave 6
  - Nó está cheio
  - Necessário particionar o nó



Árvore B+: inserção

- Insere a chave 6
  - Divide o nó, igual árvore B
  - Nova chave fica no novo nó
  - Rearranja os ponteiros



Árvore B+: remoção

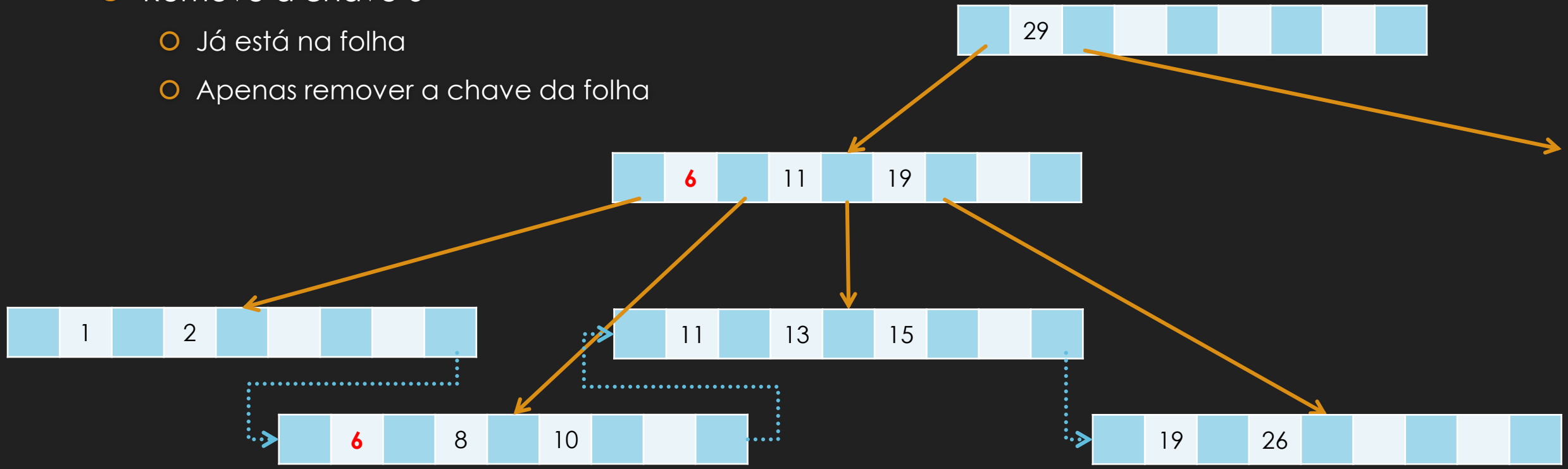
- A remoção ocorre apenas no nó folha
  - Similar a árvore B
  - Pode ser necessário fazer concatenação dos nós ou redistribuição das chaves
- As chaves excluídas continuam nos nós intermediários

Árvore B+: remoção

- Remoções que causem concatenação de folhas podem se propagar para os nós internos da árvore
  - Se a concatenação ocorrer na folha: a chave do nó pai não desce para o nó concatenado, pois ele não carrega dados com ele. Ele é simplesmente apagado
  - Se a concatenação ocorrer em nó interno: usa-se a mesma lógica utilizada na árvore B
- Remoções que causem redistribuição dos registros nas folhas provocam mudanças no conteúdo do índice, mas não na estrutura (não se propagam)

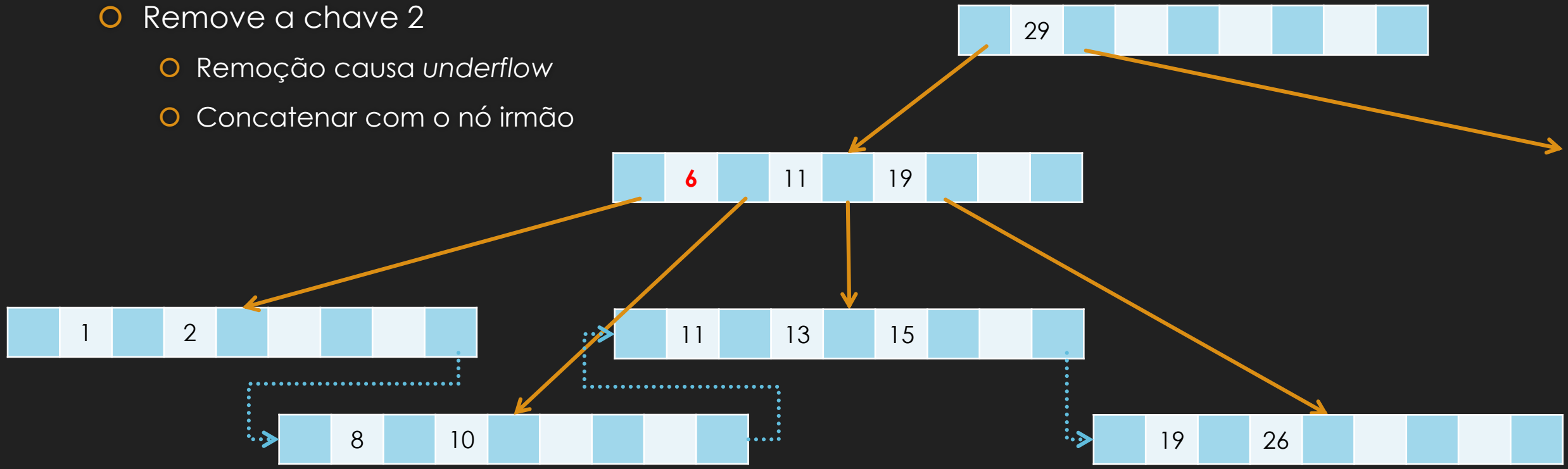
Árvore B+: remoção

- Remove a chave 6
  - Já está na folha
  - Apenas remover a chave da folha



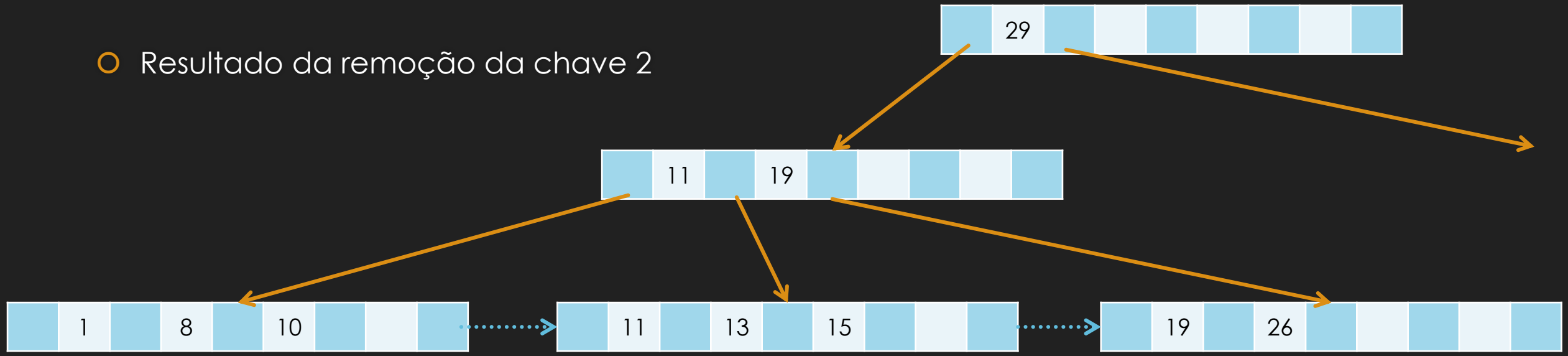
Árvore B+: remoção

- Resultado da remoção da chave 6
- Remove a chave 2
  - Remoção causa *underflow*
  - Concatenar com o nó irmão



Árvore B+: remoção

- Resultado da remoção da chave 2



Árvore B+ de prefixo simples (ou pré-fixada)

Acessando um arquivo

- Basicamente, podemos acessar um arquivo de duas maneiras:
  - Acesso indexado: arquivo é um conjunto de registros que são indexados por uma chave
  - Acesso sequencial: arquivo é acessado sequencialmente (i.e., registros fisicamente contínuos)

- Queremos que os arquivos suportem acesso indexado eficiente, e também acesso sequencial

Acessando um arquivo

- Arquivo indexado por um índice árvore-B
  - Acesso indexado pela chave
    - Desempenho excelente
    - Ordem logarítmica
  - Acesso sequencial aos registros ordenados pela chave
    - Desempenho péssimo
    - Ordem linear

Acessando um arquivo

- Arquivo com registros ordenados pela chave
  - Processamento sequencial (acessar todos registros)
    - Desempenho apropriado
    - Buferização
  - Processamento randômico
    - Desempenho inapropriado
    - Logarítmico (ordem 2)

Acessando um arquivo

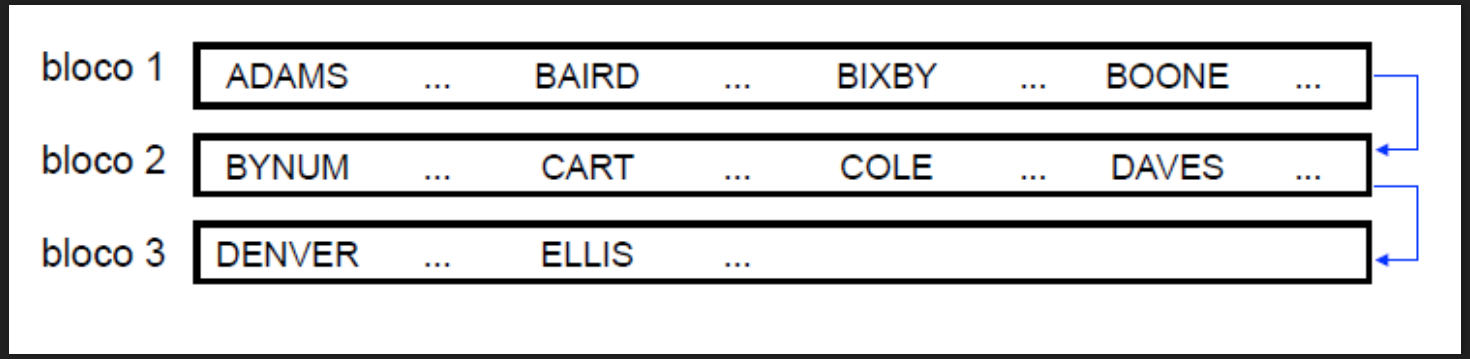
- Solução: usar um modelo híbrido
  - Organizar um arquivo de modo que seja eficiente tanto para processamento sequencial quanto aleatório
- Estrutura híbrida
  - Chaves: organizadas como árvore B (i.e., *index set*)
  - Nós folhas: consistem em blocos de dados (*sequence set*)

Árvore B+ de prefixo simples

- Variação da árvore B+ que utiliza um prefixo comum para armazenar e pesquisar chaves de forma mais eficiente
  - Armazena na árvore as cadeias separadoras mínimas entre cada par de blocos
  - Se as chaves forem ***strings***, um prefixo comum pode ser uma sequência de caracteres iniciais compartilhados entre as chaves
  - Usar separadores mínimos faz com que os nós possam ser maiores
  - Necessidade de maior controle do tamanho do nó e de onde começa e termina cada cadeia separadora

Árvore B+ de prefixo simples

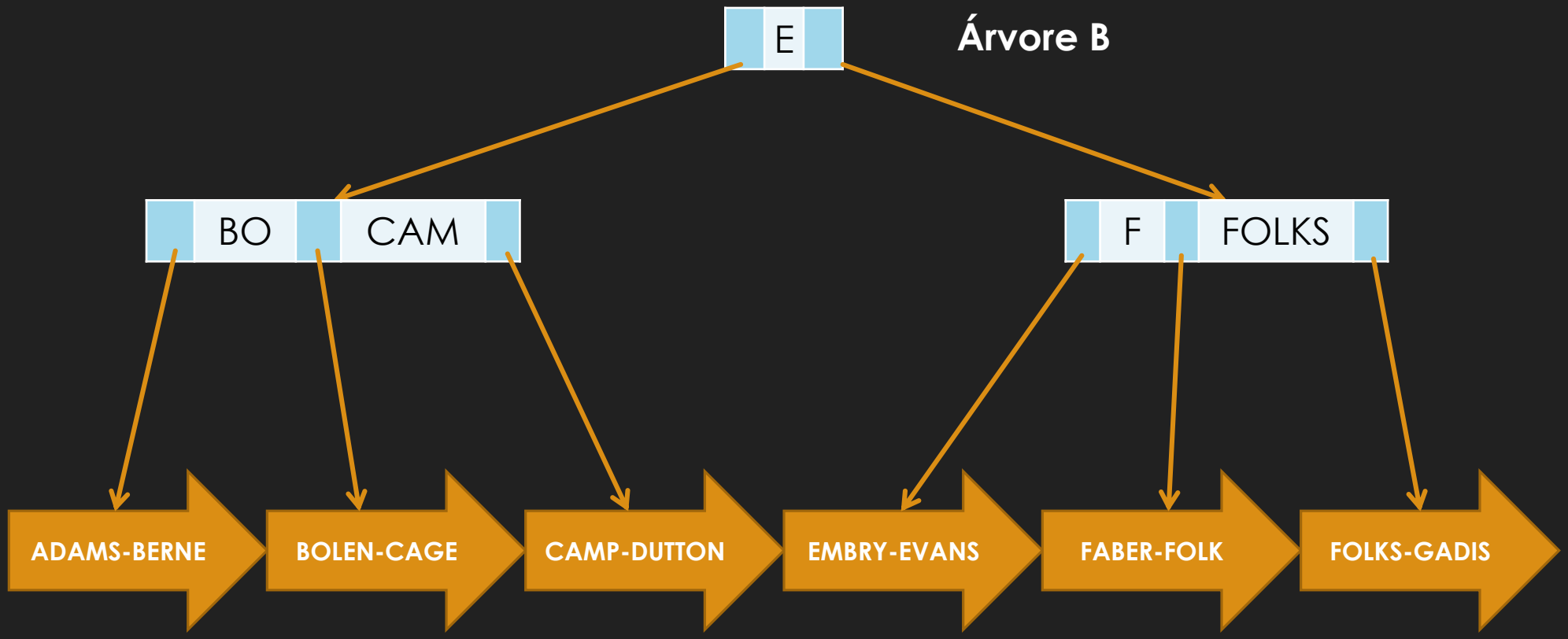
- *Sequence Set*
  - **A**rquivo de dados é organizado em blocos de tamanho fixo, de registros sequenciais, ordenados pelas chaves, e encadeados
  - Privilegia o acesso sequencial do arquivo



Árvore B+ de prefixo simples

- Arquivo de índices (*index set*) é organizado como uma Árvore B
  - As folhas são os blocos de registros sequenciais
  - Privilegia a busca aleatória no arquivo
  - Páginas não folhas contêm chaves ou partes de chaves separadoras para os filhos

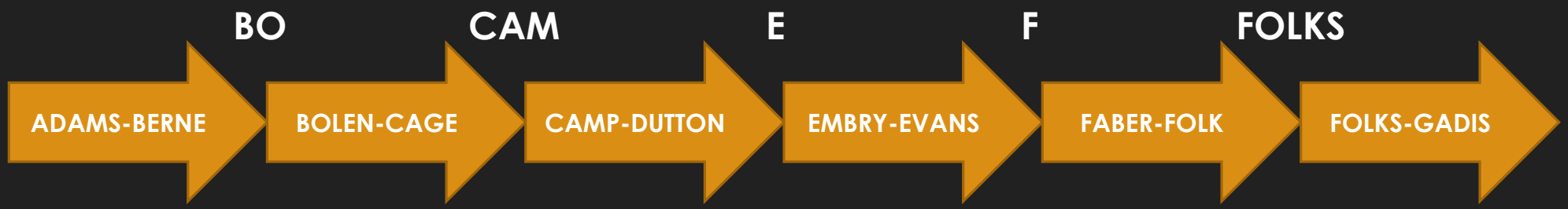
Árvore B+ de prefixo simples



Blocos de registros de dados, ordenados pelas chaves

Árvore B+ de prefixo simples

- Uso de separadores de blocos no lugar das chaves de busca
  - Os separadores são mantidos no índice, ao invés das chaves de busca
  - Escolhemos o menor separador possível
  - Possuem tamanho variável
  - Estruturas semelhantes podem ser consideradas como alternativas
    - Para chaves muito grandes e / ou repetitivas



Manutenção

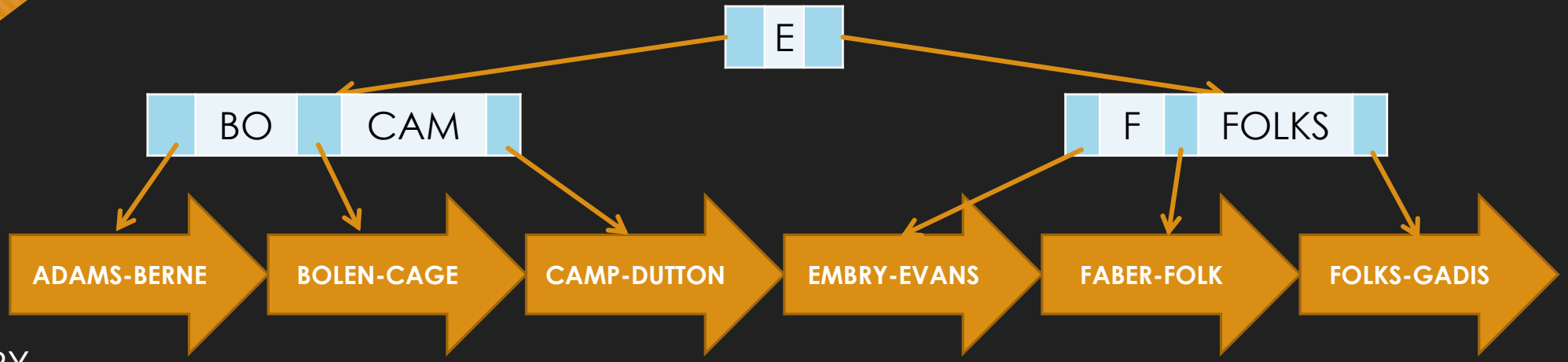
- É importante notar que as operações são realizadas dentro do *Sequence Set*, pois é lá que os registros estão
- Mudanças no índice são consequências das operações fundamentais aplicadas ao *Sequence Set*
  - Adicionamos um novo separador no índice apenas se um novo bloco é formado no *Sequence Set* como consequência de uma operação de divisão
  - Um separador é eliminado do índice apenas se um bloco é removido do *Sequence Set*, como consequência de uma concatenação

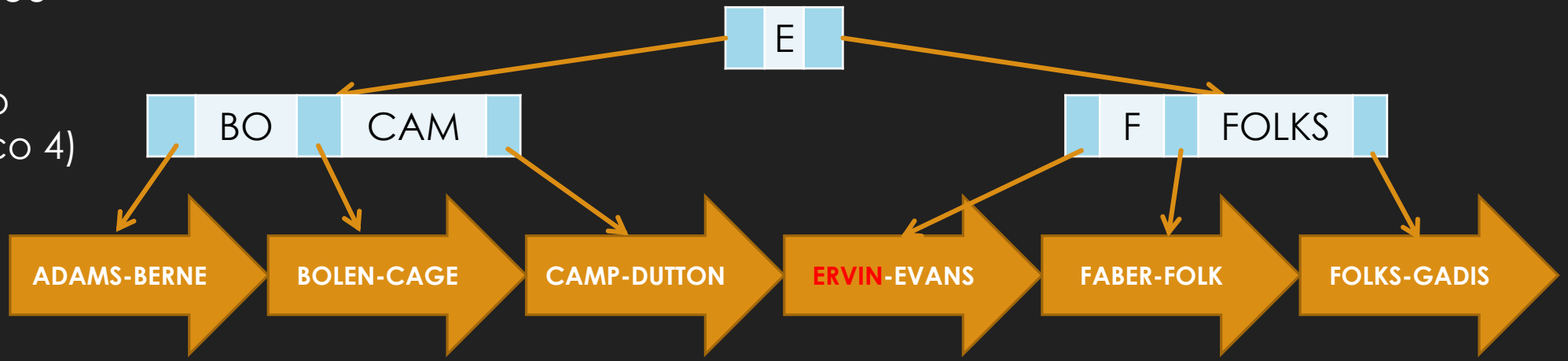
Manutenção

- A ocorrência de *overflow* e *underflow* dos nós do índice não acompanha a ocorrência de *overflow*/*underflow* no *Sequence Set*
- E uma divisão/concatenação de blocos no *Sequence Set* não resulta necessariamente em uma divisão/concatenação de nós do índice

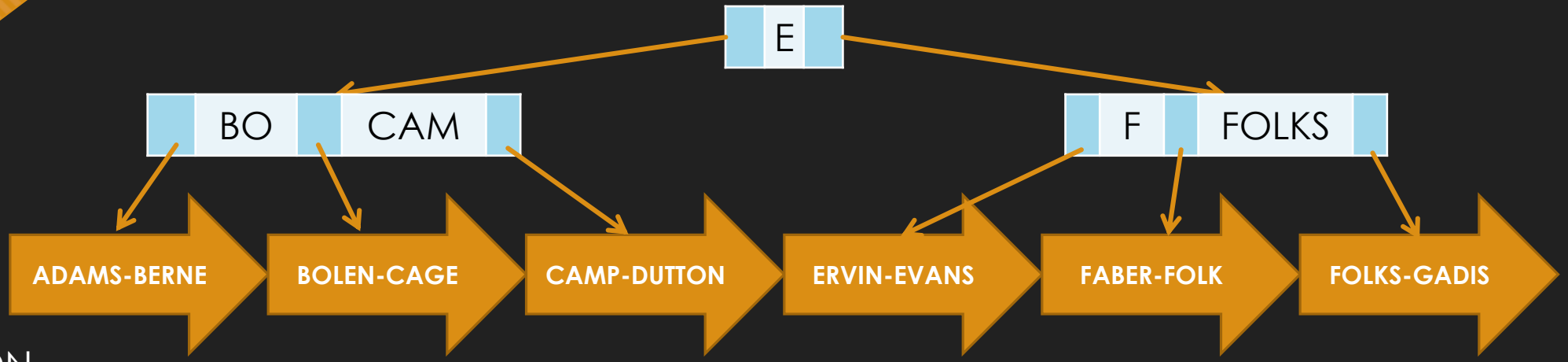
Exemplo: remoção

- Remoção de EMBRY
- Sem redistribuição ou concatenação
- Efeito é limitado ao *sequence set* (bloco 4)

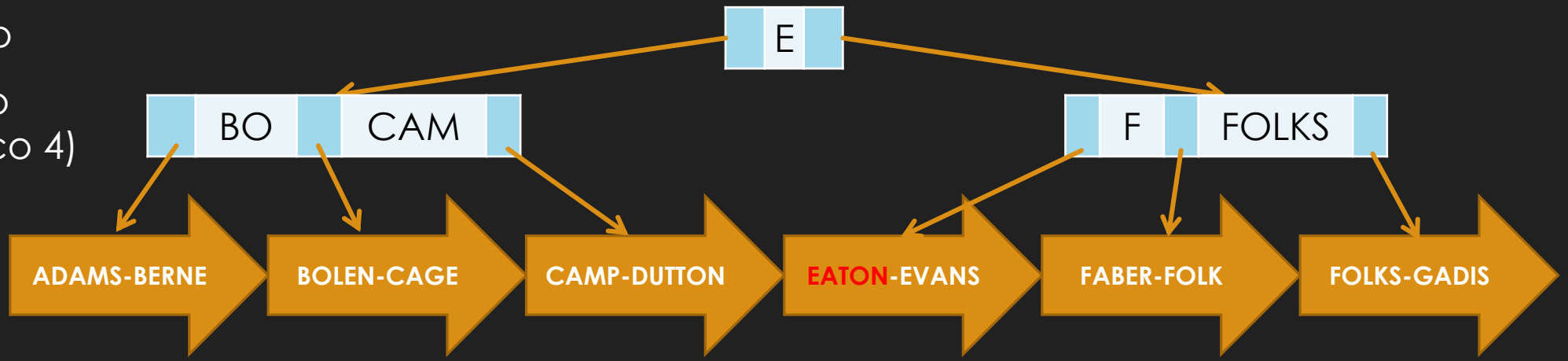




Exemplo: inserção



- Remoção de EATON
- Considera espaço disponível no bloco
- Efeito é limitado ao *sequence set* (bloco 4)



Material complementar

- Estrutura de Dados em C | Aula 154 – Árvore B Virtual
  - https://youtu.be/3KbA1EgSH5I
- Estrutura de Dados em C | Aula 155 – Árvore B*
  - https://youtu.be/kMlszH2un80
- Estrutura de Dados em C | Aula 156 - Árvore B+
  - https://youtu.be/ZbzbId1UGJ0
- Estrutura de Dados em C | Aula 157 - Árvore B+ de prefixo simples
  - https://youtu.be/uN4lC4OUGPk