B-Trees

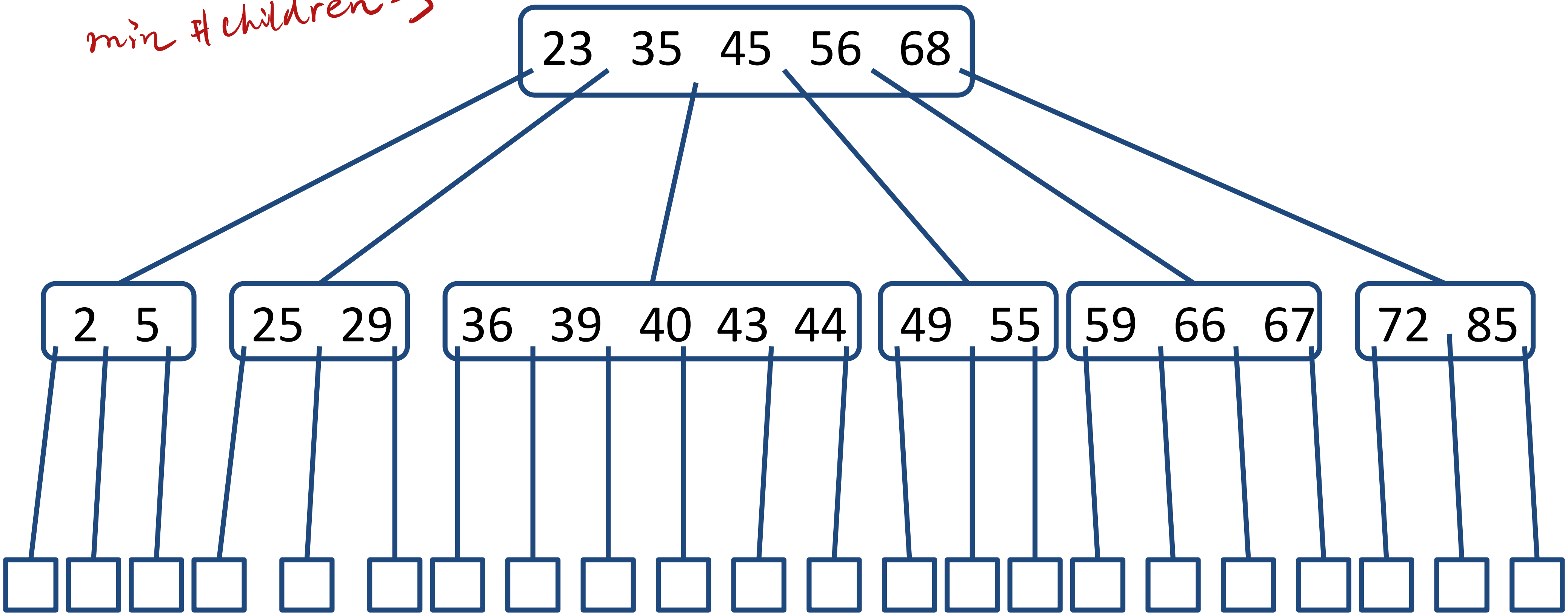
A B-tree of order $\boldsymbol{d}$ is a multiway search tree with the following properties:

- The root has at least 2 children and at most $\boldsymbol{d}$.
- All internal nodes other than the root have at least $\left\lceil \frac{d}{2} \right\rceil$ and at most $\boldsymbol{d}$ children
- All the leaves are at the same level

B-Trees

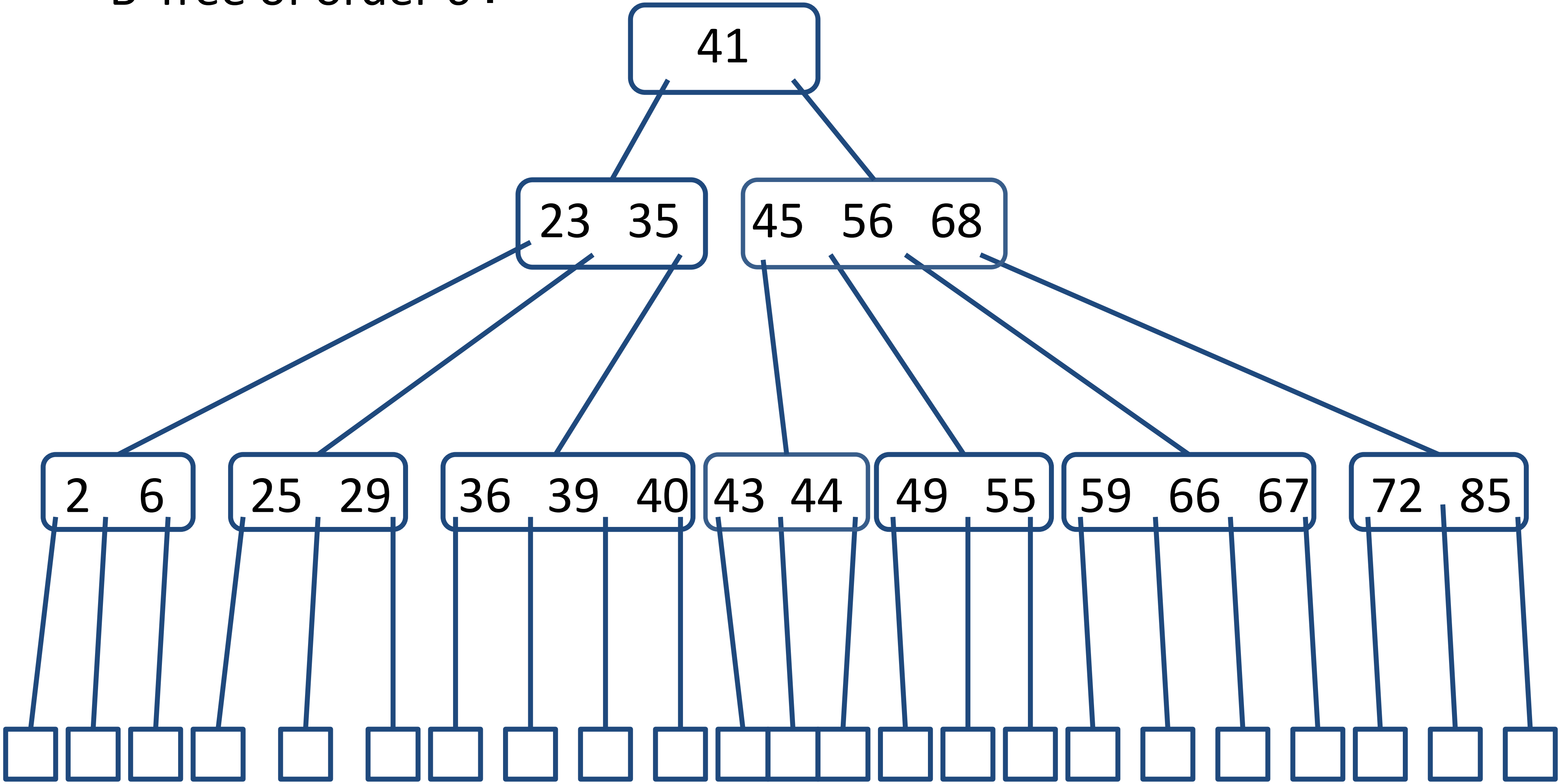
B-Tree of order 6?

min # children = 3



B-Trees

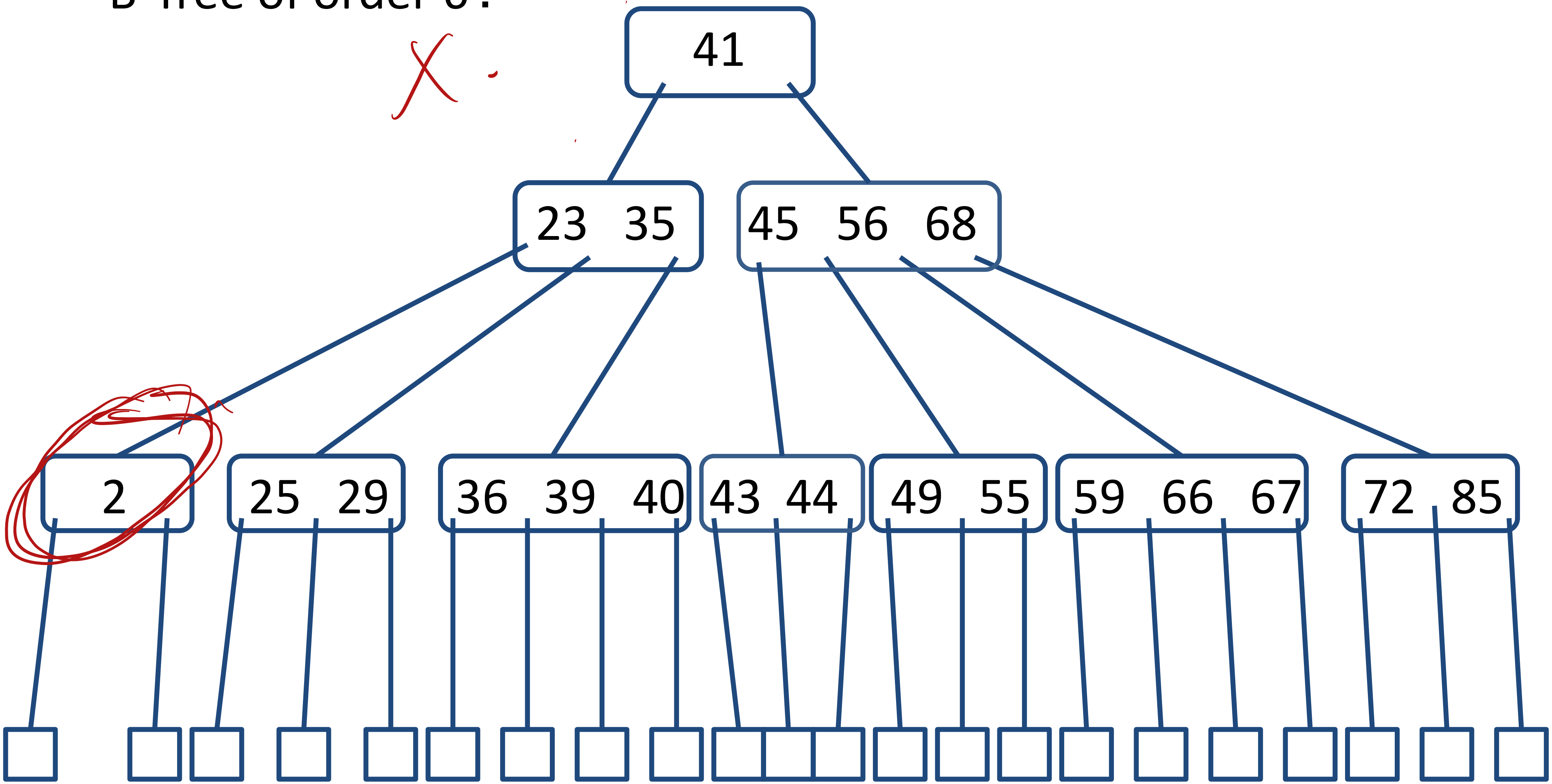
B-Tree of order 6?



B-Trees

B-Tree of order 6?

X



What is the Maximum Height of a B-Tree?

Height is $O(\log_d n)$

=> This may be asked in the final.

$$1+\frac{d}{2}+\left(\frac{d}{2}\right)^2+\cdots+\left(\frac{d}{2}\right)^h = n.$$

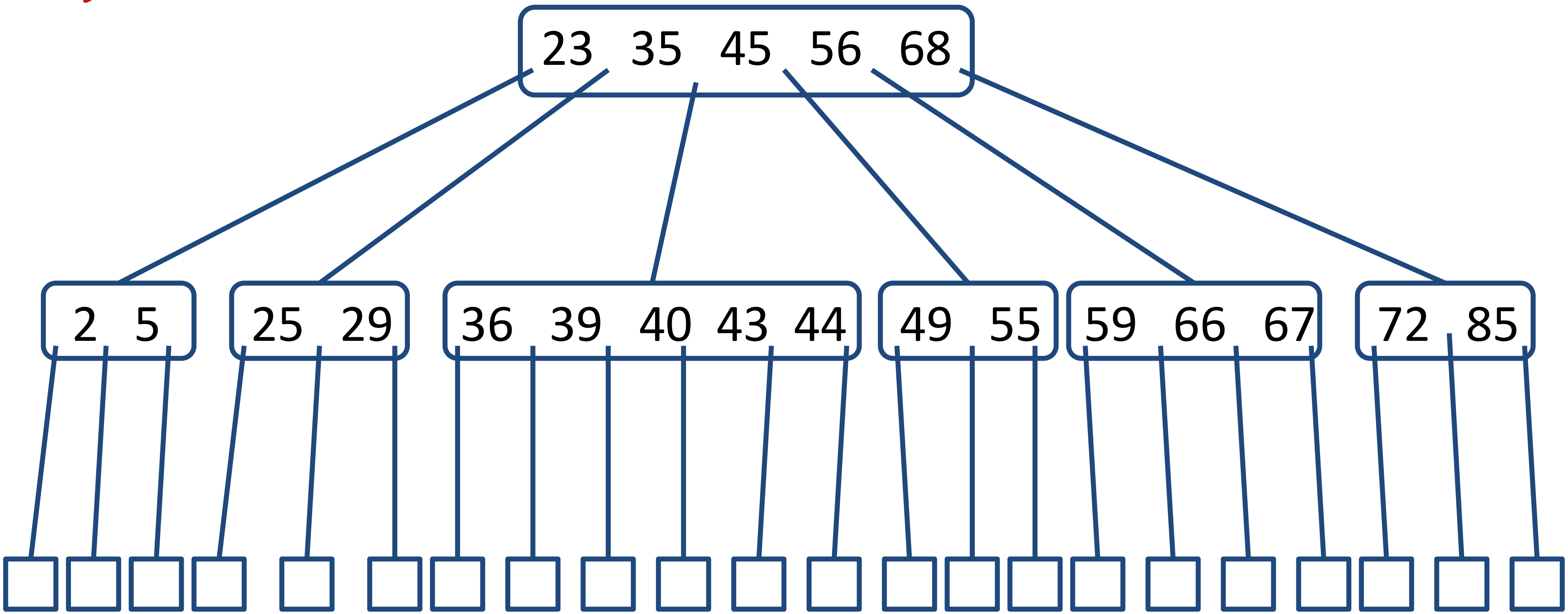
$$n = \frac{\left(\frac{d}{2}\right)^h - 1}{\frac{d}{2}-1}\cdot 1$$

$$h = \left\lfloor \log_{\frac{d}{2}} \left(\frac{d}{2}-1\right) n \right\rfloor + 1 \Rightarrow O(f(n)) = O(\log_d n)$$

B-Trees

B-Tree of order 6

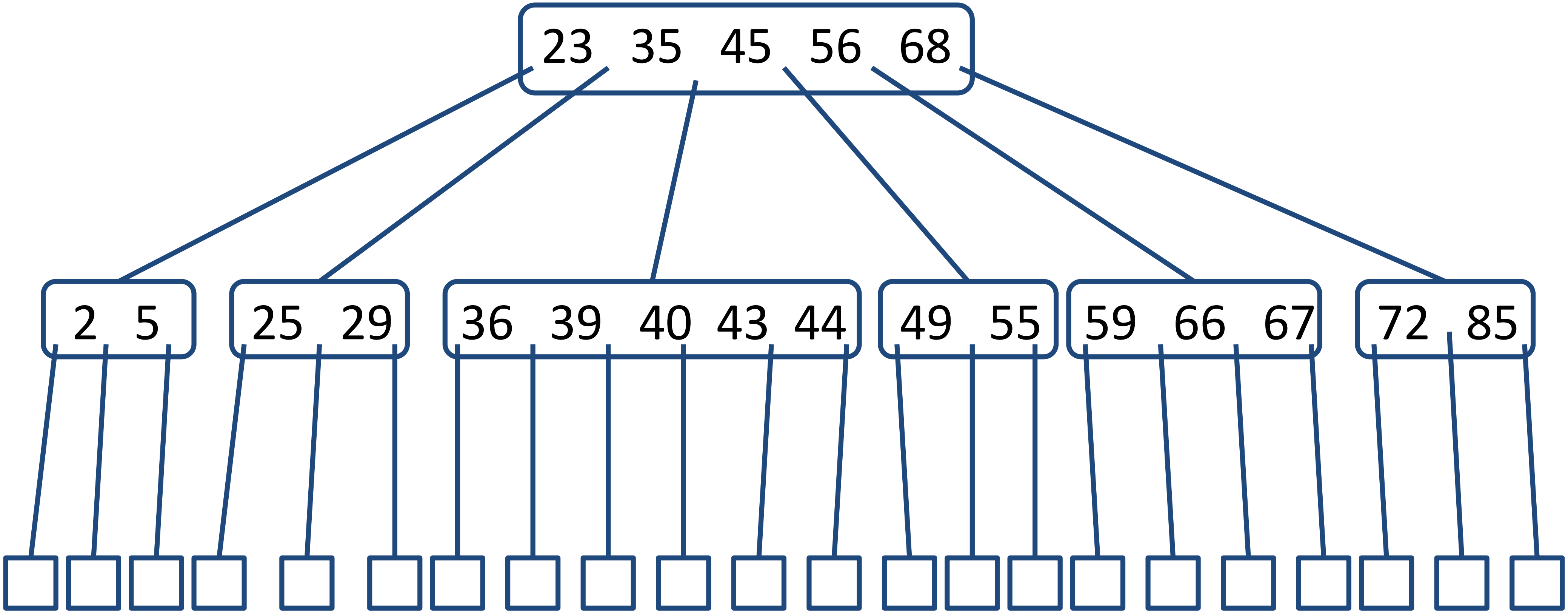
3 ≤ # children ≤ 6.



B-Trees

B-Tree of order 6

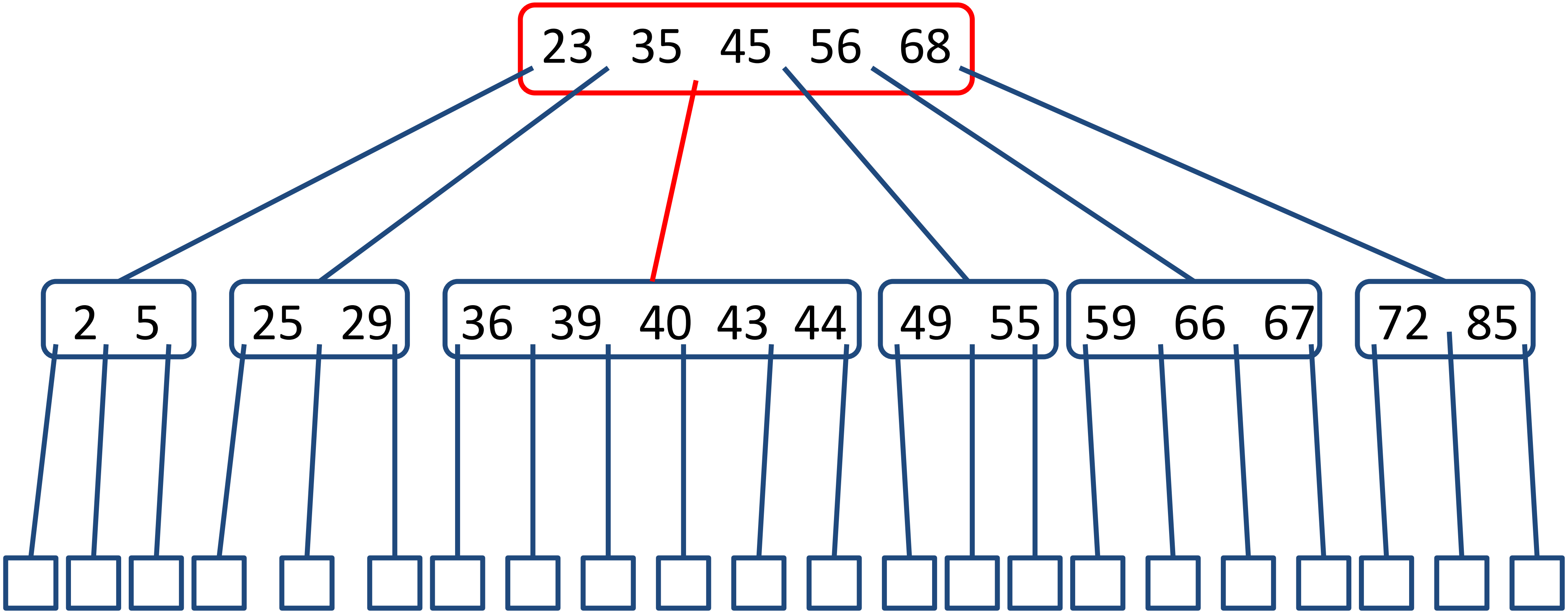
Put 38



B-Trees

B-Tree of order 6

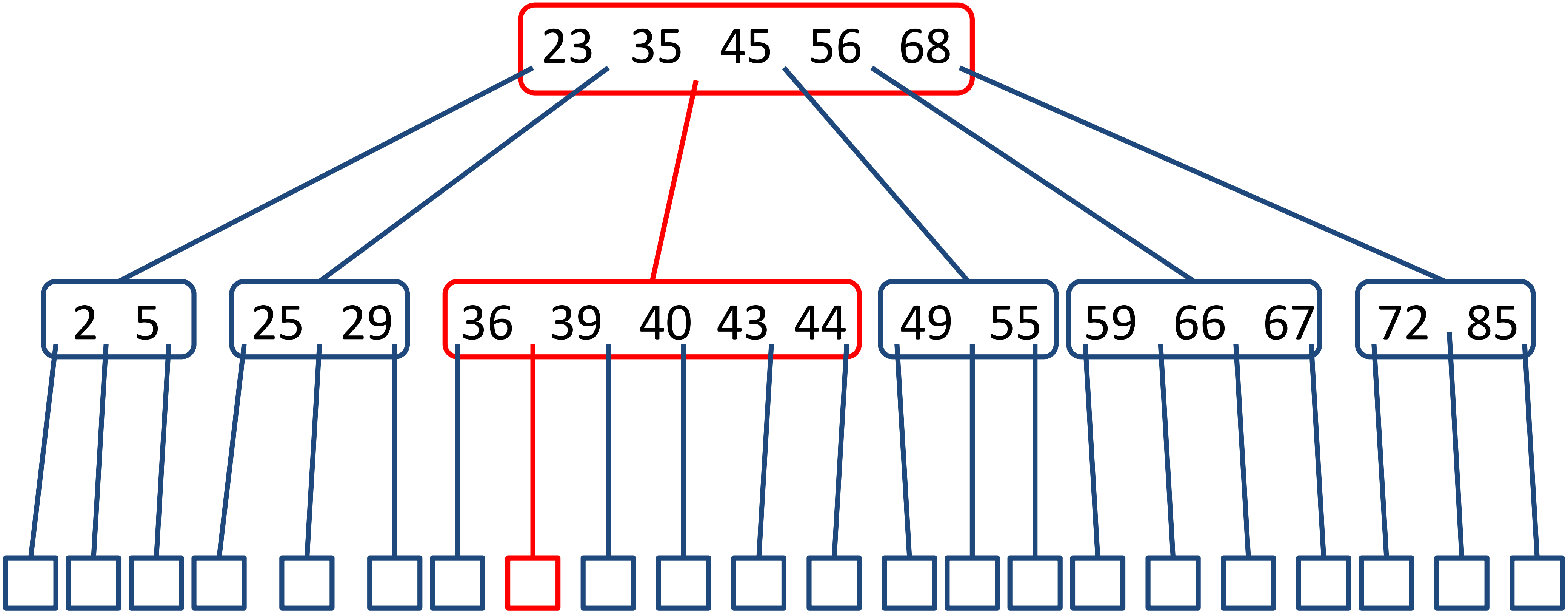
Put 38



B-Trees

B-Tree of order 6

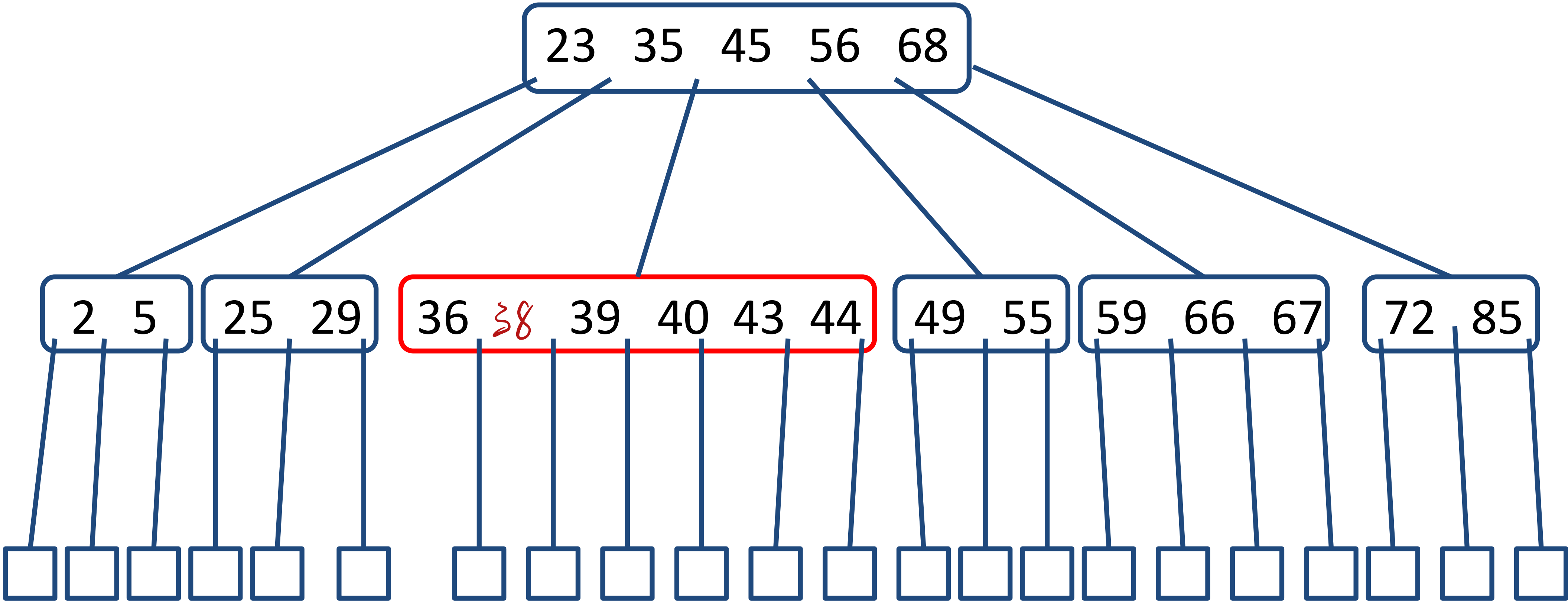
Put 38



B-Trees

B-Tree of order 6

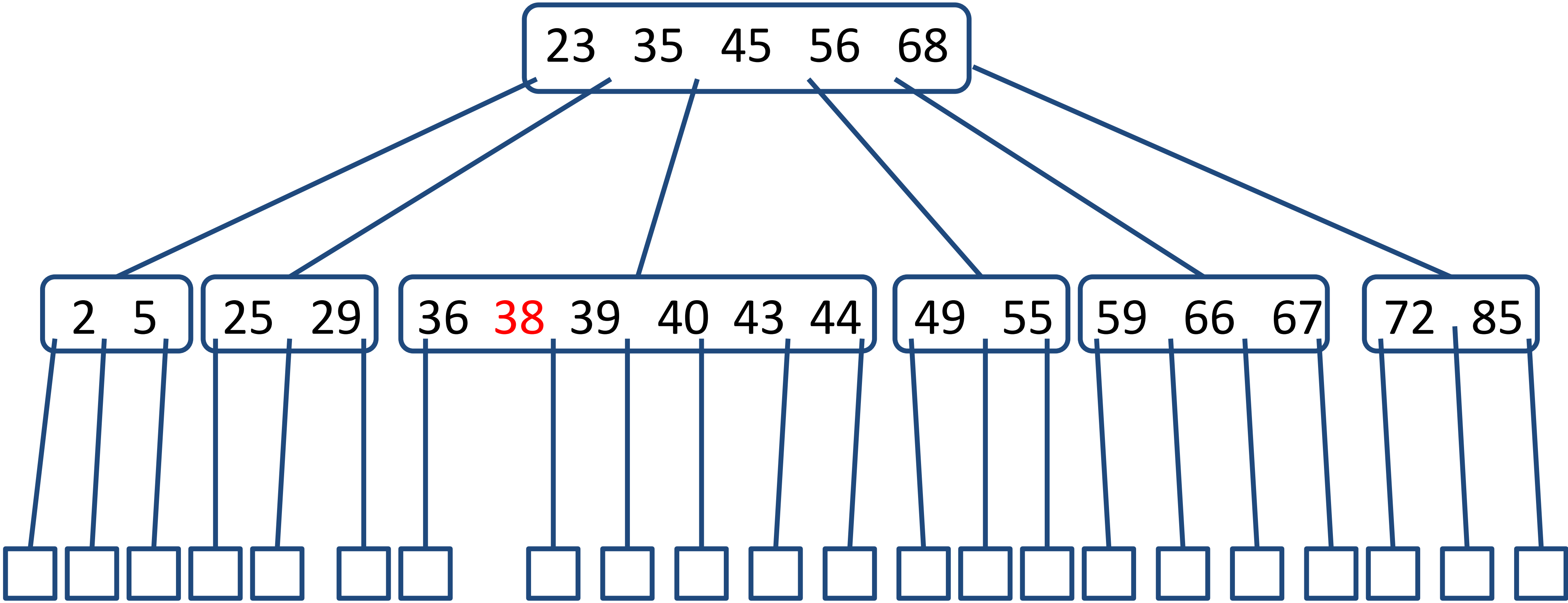
Put 38



B-Trees

B-Tree of order 6

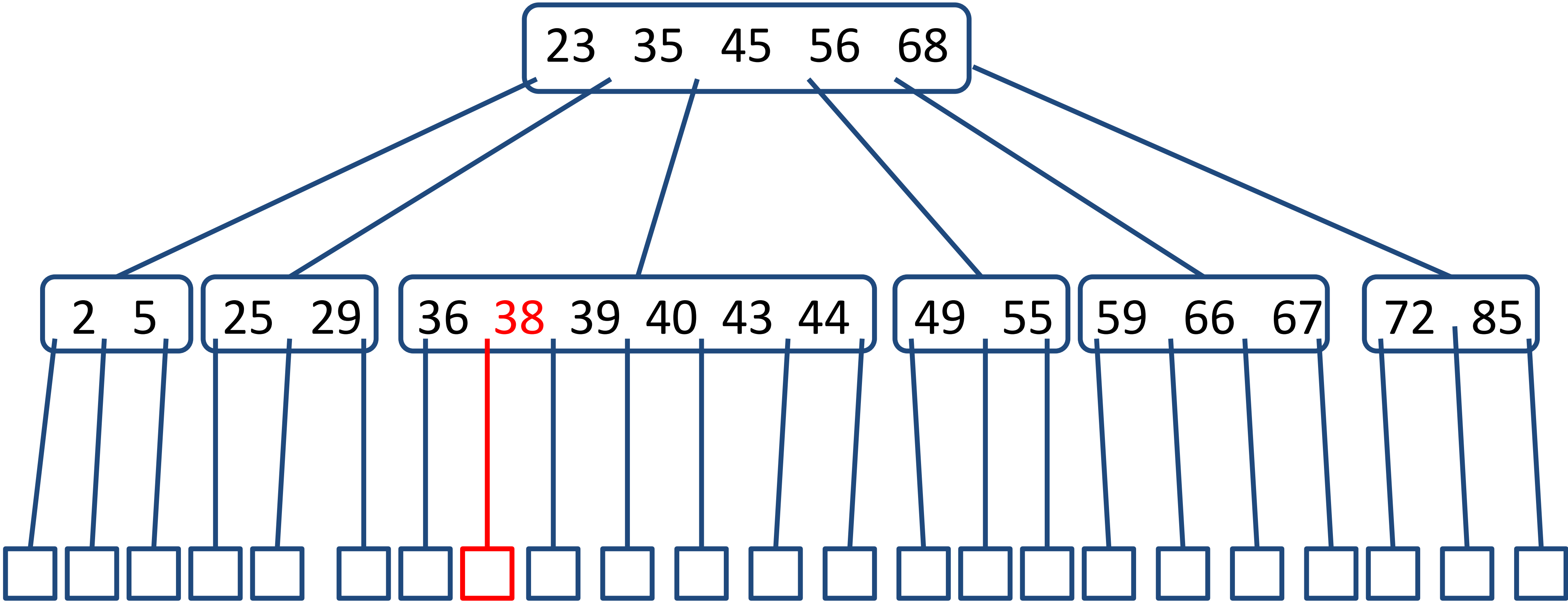
Put 38



B-Trees

B-Tree of order 6

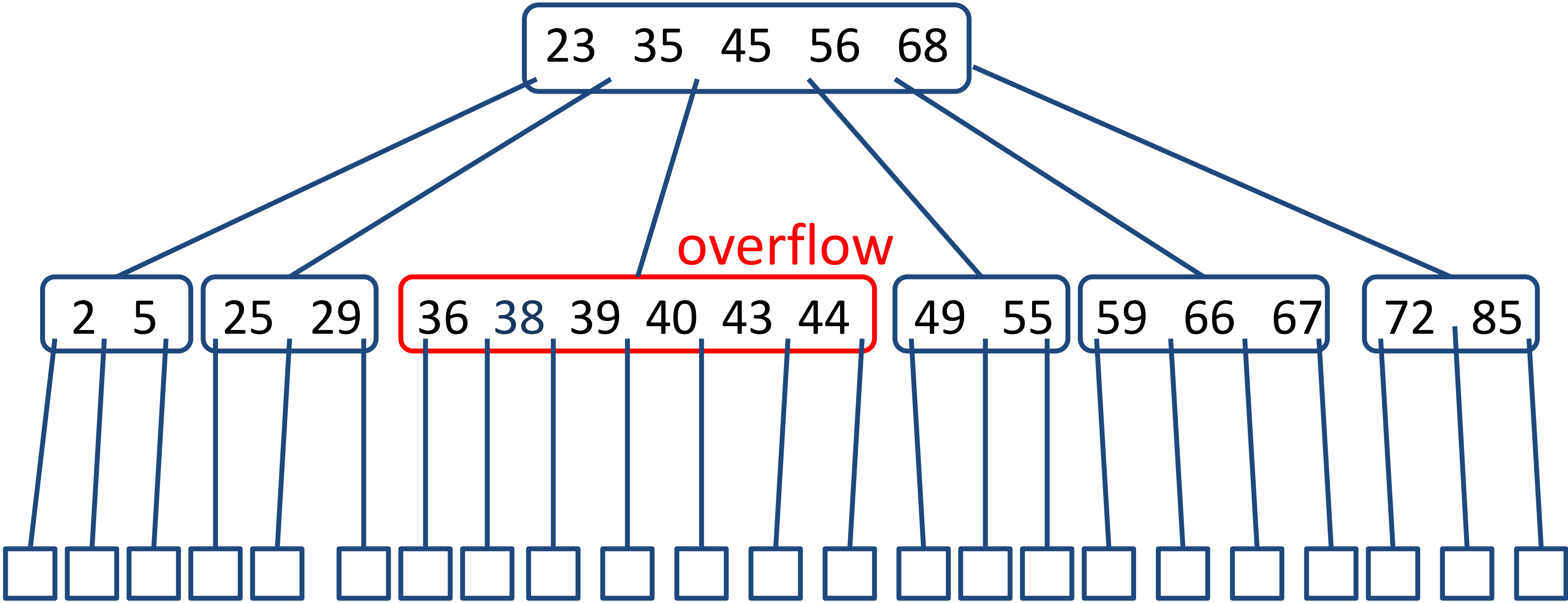
Put 38



B-Trees

B-Tree of order 6

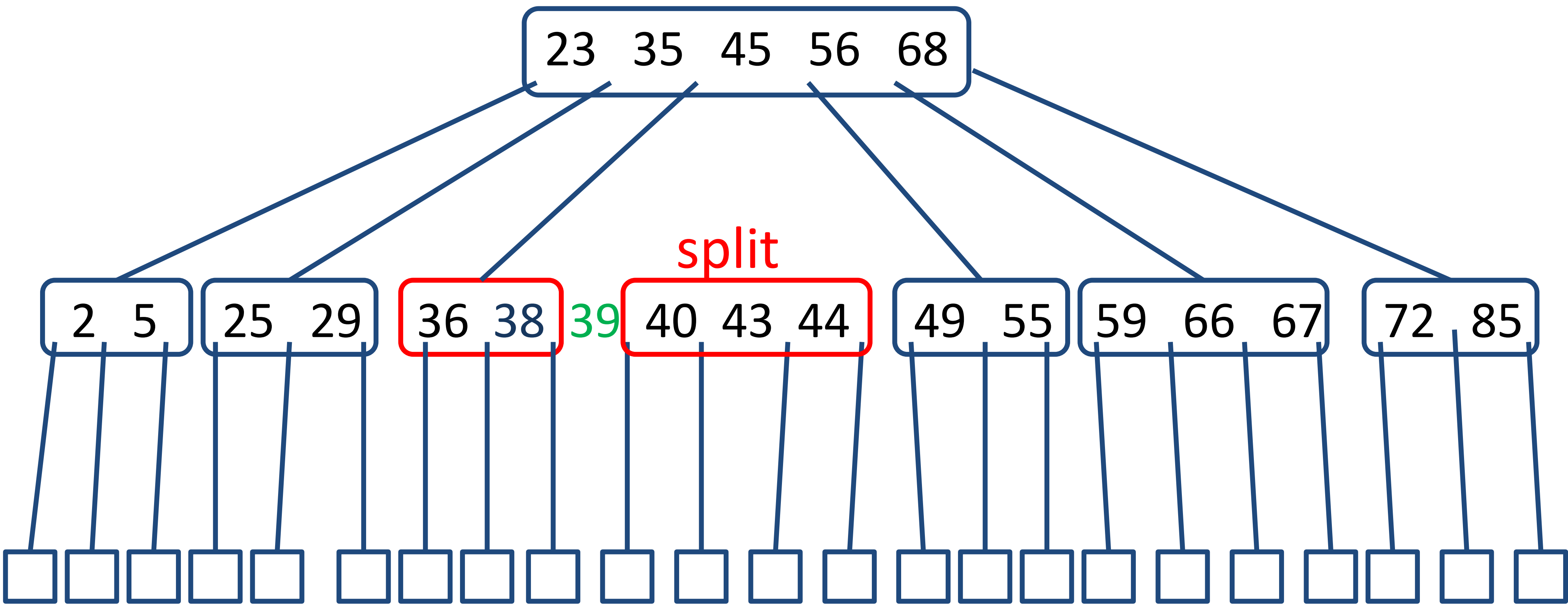
Put 38



B-Trees

B-Tree of order 6

Put 38

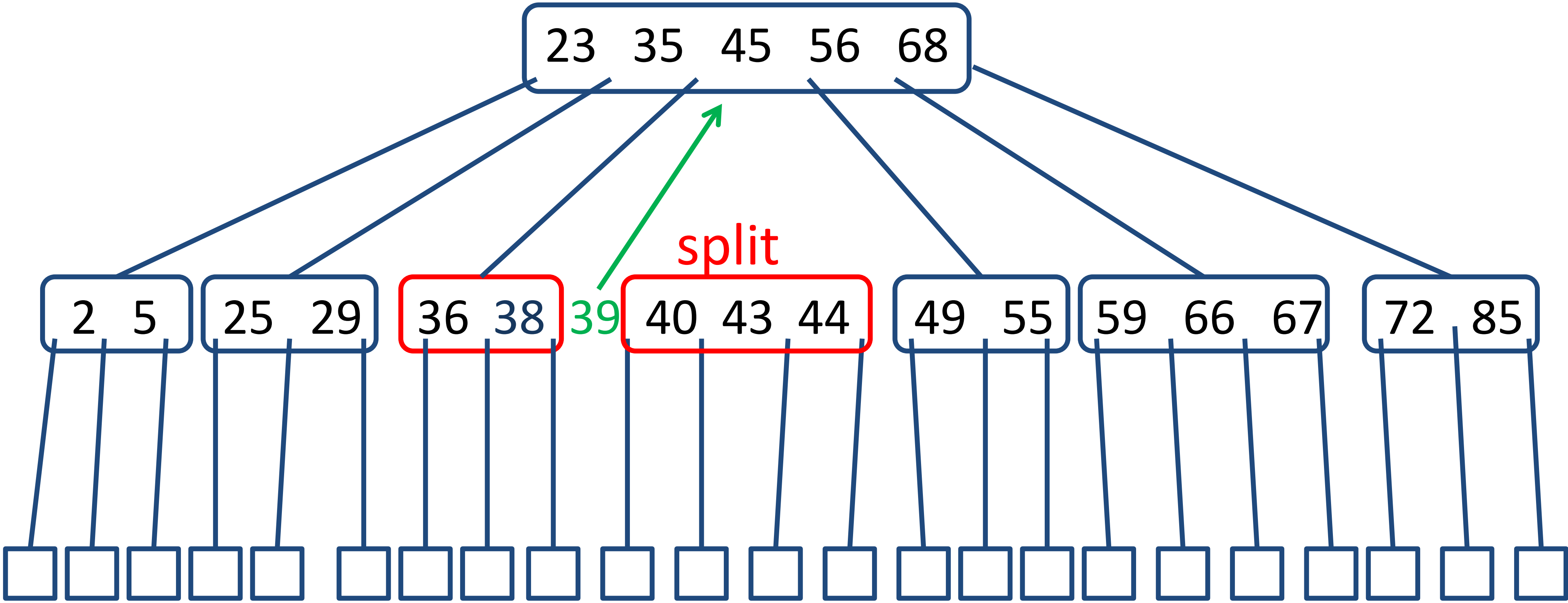


list[m/2 - 2]

B-Trees

B-Tree of order 6

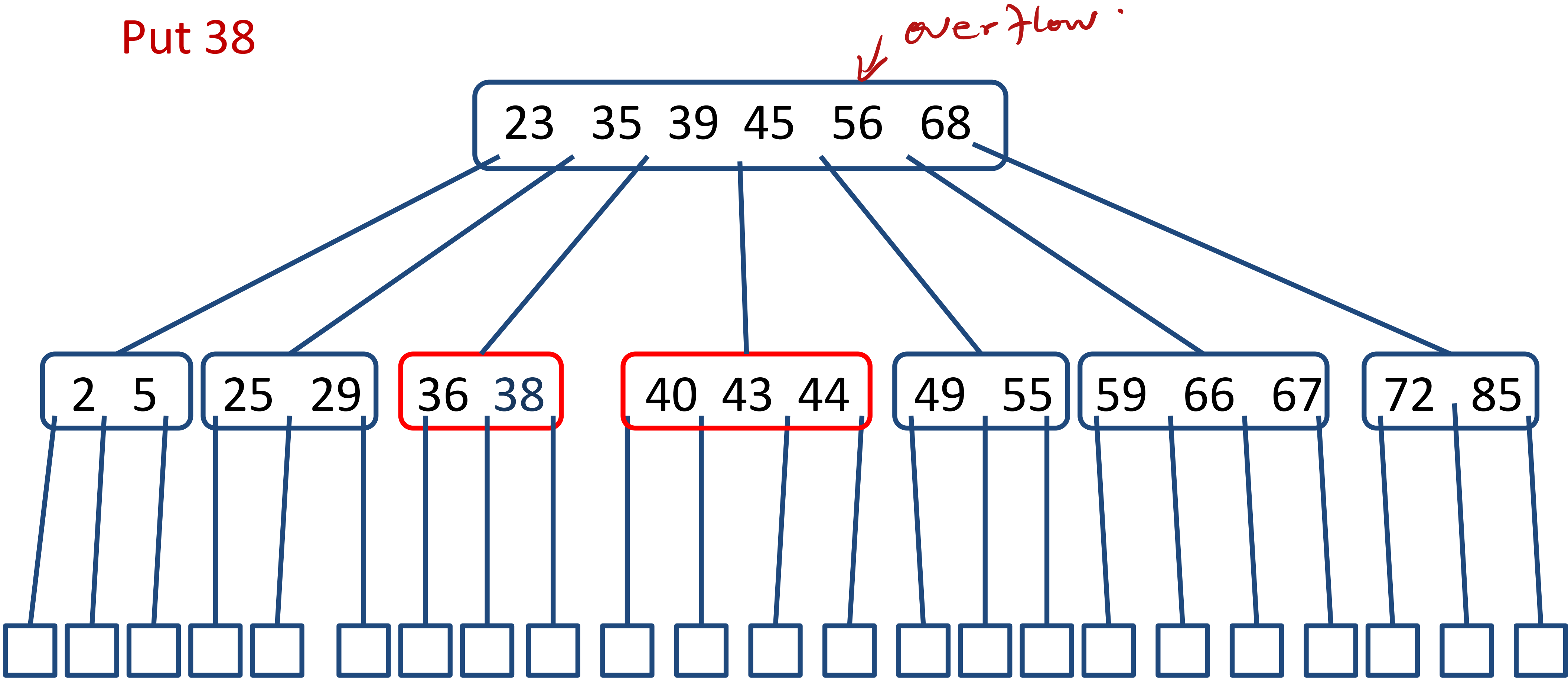
Put 38



B-Trees

B-Tree of order 6

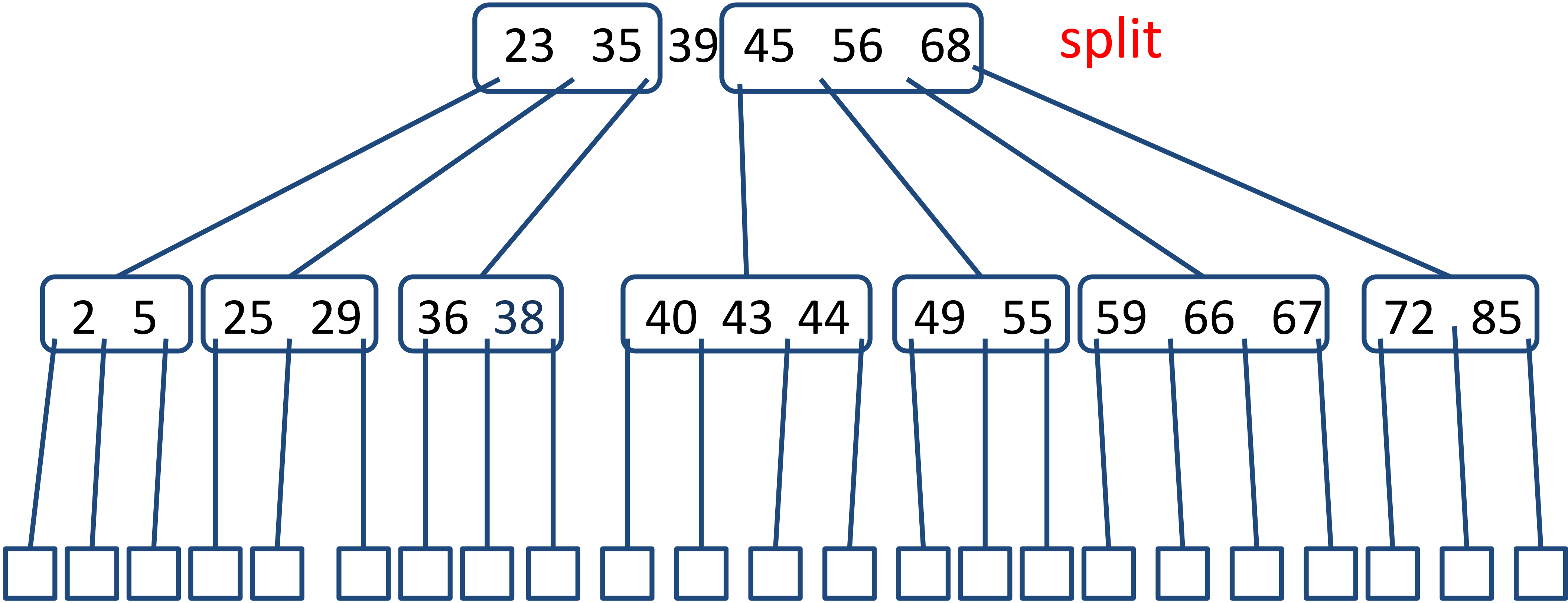
Put 38



B-Trees

B-Tree of order 6

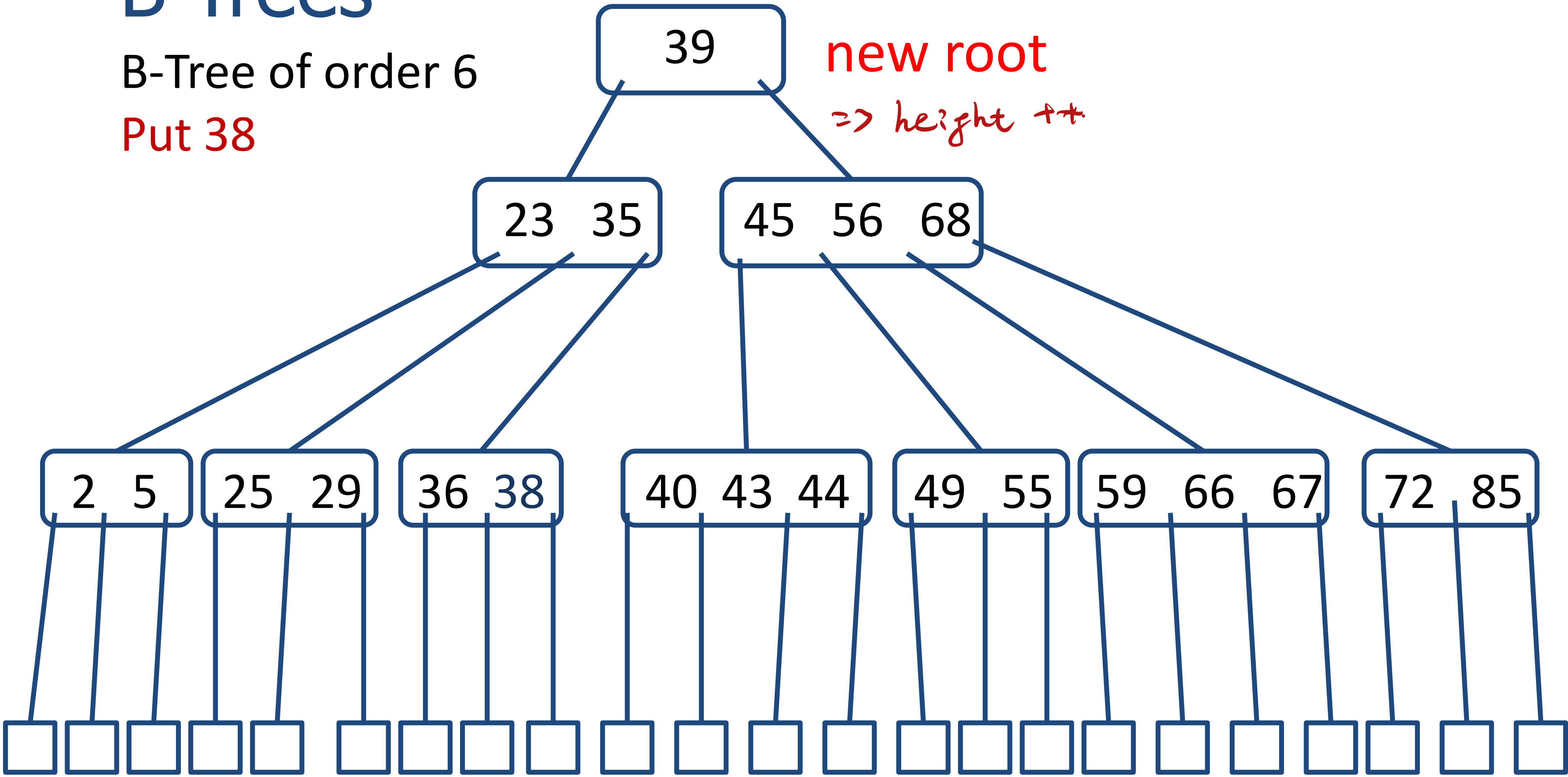
Put 38



B-Trees

B-Tree of order 6

Put 38



Algorithm ***put*** **(*r*,*k*,*o*)**

In: Root r of a B-tree, data item (k,o)

Out: {Insert data item (k,o) in the B-tree

Search for ***k*** to find the lowest insertion internal node ***v***

Add the new data item (***k***, ***o***) at node ***v***

try to fill out the full tree.

while node *v* ***overflows*** **do** {

if ***v*** is the root **then**

Create a new empty root and set as parent of ***v***

Split ***v*** around the middle key ***k'***, move ***k'*** to parent, and update parent's children

v ← parent of ***v***

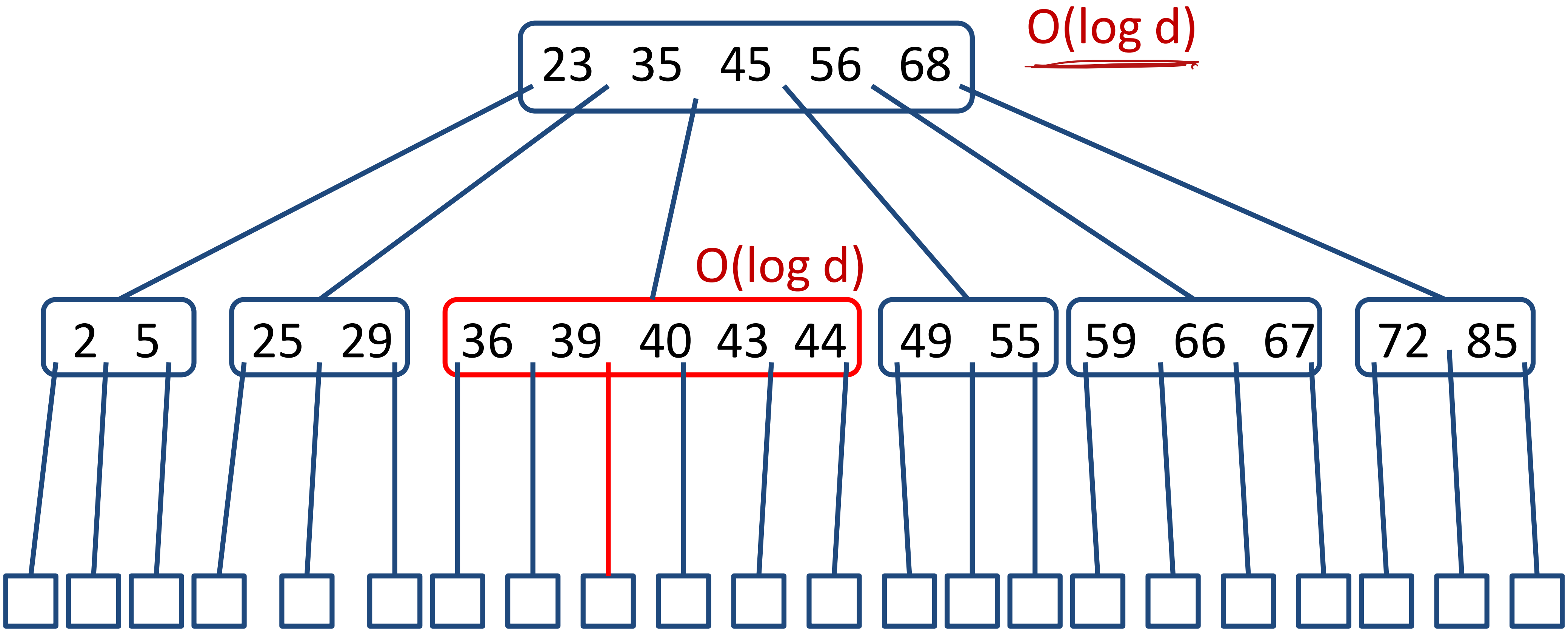
}

move the overflow to upper level.

B-Trees

B-Tree of order 6

B tree stores both data and keys in internal nodes.
B+ tree stores data at leave nodes, and kes in internal nodes.
B- tree is another name of B tree.



Algorithm ***put*** ***(r,k,o)***

In: Root *r* of a B-tree, data item (*k,o*)

Out: {Insert data item (*k,o*) in the B-tree

$O(\log d \times \underbrace{\log_d n}_{\text{recursive}})$

Search for ***k*** to find the lowest insertion internal node ***v***

Add the new data item (***k***, ***o***) at node ***v***

while node *v* ***overflow***s **do {**

if ***v*** is the root **then**

Create a new empty root and set as parent of ***v***

Split ***v*** around the middle key ***k'***, move ***k'*** to parent, and update parent's children

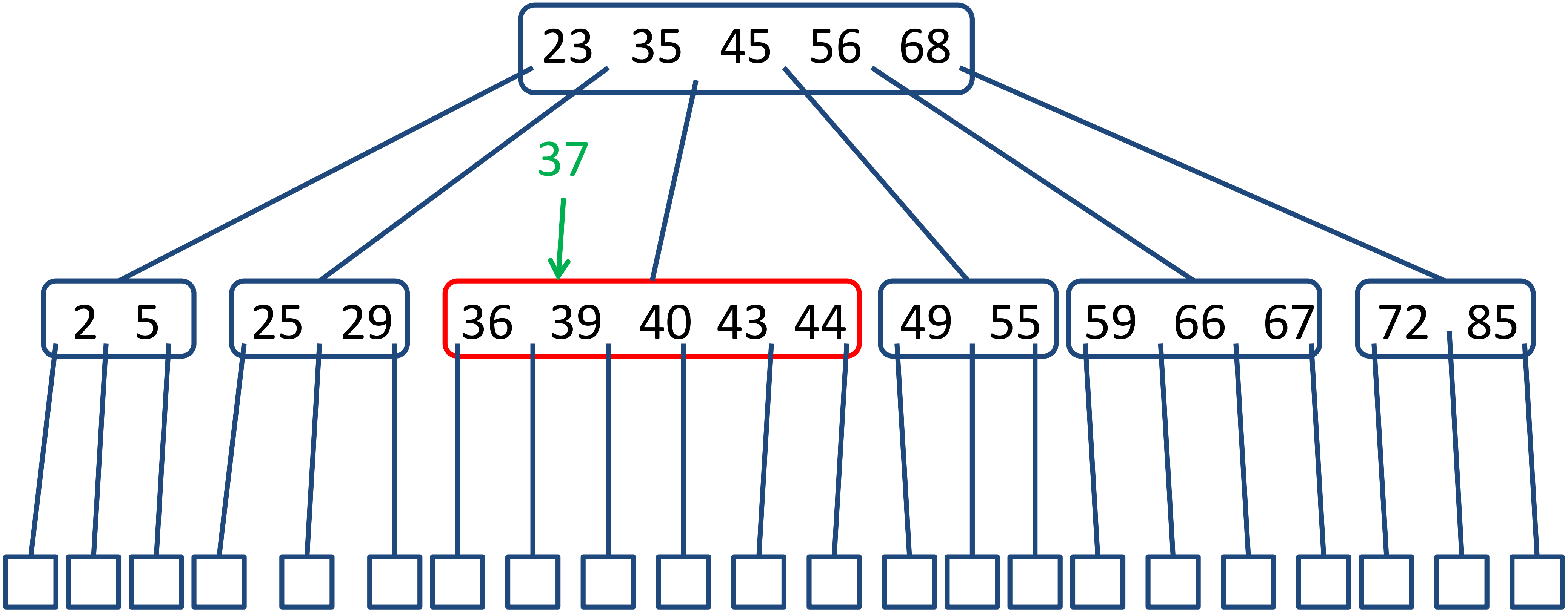
v ← parent of ***v***

}

B-Trees

B-Tree of order 6

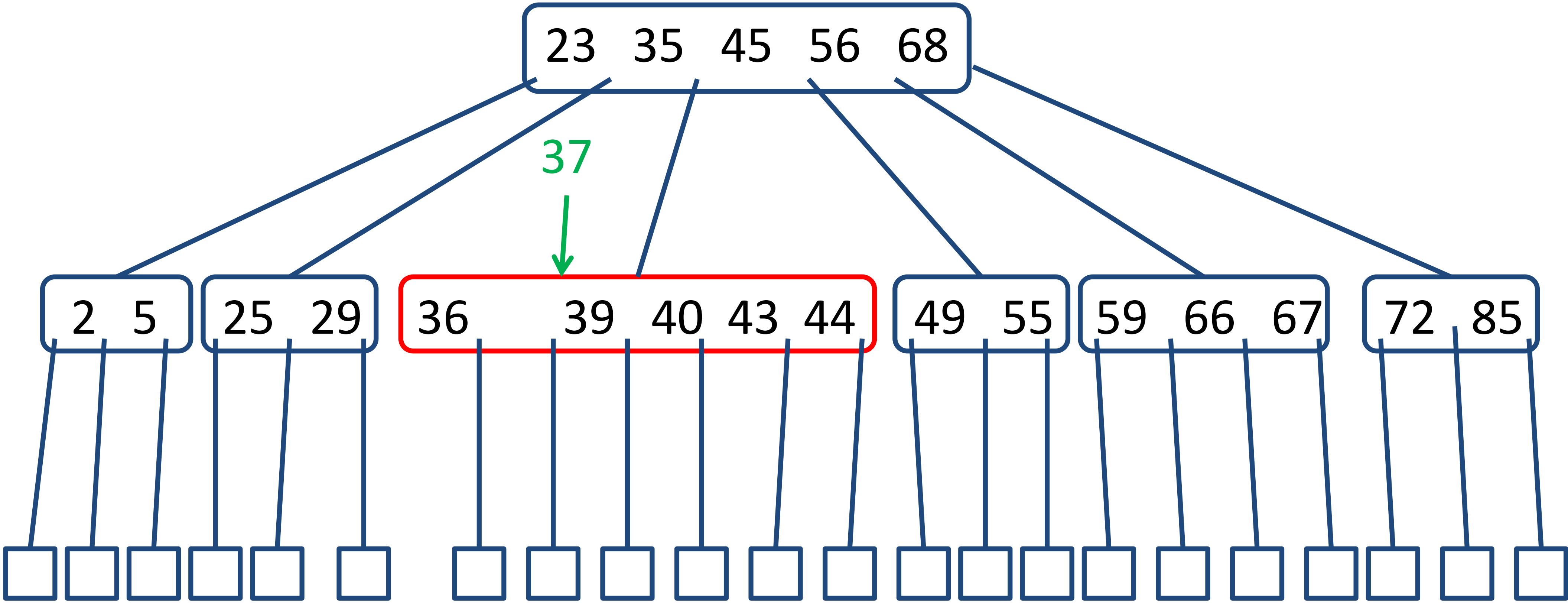
Put 38



B-Trees

B-Tree of order 6

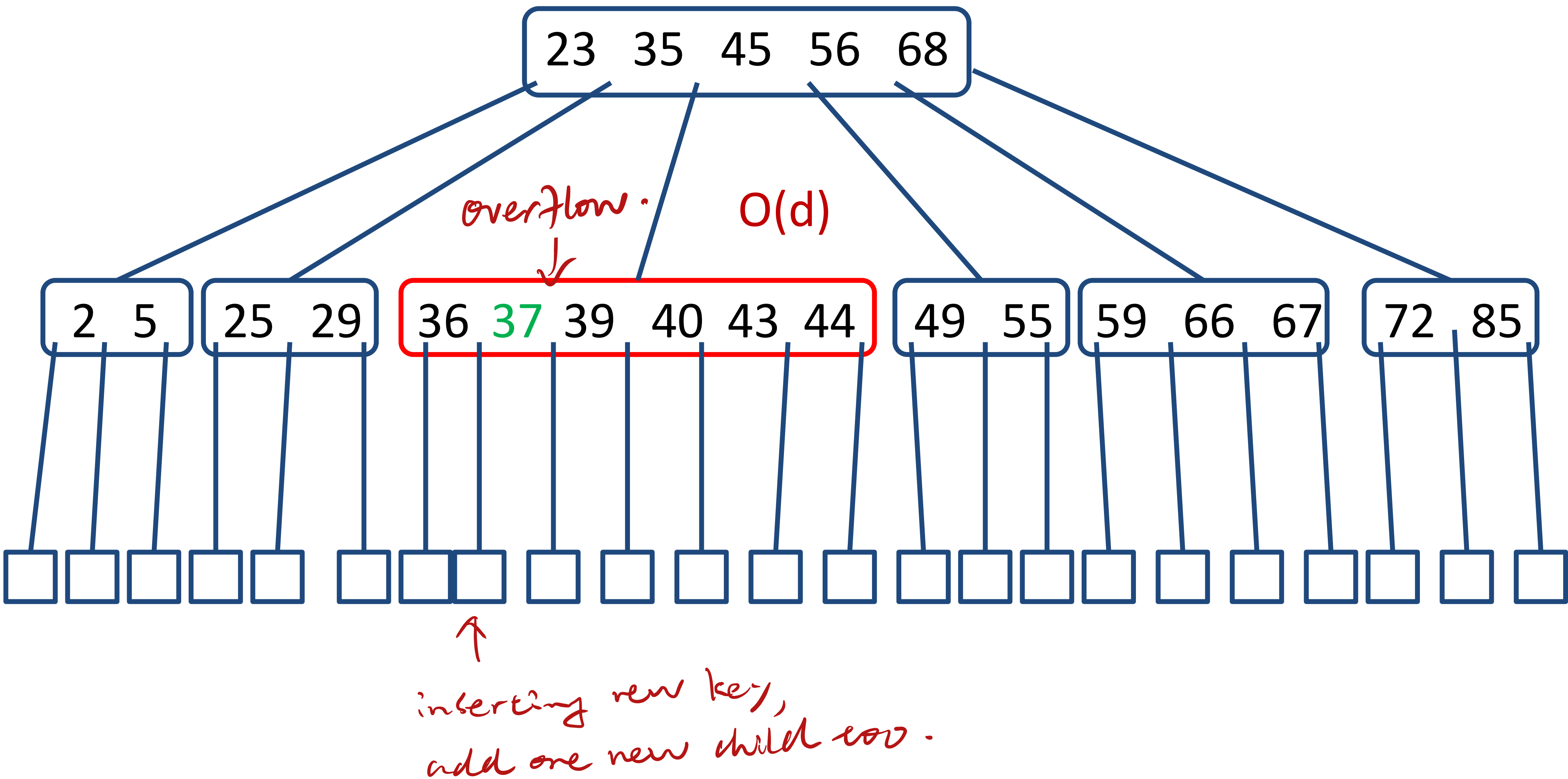
Put 38



B-Trees

B-Tree of order 6

Put 38



$$O\left[\underbrace{\log_2\left(\frac{m}{2}-1\right)}_{\text{max time searching each level}} \times \underbrace{\log_{\frac{m}{2}}\left(\frac{N}{\frac{m}{2}-1}\right)}_{\text{max height}}\right]$$

Algorithm ***put*** ***(r,k,o)***

In: Root *r* of a B-tree, data item (*k,o*)

Out: {Insert data item (*k,o*) in the B-tree

Search for ***k*** to find the lowest insertion internal node ***v*** } $O(\log d \times \log_d n)$

Add the new data item (***k***, ***o***) at node ***v*** } O(d)

while node *v* ***overflows*** **do {**

if ***v*** is the root **then** {

Create a new empty root and set as parent of ***v*** }.

Split ***v*** around the middle key ***k'***, move ***k'*** to parent, and update parent's children

v ← parent of ***v***

}

Algorithm ***put*** ***(r,k,o)***

In: Root *r* of a B-tree, data item (*k,o*)

Out: {Insert data item (*k,o*) in the B-tree

$O(\log d \times \log_d n)$

Search for ***k*** to find the lowest insertion internal node ***v***

Add the new data item (***k***, ***o***) at node ***v*** $O(d)$ c_1

while node *v* ***overflows*** **do {**

if ***v*** is the root **then**

Create a new empty root and set as parent of ***v***

Split ***v*** around the middle key ***k'***, move ***k'*** to parent, and update parent's children

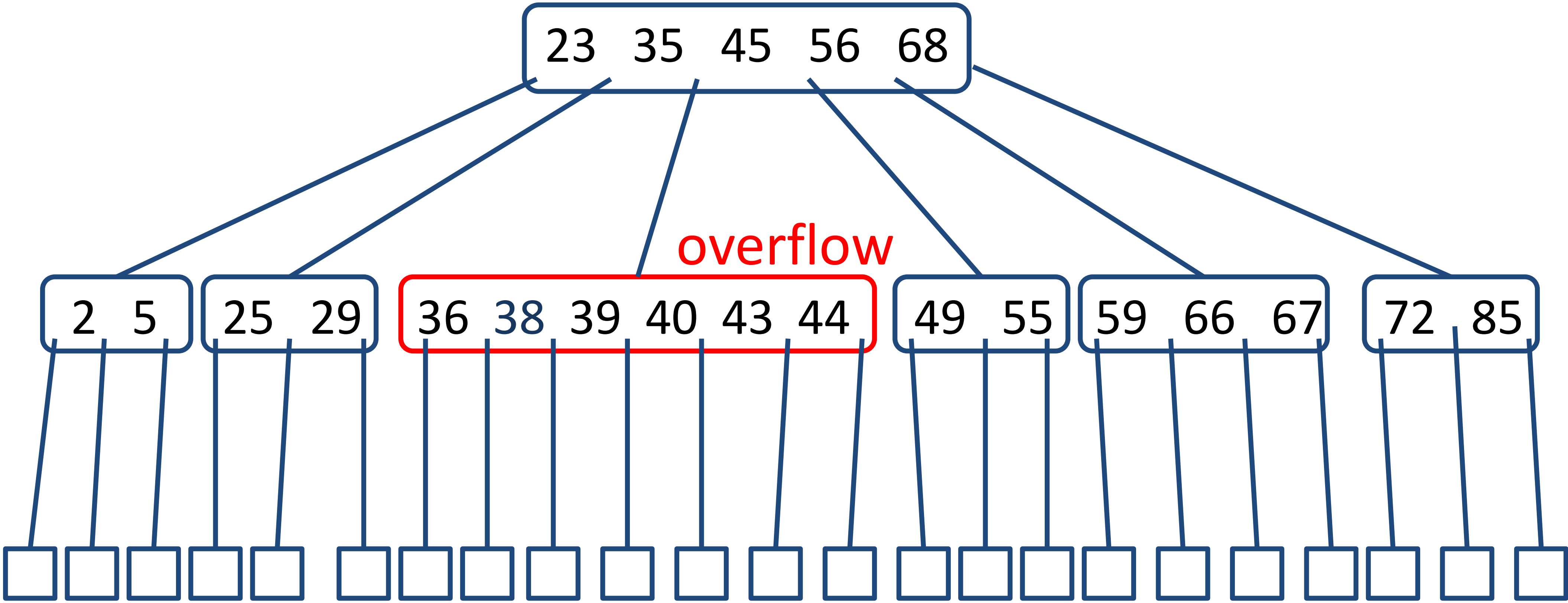
v ← parent of ***v***

}

B-Trees

B-Tree of order 6

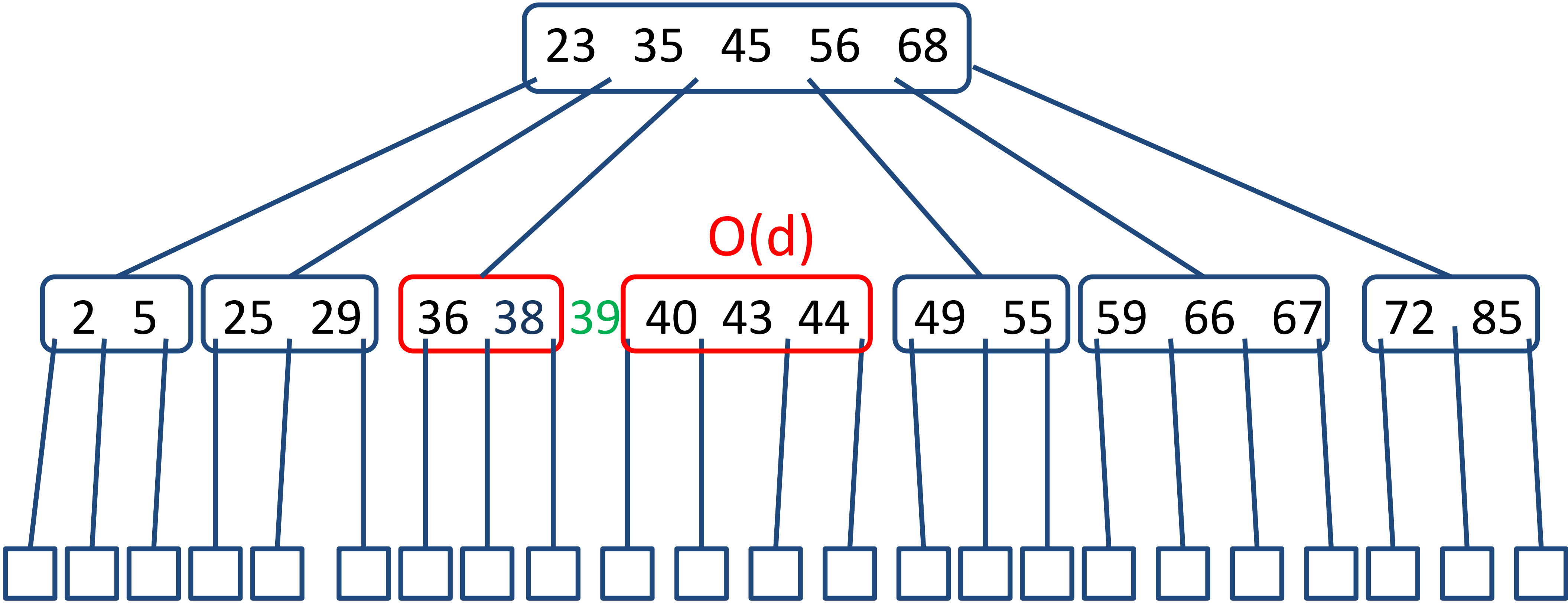
Put 38



B-Trees

B-Tree of order 6

Put 38

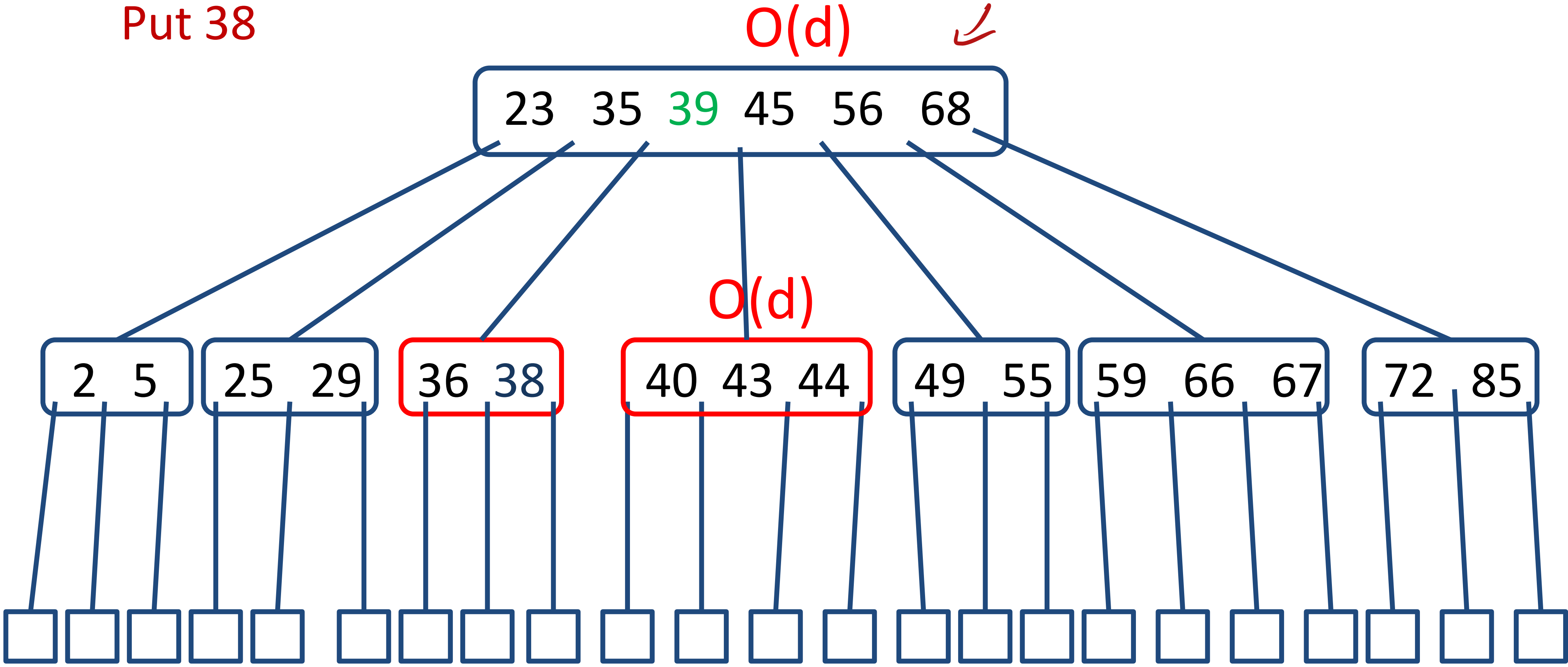


B-Trees

B-Tree of order 6

Put 38

overflow.



Algorithm ***put*** **(*r*,*k*,*o*)**

In: Root r of a B-tree, data item (k,o)

Out: {Insert data item (k,o) in the B-tree

O(log d × $\log_d$ n)

Search for $\boldsymbol{k}$ to find the lowest insertion internal node $\boldsymbol{v}$

Add the new data item $(\boldsymbol{k}, \boldsymbol{o})$ at node $\boldsymbol{v}$ O(d)

while node v ***overflow***s **do {**

O(d)

if $\boldsymbol{v}$ is the root **then**

Create a new empty root and set as parent of $\boldsymbol{v}$

Split $\boldsymbol{v}$ around the middle key $\boldsymbol{k'}$, move $\boldsymbol{k'}$ to parent, and update parent's children

$\boldsymbol{v} \leftarrow$ parent of $\boldsymbol{v}$

}

Algorithm ***put*** **(*r*,*k*,*o*)**

In: Root r of a B-tree, data item (k,o)

Out: {Insert data item (k,o) in the B-tree

height.

$O(\log d \times \log_d n)$

Search for $\boldsymbol{k}$ to find the lowest insertion internal node $\boldsymbol{v}$

Add the new data item $(\boldsymbol{k}, \boldsymbol{o})$ at node $\boldsymbol{v}$ O(d)

while node v ***overflows*** **do {**

O(d)

if $\boldsymbol{v}$ is the root **then**

$O(d \log_d n)$

Create a new empty root and set as parent of $\boldsymbol{v}$

Split $\boldsymbol{v}$ around the middle key $\boldsymbol{k'}$, move $\boldsymbol{k'}$ to parent, and update parent's children

$\boldsymbol{v} \leftarrow$ parent of $\boldsymbol{v}$

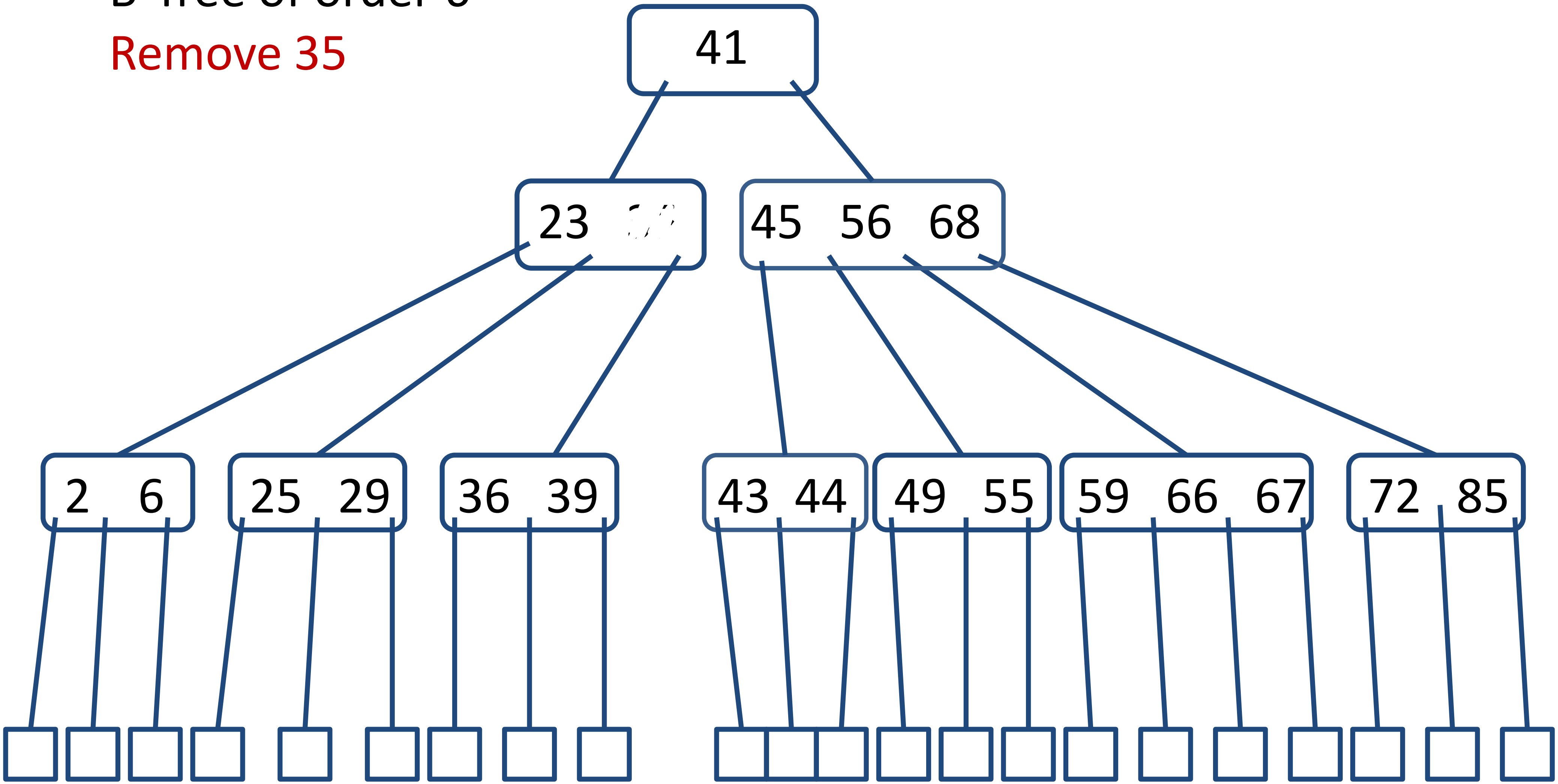
}

Time complexity of put is $O(d \log_d n)$

B-Trees

B-Tree of order 6

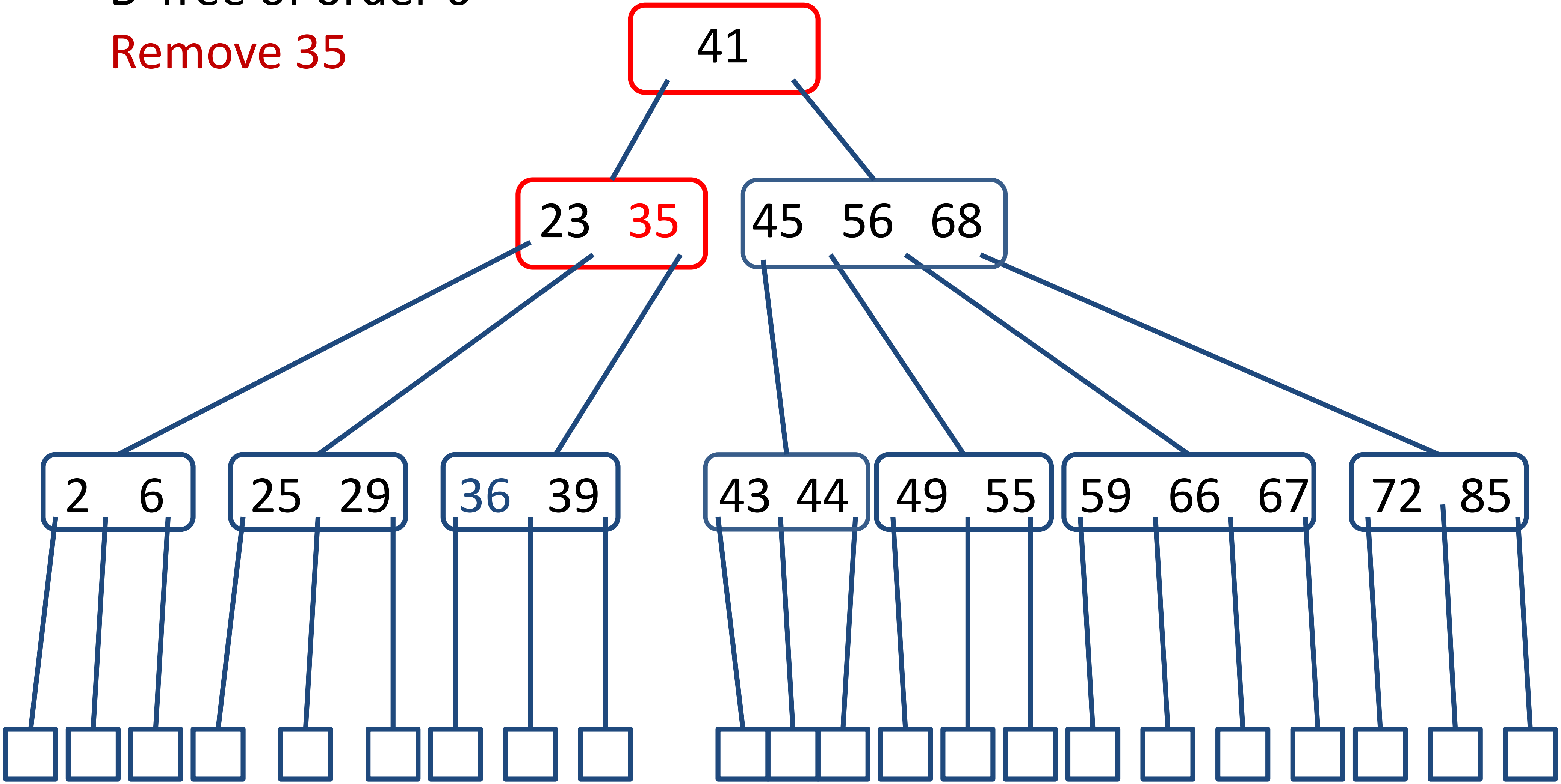
Remove 35



B-Trees

B-Tree of order 6

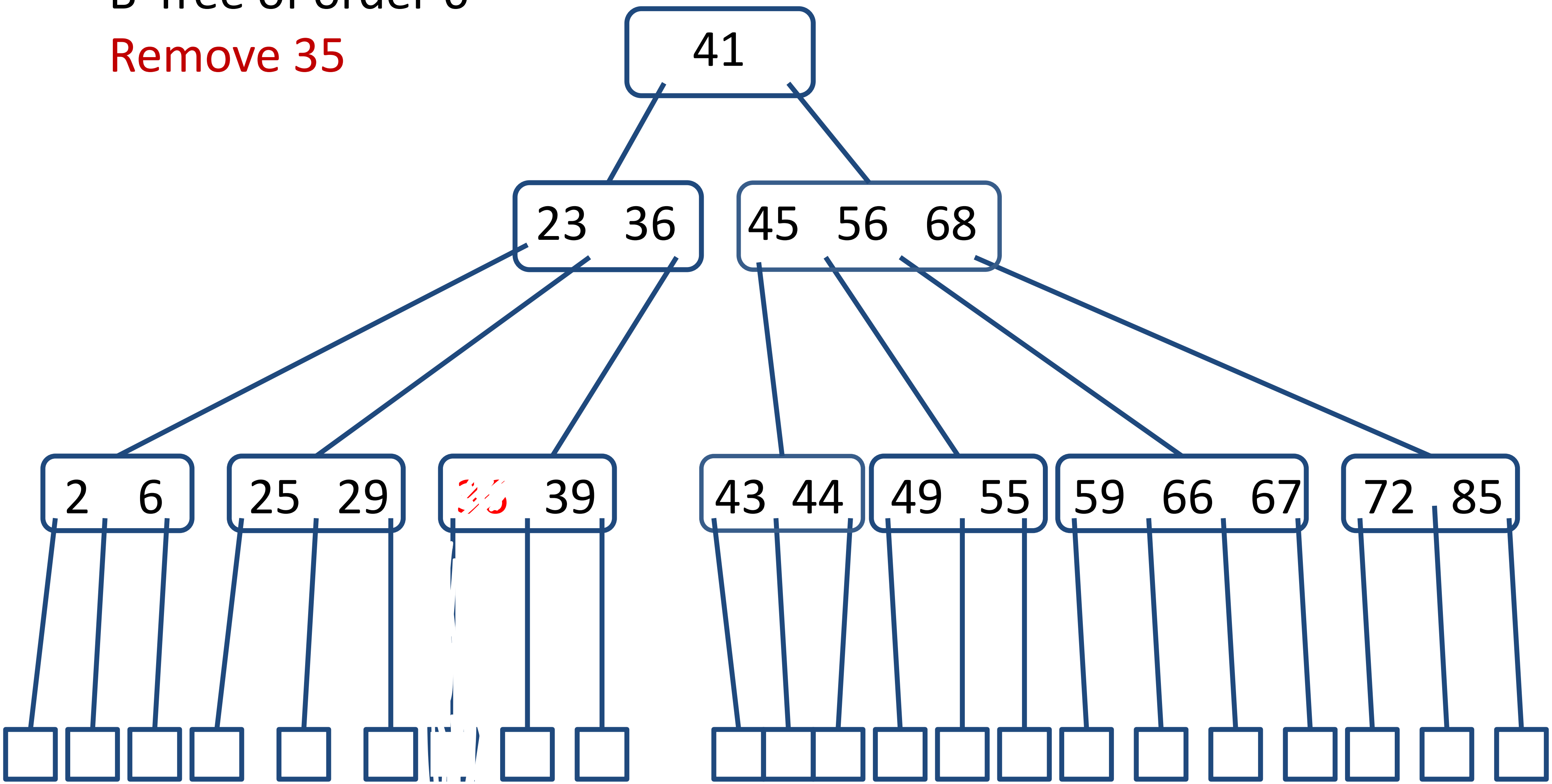
Remove 35



B-Trees

B-Tree of order 6

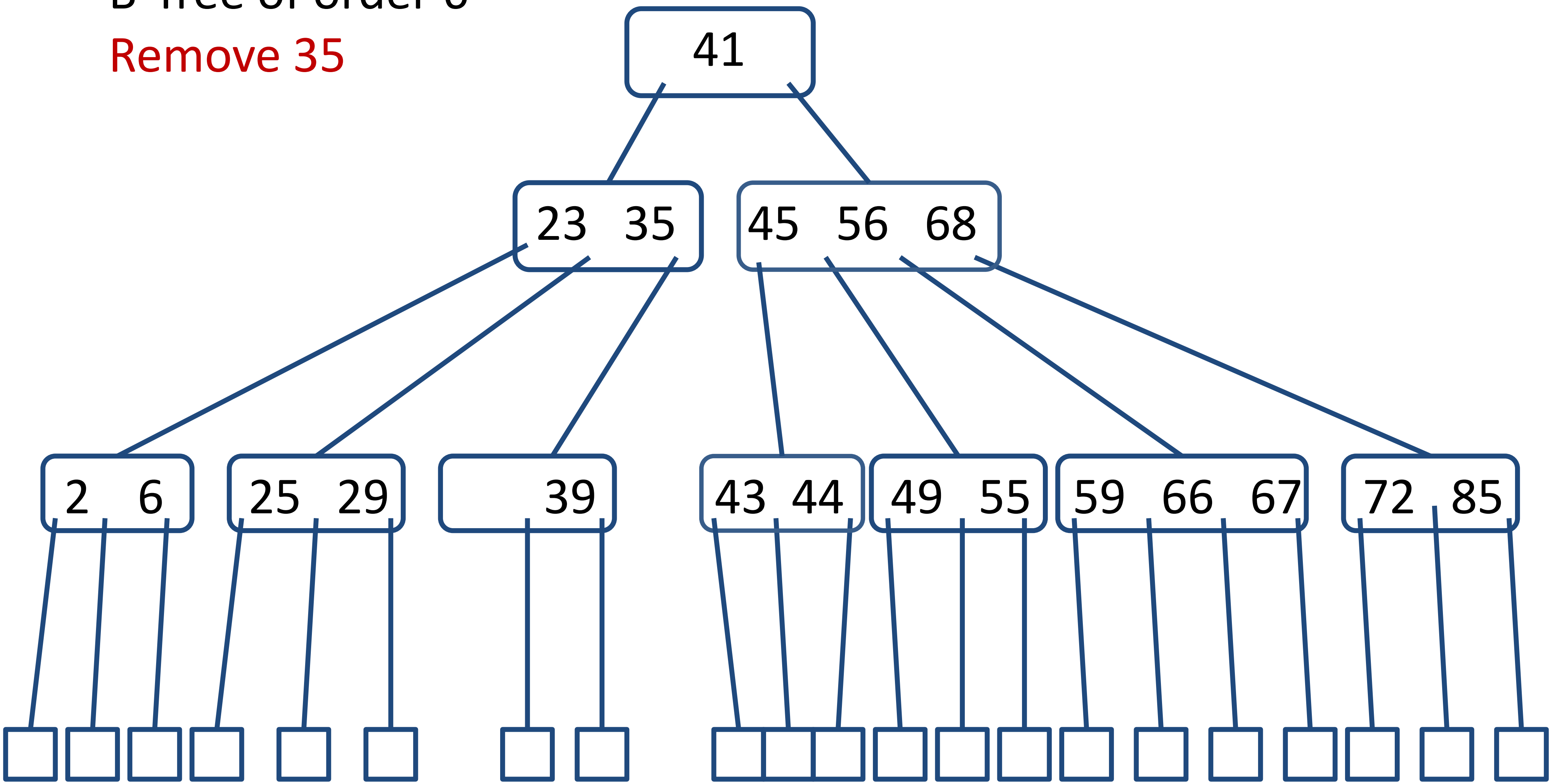
Remove 35



B-Trees

B-Tree of order 6

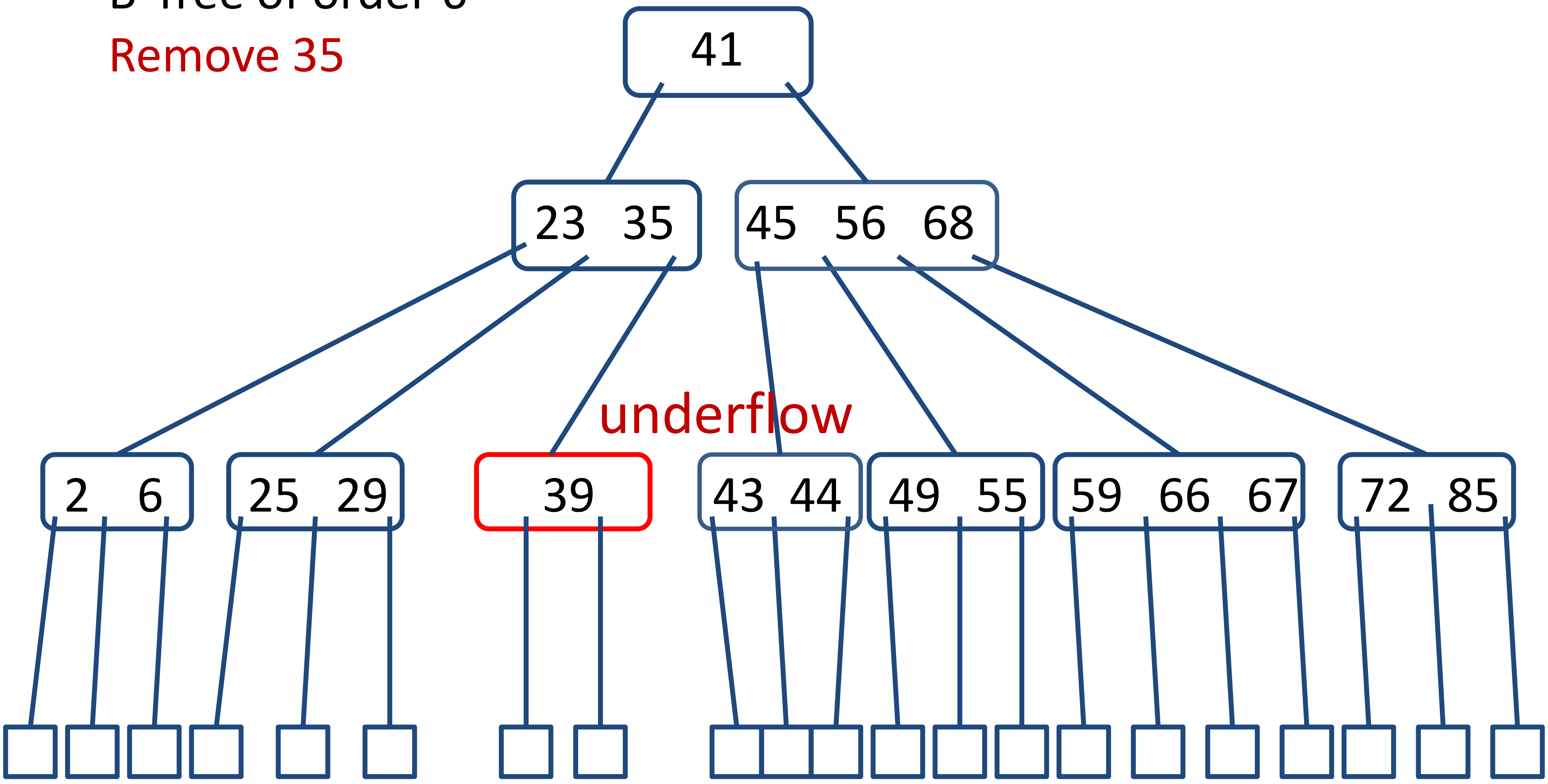
Remove 35



B-Trees

B-Tree of order 6

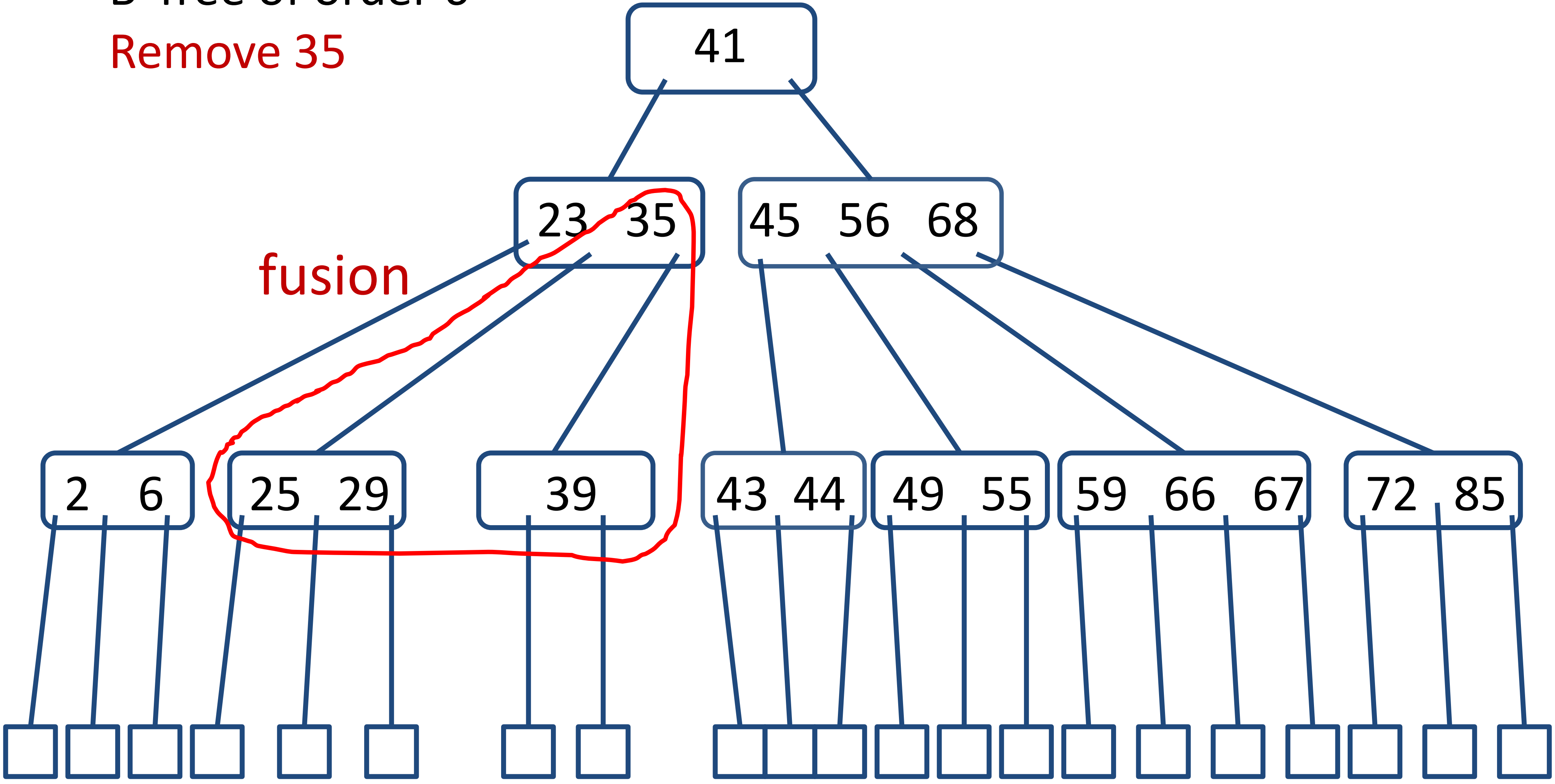
Remove 35



B-Trees

B-Tree of order 6

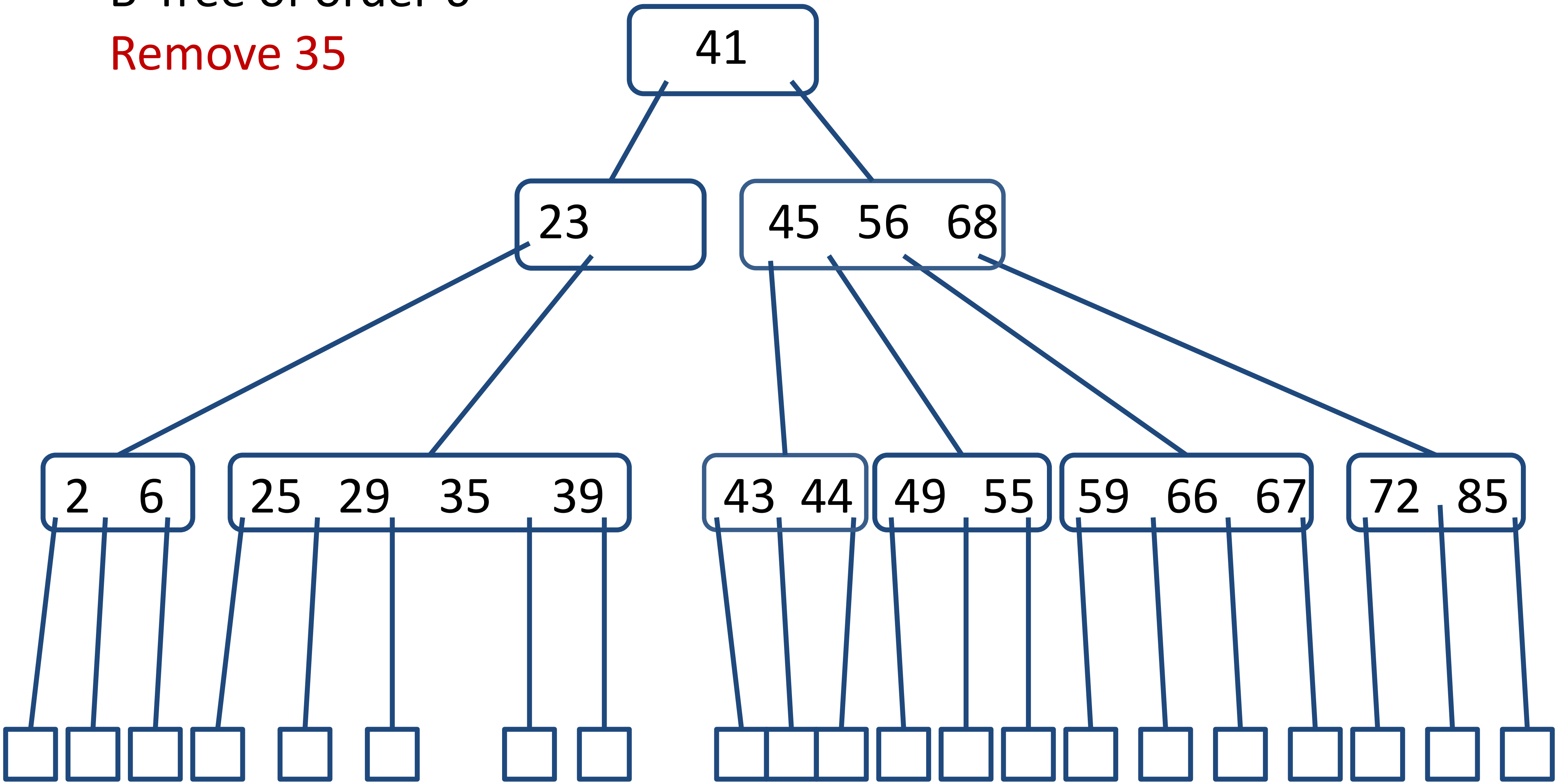
Remove 35



B-Trees

B-Tree of order 6

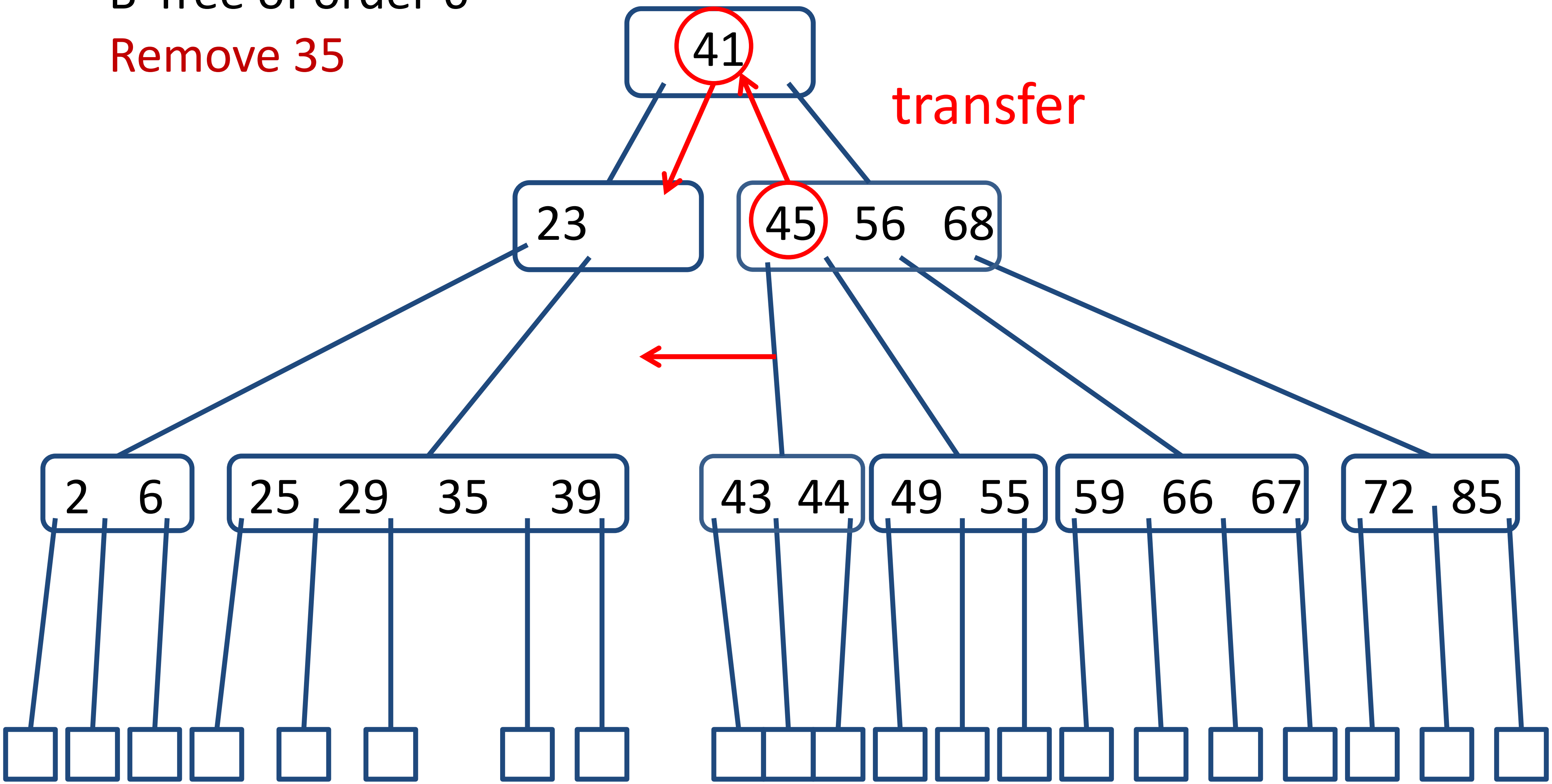
Remove 35



B-Trees

B-Tree of order 6

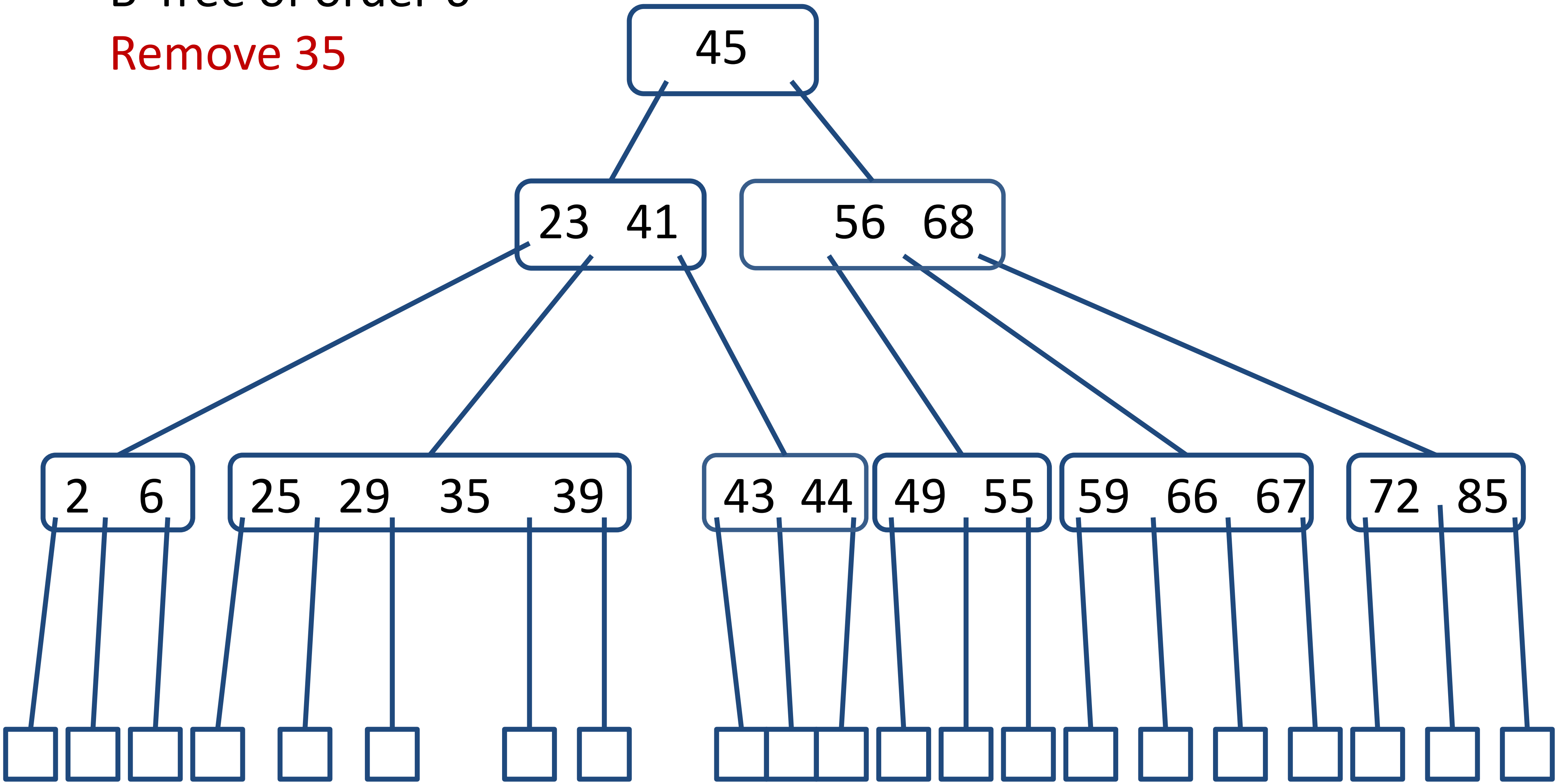
Remove 35



B-Trees

B-Tree of order 6

Remove 35



Algorithm ***remove(r,k)***

In: Root r of a B-tree, key k

Out: {remove data item with key k from the tree}

Find the node ***v*** storing key ***k***

Remove (***k***, ***o***) from ***v*** replacing it with successor if needed

while node *v* ***underflows*** **do {**

if ***v*** is the root then

make the first child of ***v*** the new root

else if a sibling has more than $\lceil d/2 \rceil$ keys **then**

perform a transfer operation

else {

perform a fusion operation

v ← parent of ***v***

}

}

Algorithm ***remove*(*r*,*k*)**

In: Root r of a B-tree, key k

Out: {remove data item with key k from the tree}

Find the node ***v*** storing key ***k*** } $O(\log d \times \log_d n)$

Remove (***k***, ***o***) from ***v*** replacing it with successor if needed

while node *v* ***underflow*s** **do {**

if ***v*** is the root then

make the first child of ***v*** the new root

else if a sibling has at least $\lceil d/2 \rceil$ keys **then**

perform a transfer operation

else {

perform a fusion operation

v ← parent of ***v***

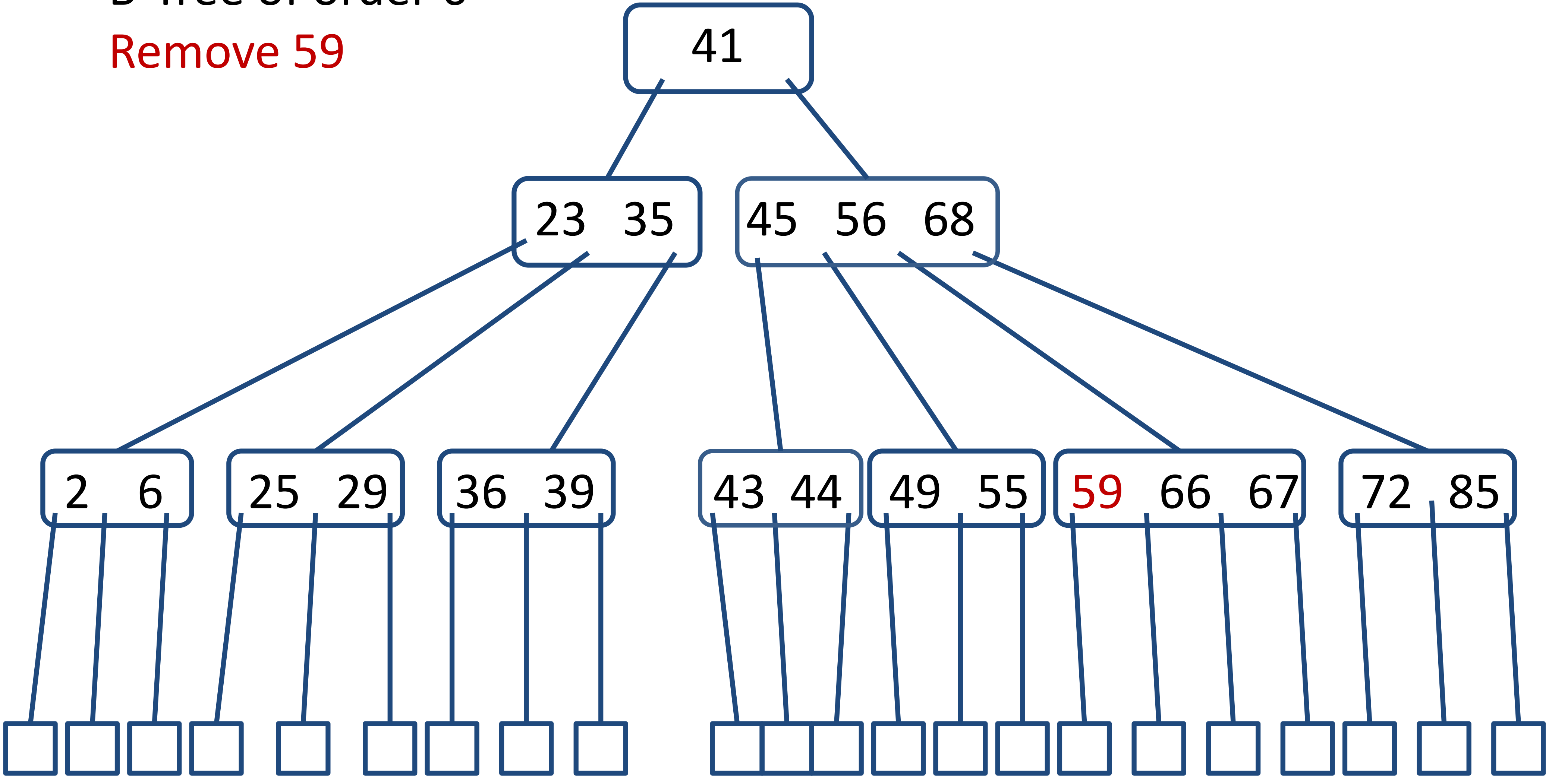
}

}

B-Trees

B-Tree of order 6

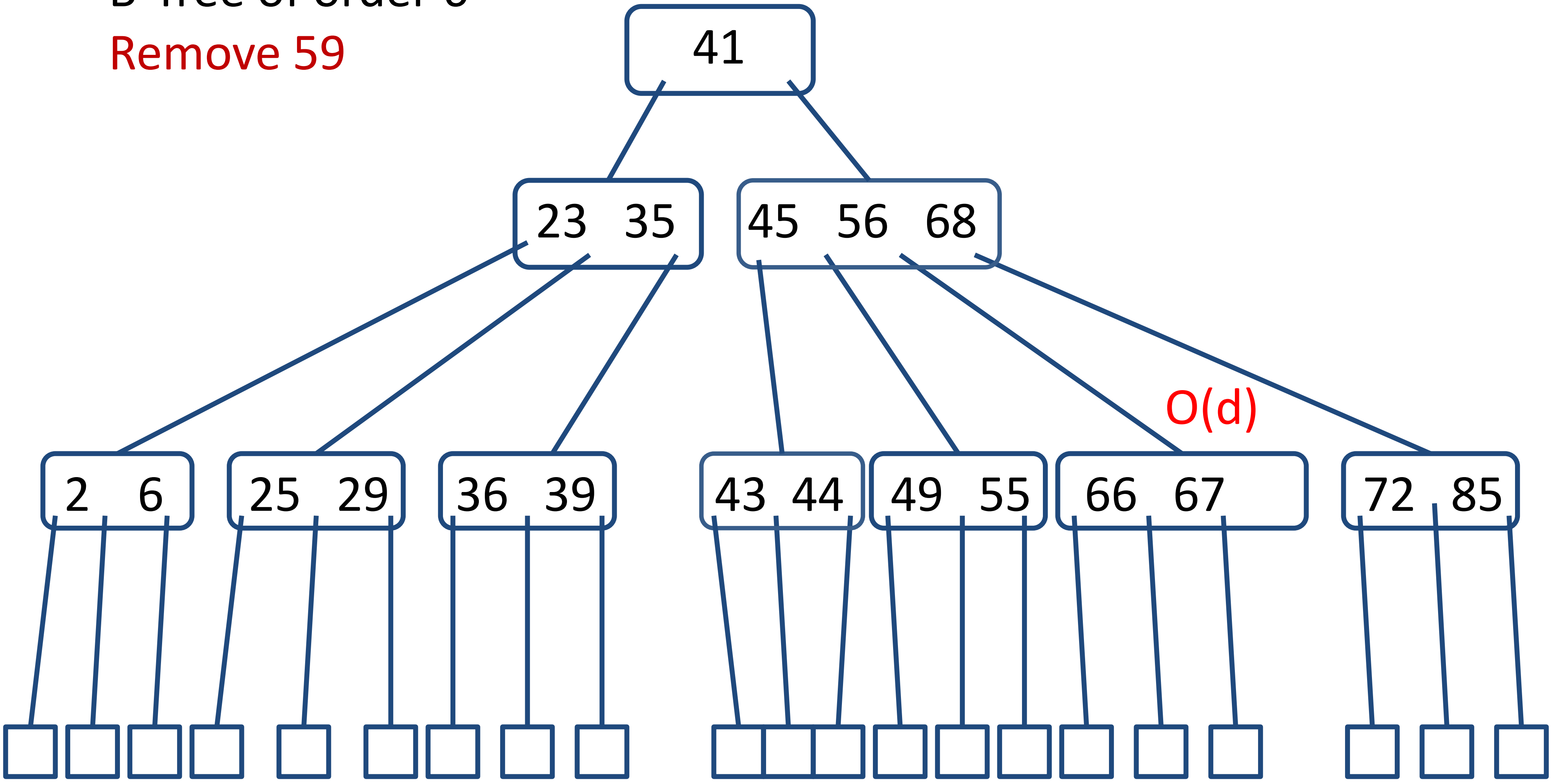
Remove 59



B-Trees

B-Tree of order 6

Remove 59



Algorithm ***remove(r,k)***

In: Root r of a B-tree, key k

Out: {remove data item with key k from the tree}

Find the node ***v*** storing key ***k*** } O(log d × $\log_d$ n)

Remove (***k***, ***o***) from ***v*** replacing it with successor if needed }

while node v ***underflows*** **do** { O(d)

if ***v*** is the root then

make the first child of ***v*** the new root r:= r. child[0]

else { **if** a sibling has at least ⌈d/2⌉ keys **then** {

perform a transfer operation }

else {

perform a fusion operation

v ← parent of ***v***

} }

}

Algorithm ***remove*(*r*,*k*)**

In: Root r of a B-tree, key k

Out: {remove data item with key k from the tree}

Find the node ***v*** storing key ***k*** O(log d × log$_d$ n)

Remove (***k***, ***o***) from ***v*** replacing it with successor if needed

while node v ***underflows*** **do {** O(d + log d × log$_d$ n)

if ***v*** is the root then

make the first child of ***v*** the new root

else if a sibling has at least $\lceil d/2 \rceil$ keys **then**

perform a transfer operation

else {

perform a fusion operation

v ← parent of ***v***

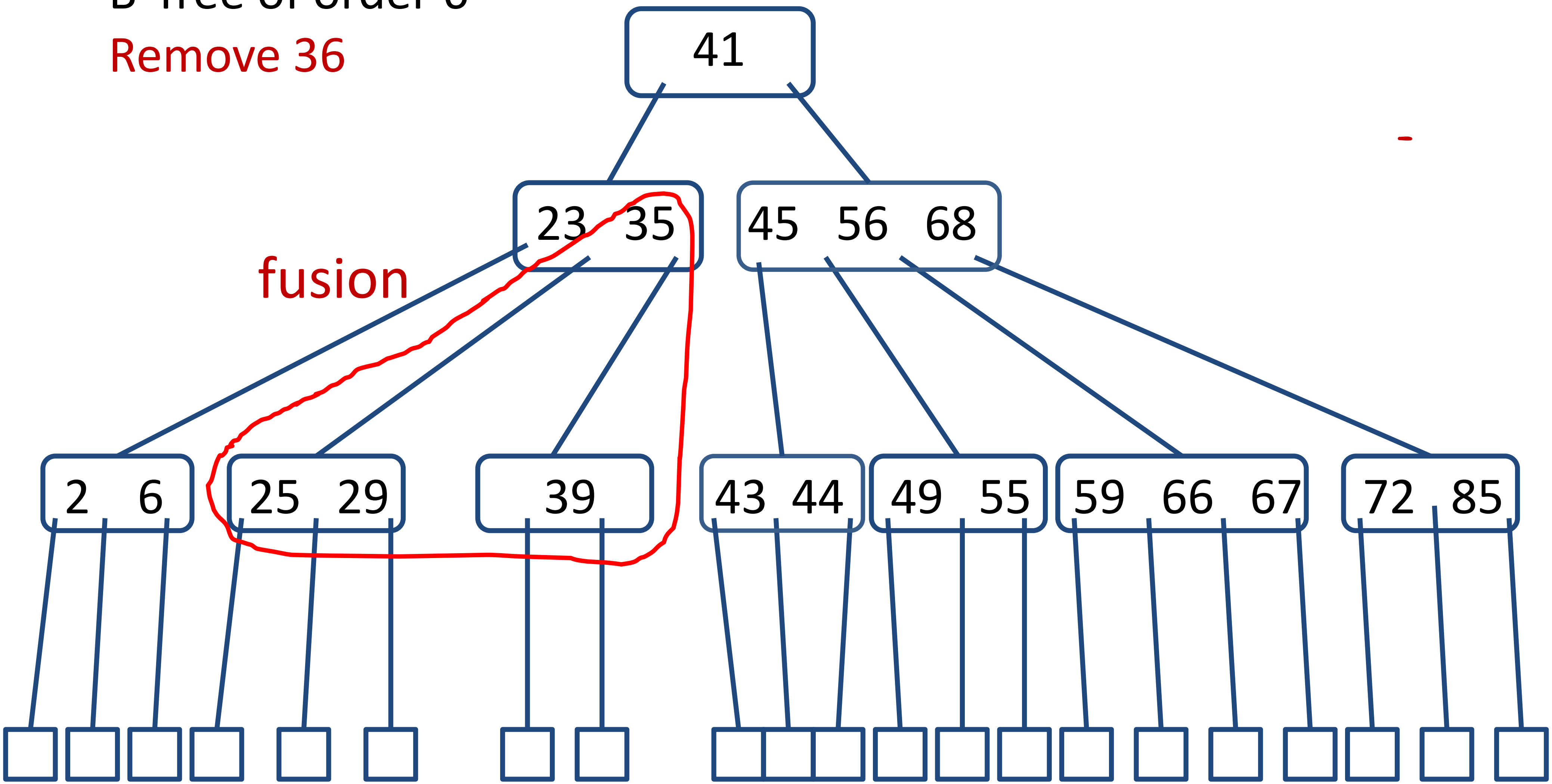
}

}

B-Trees

B-Tree of order 6

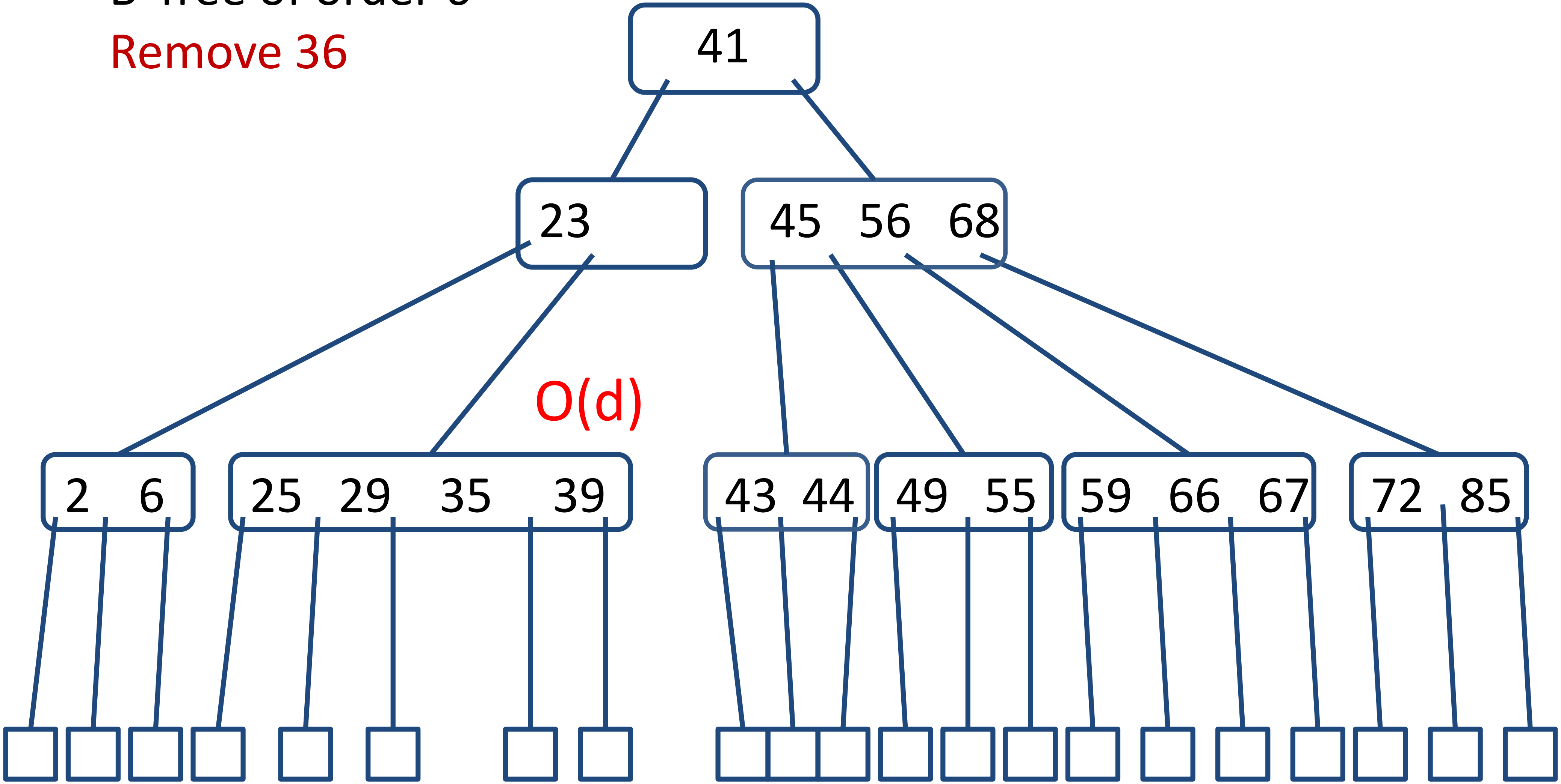
Remove 36



B-Trees

B-Tree of order 6

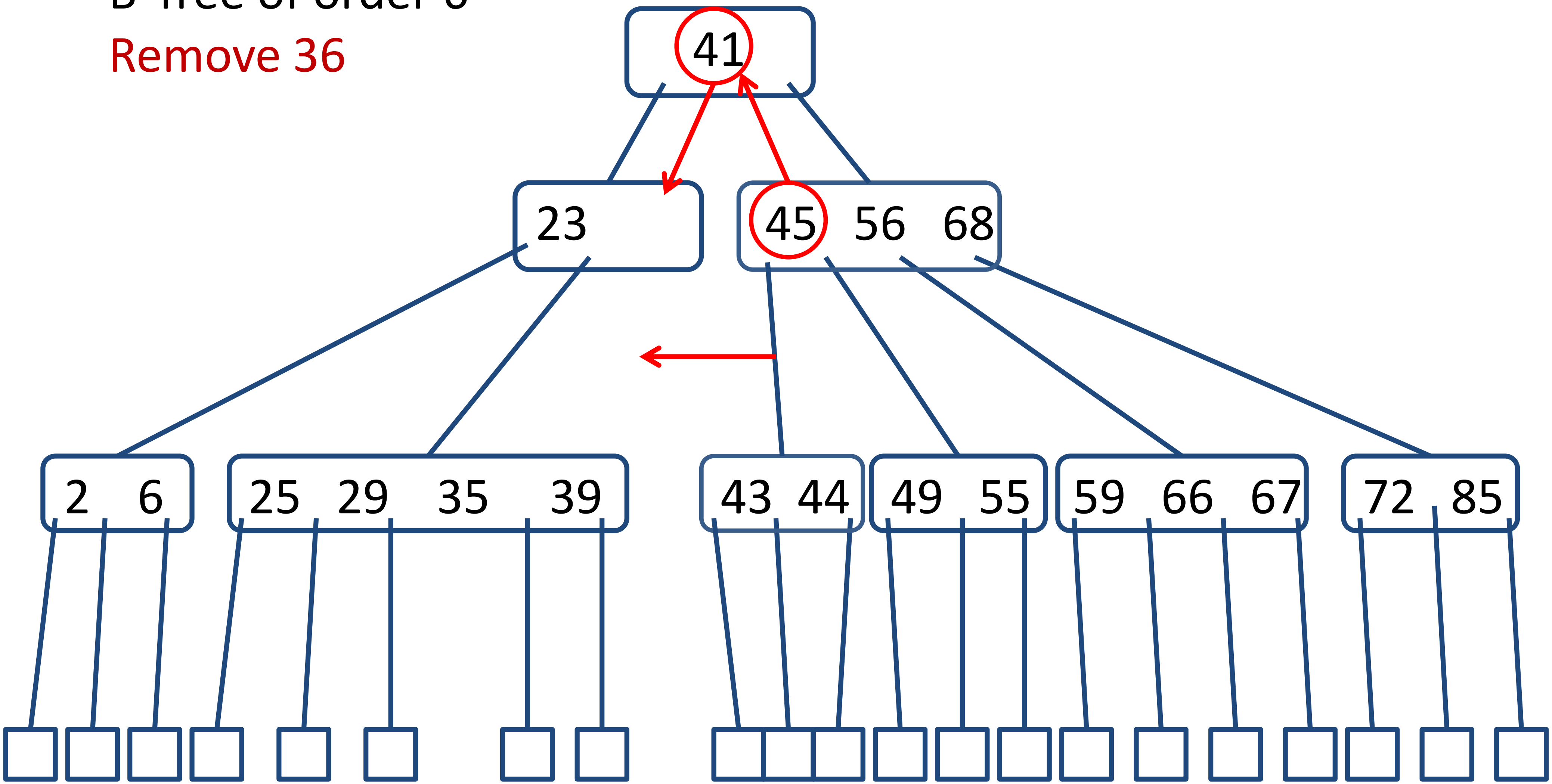
Remove 36



B-Trees

B-Tree of order 6

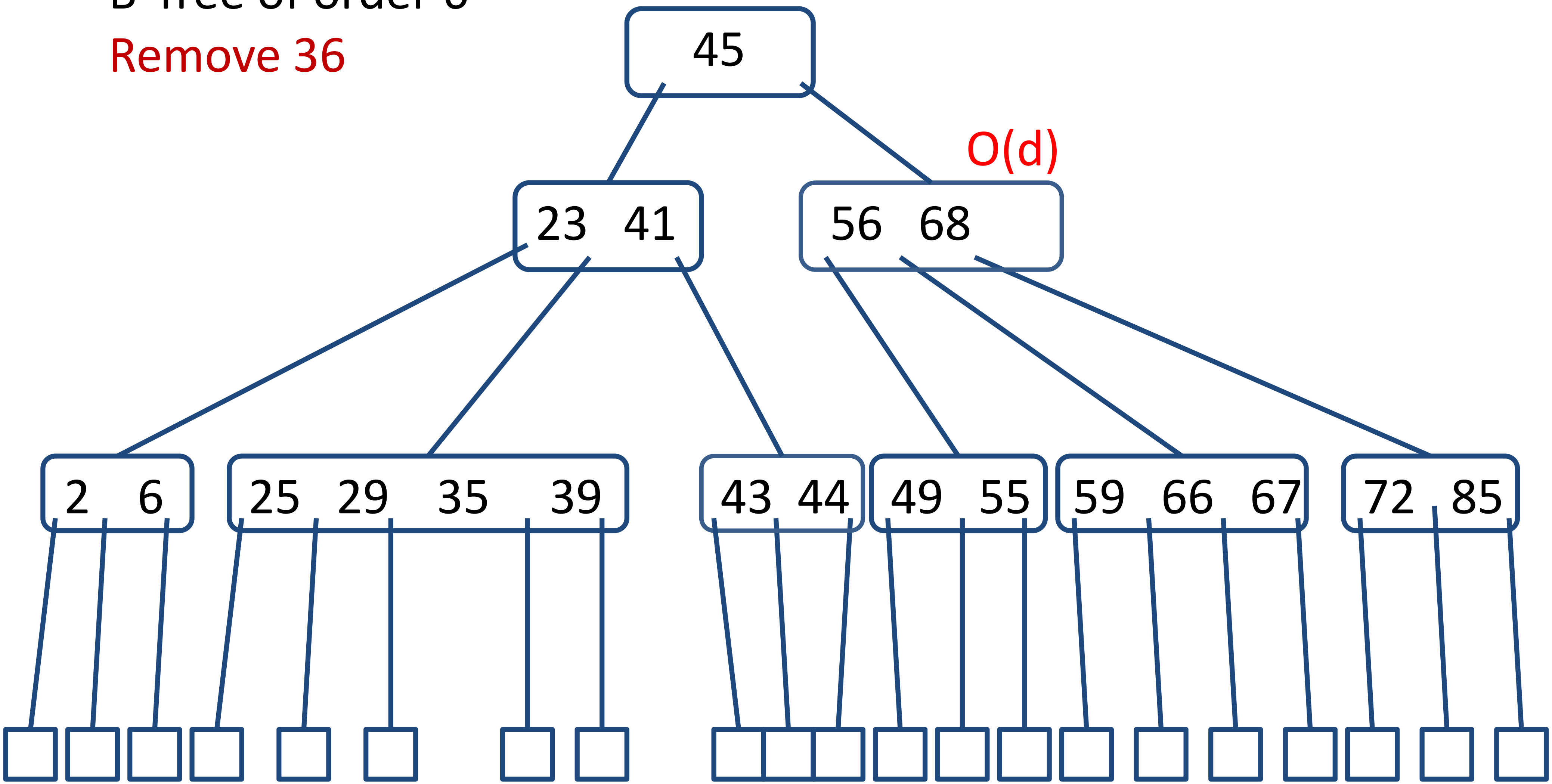
Remove 36



B-Trees

B-Tree of order 6

Remove 36



Algorithm ***remove*(*r*,*k*)**

In: Root r of a B-tree, key k

Out: {remove data item with key k from the tree}

Find the node ***v*** storing key ***k*** } O(log d × $\log_d$ n)

Remove (***k***, ***o***) from ***v*** replacing it with successor if needed } O(d + log d × $\log_d$ n)

while node *v* ***underflows*** **do** {

if ***v*** is the root then

make the first child of ***v*** the new root

else if a sibling has at least ⌈d/2⌉ keys **then**

perform a transfer operation

else {

perform a fusion operation

v ← parent of ***v***

}

} O(d)

Algorithm ***remove*(*r*,*k*)** Time complexity $O(d \log_d n)$

In: Root *r* of a B-tree, key *k*

Out: {remove data item with key *k* from the tree}

Find the node ***v*** storing key ***k*** $O(\log d \times \log_d n)$

Remove (***k***, ***o***) from ***v*** replacing it with successor if needed $O(d + \log d \times \log_d n)$

$O(d \log_d n)$

while node *v* ***underflows*** **do** {

if ***v*** is the root then

make the first child of ***v*** the new root

else if a sibling has at least $\lceil d/2 \rceil$ keys **then**

perform a transfer operation

else {

perform a fusion operation

v ← parent of ***v***

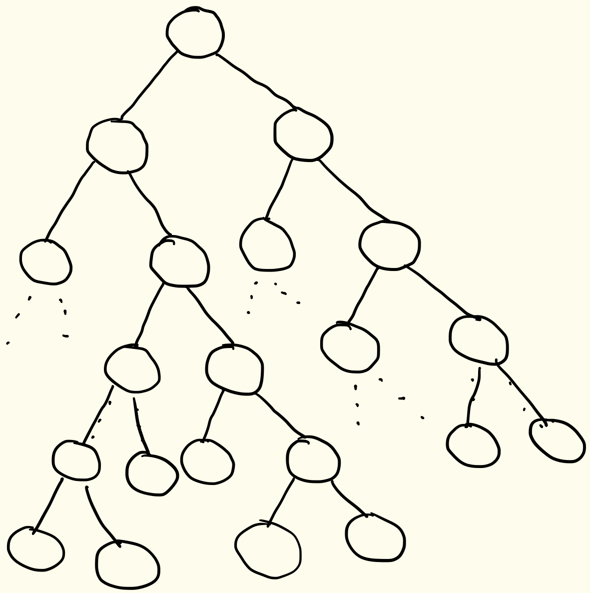
}

}

$O(d)$

Collection of 10^9 records

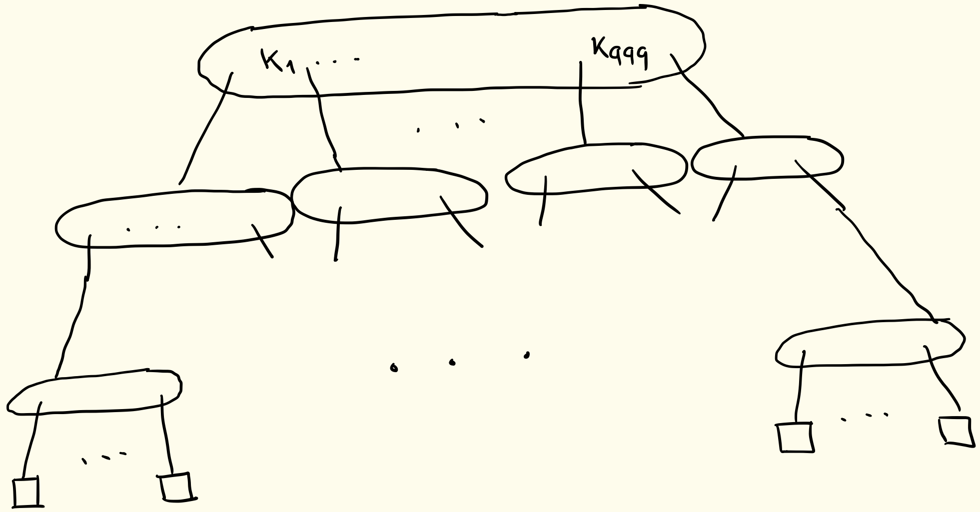
AVL tree



Height: $O(\log_2 n)$

$\log_2 10^9 \approx 30$

B-tree of degree 10^3

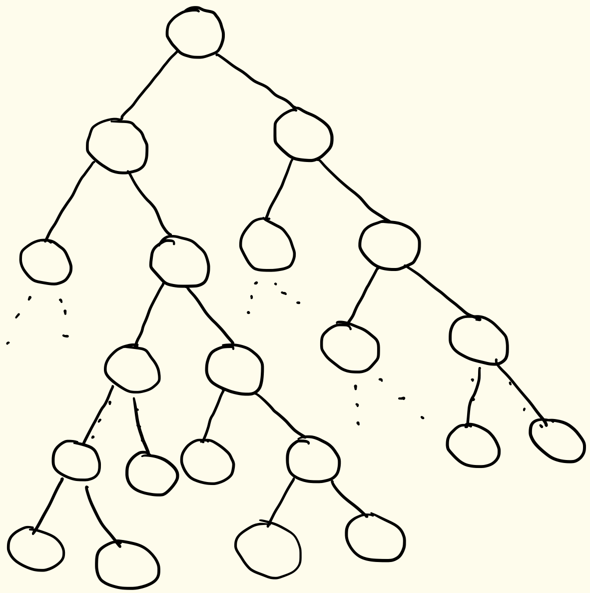


Height: $O(\log_d n)$

$\log_{1000} 10^9 = 3$

Collection of 10^9 records

AVL tree



Height: $O(\log_2 n)$

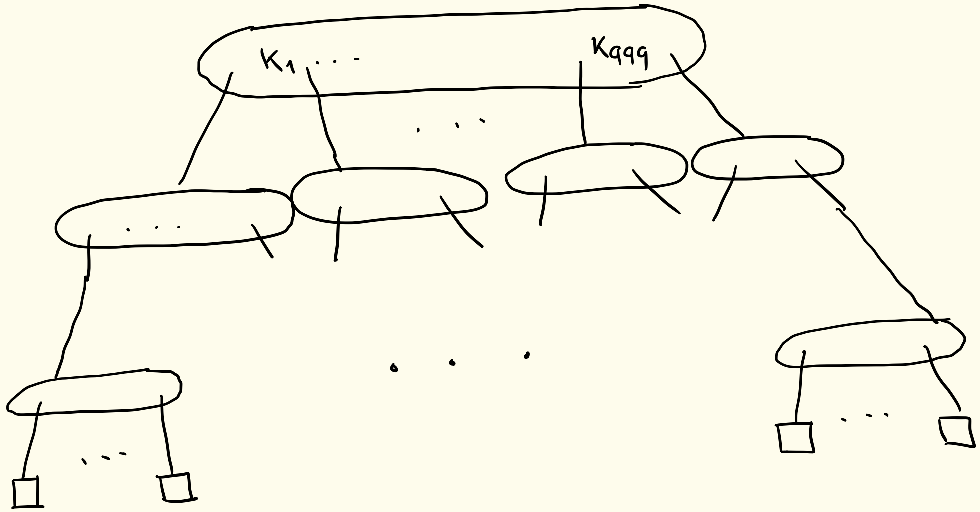
$\log_2 10^9 \approx 30$

Number of comparisons to find a key ≈ 30

B-tree of degree 10^3

Binary search: $O(\log_2 d)$

$\log_2 10^3 \approx 10$



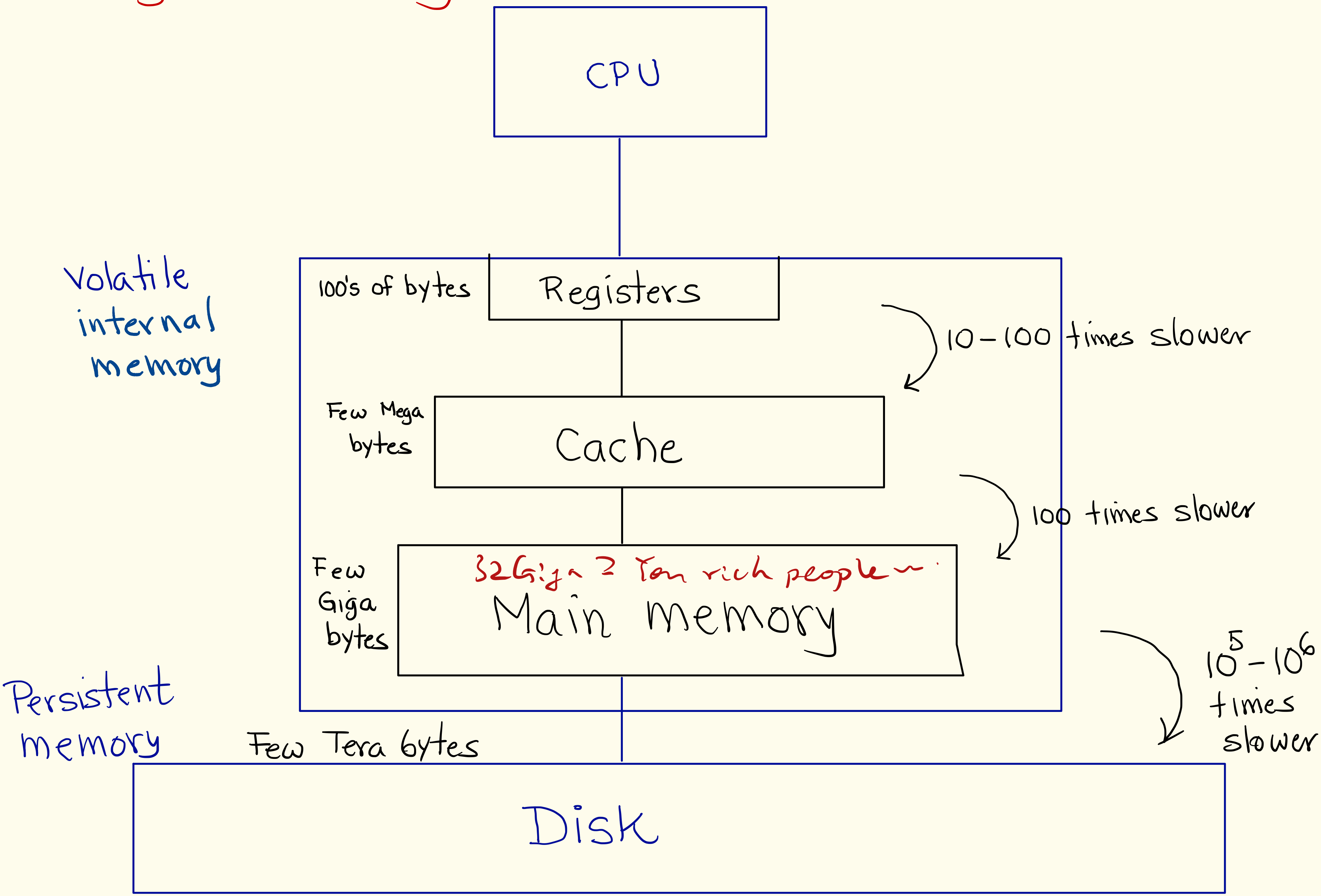
Height: $O(\log_d n)$

$\log_{1000} 10^9 = 3$

$O(\log d \times \log_d n) = \log 10^3 \times \log_{10^3} 10^9$

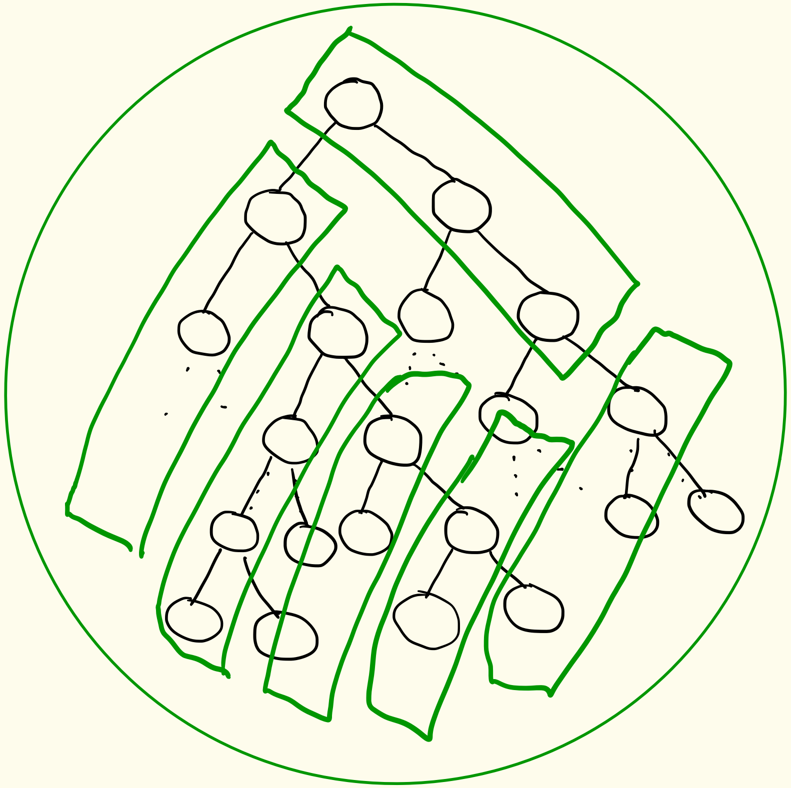
≈ 30

Memory Hierarchy



Collection of 10^9 records

AVL tree

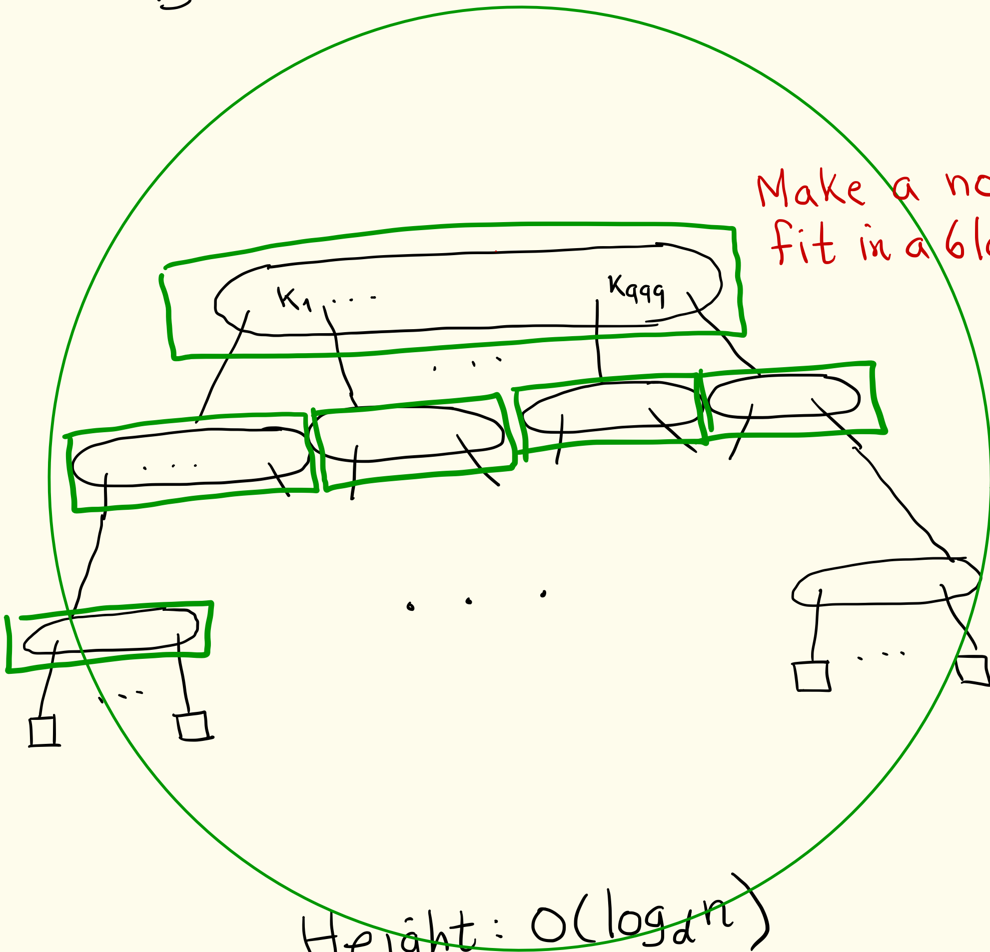


Height: $O(\log_2 n)$

$\log_2 10^9 \quad 30$

B-tree of degree 10^3

Make a node fit in a block



Height: $O(\log_d n)$

$\log_{1000} 10^9 = 3$

	AVL	B-tree
Number of disk accesses:	30	$\underline{3}$

↑ height

So a B-tree is much better.

the tree.

Disk Blocks

- Consider the problem of maintaining a large collection of items that does not fit in main memory, such as a typical database.
- In this context, we refer to the external memory is divided into blocks, which we call **disk blocks**.
- The transfer of a block between external memory and primary memory is a **disk transfer** or **I/O**.
- There is a great time difference that exists between main memory accesses and disk accesses
- Thus, we want to minimize the number of disk transfers needed to perform a query or update. We refer to this count as the **I/O complexity** of the algorithm involved.

Memory Hierarchies

- Computers have a hierarchy of different kinds of memories, which vary in terms of their size and distance from the CPU.
- Closest to the CPU are the internal **registers**. Access to such locations is very fast, but there are relatively few such locations.
- At the second level in the hierarchy are the memory **caches**.
- At the third level in the hierarchy is the **internal memory**, which is also known as main memory or core memory.
- Another level in the hierarchy is the **external memory**, which usually consists of disks.