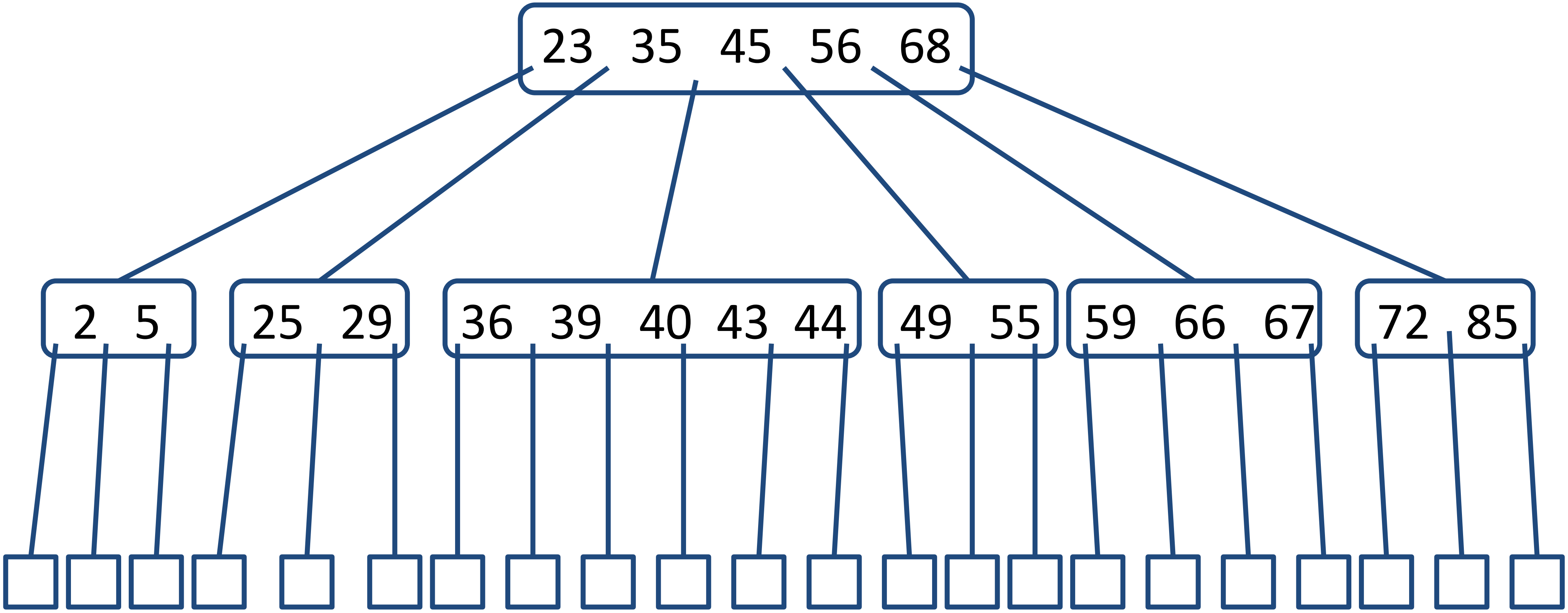
B-Trees

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

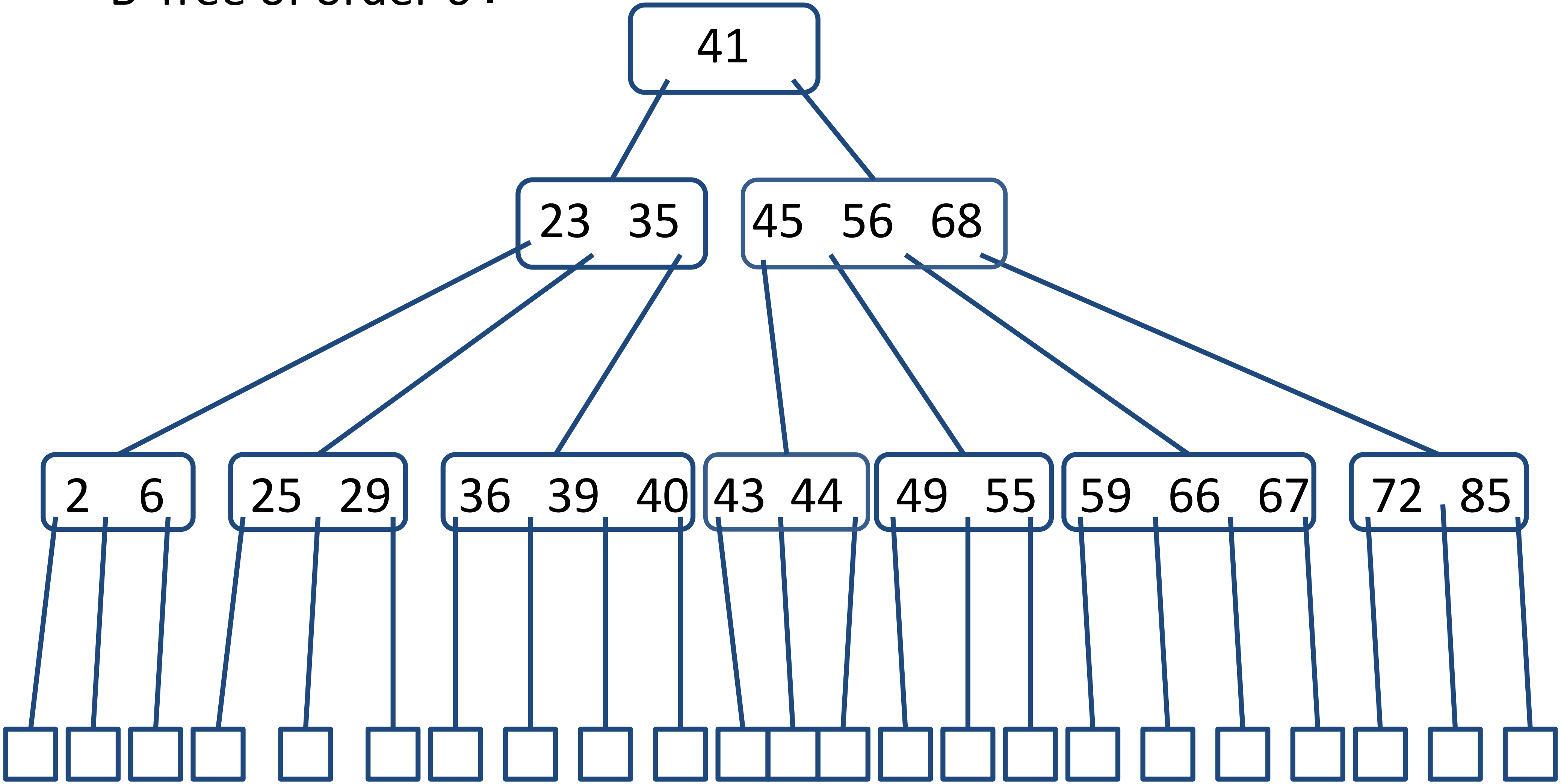
B-Trees

B-Tree of order 6?



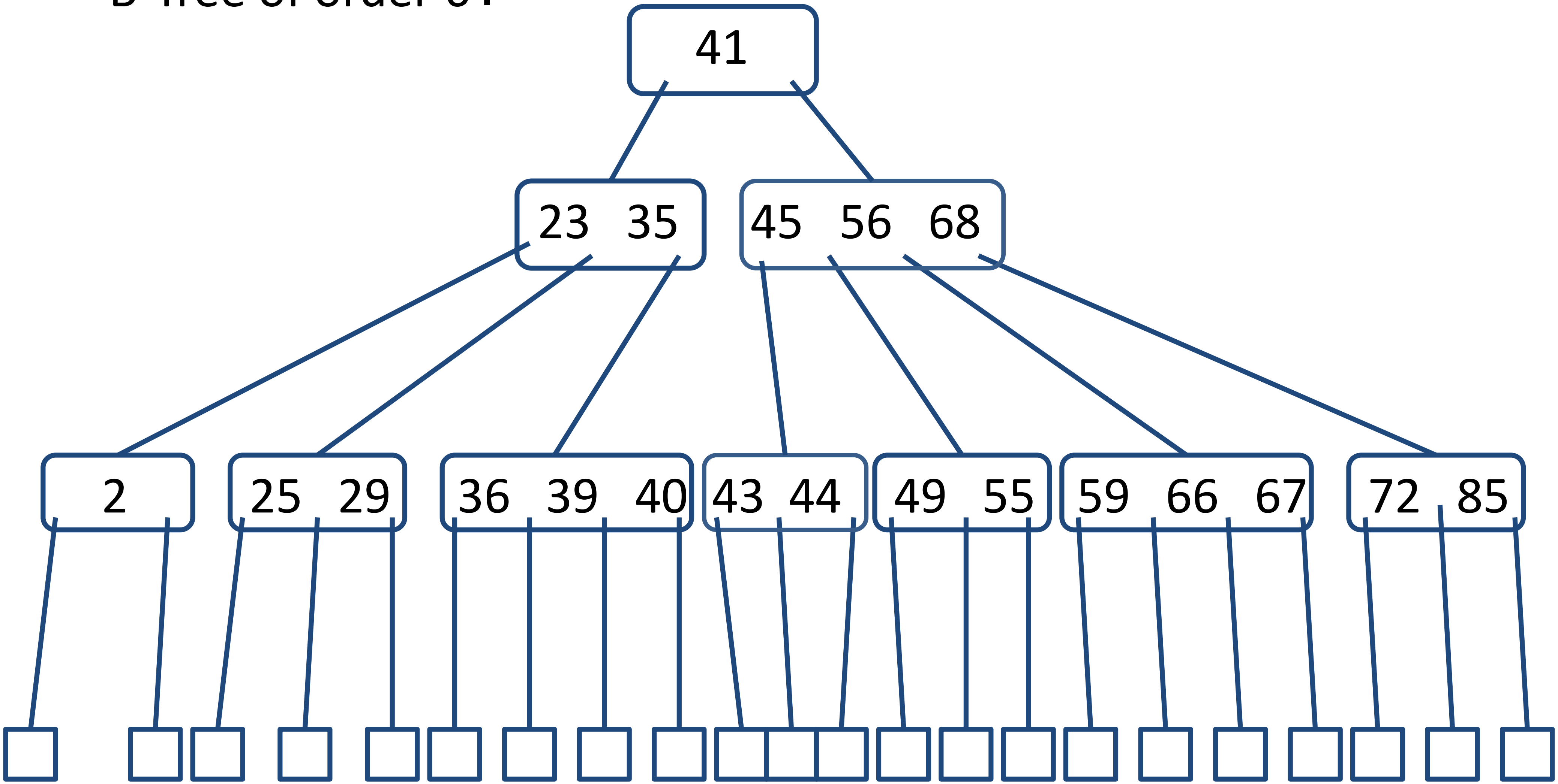
B-Trees

B-Tree of order 6？



B-Trees

B-Tree of order 6?

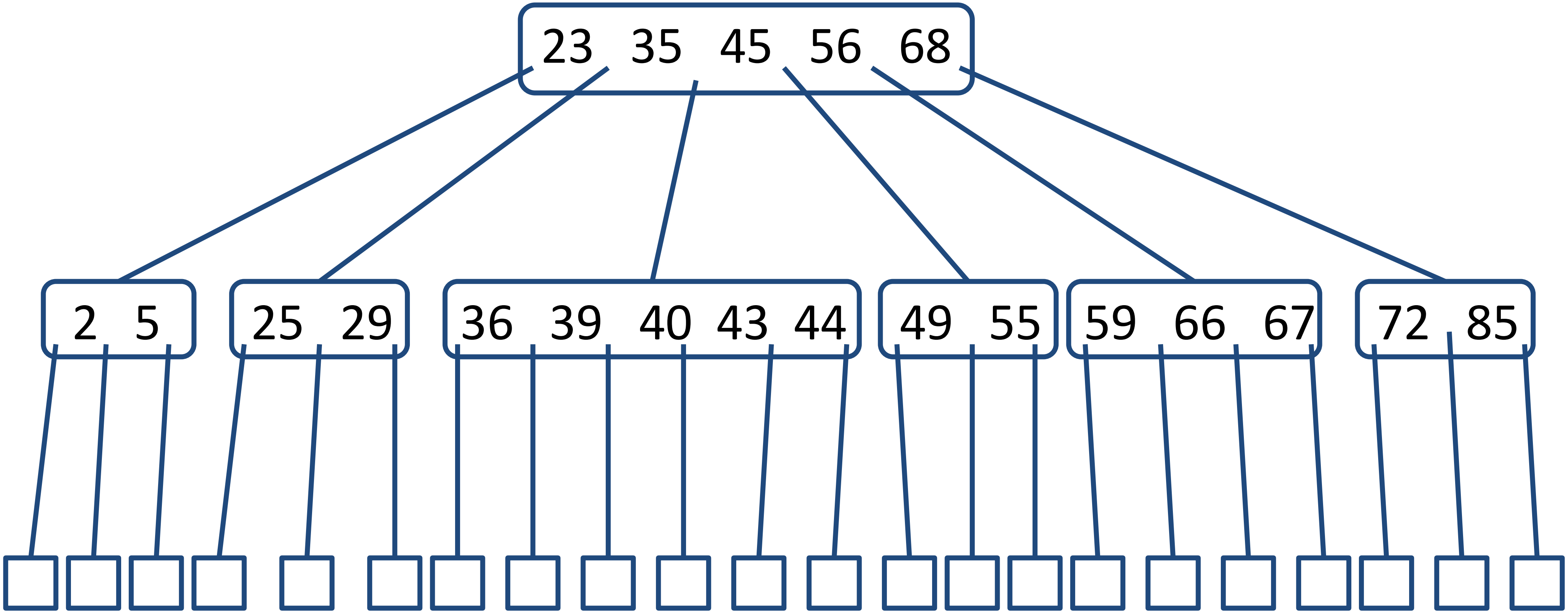


What is the Maximum Height of a B-Tree?

Height is $O(\log_d n)$

B-Trees

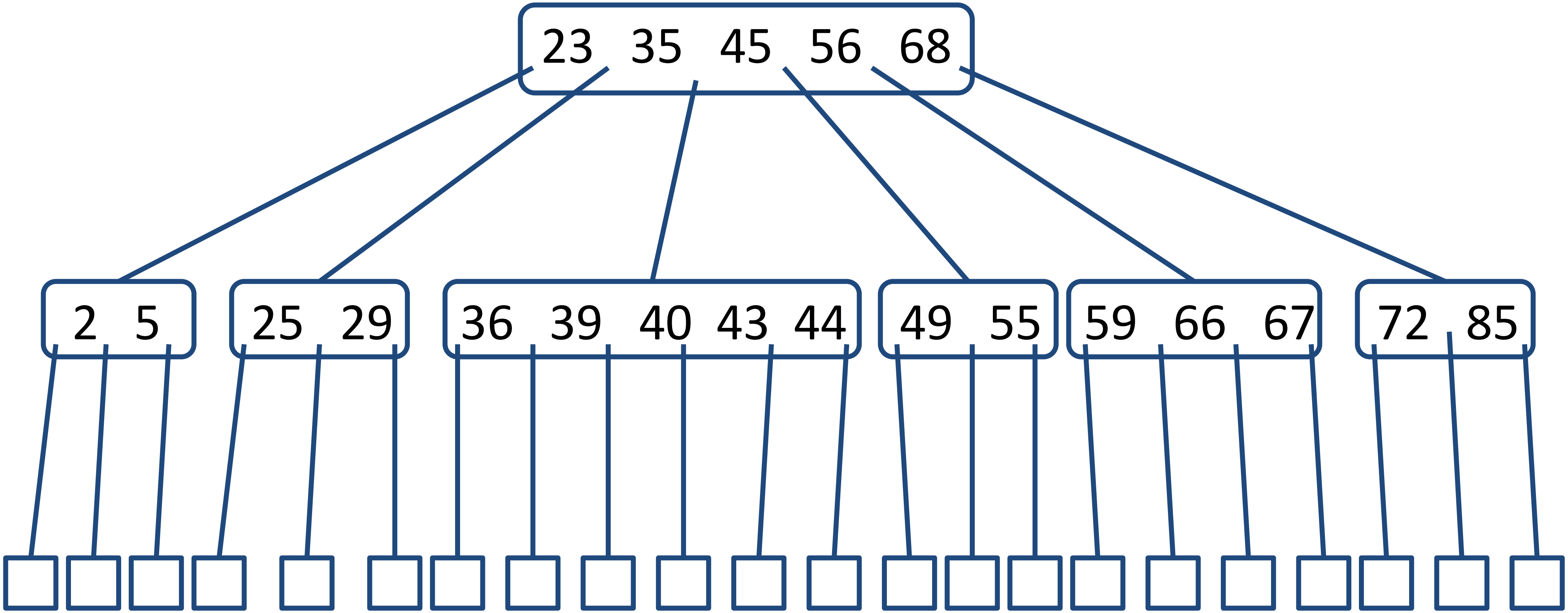
B-Tree of order 6



B-Trees

B-Tree of order 6

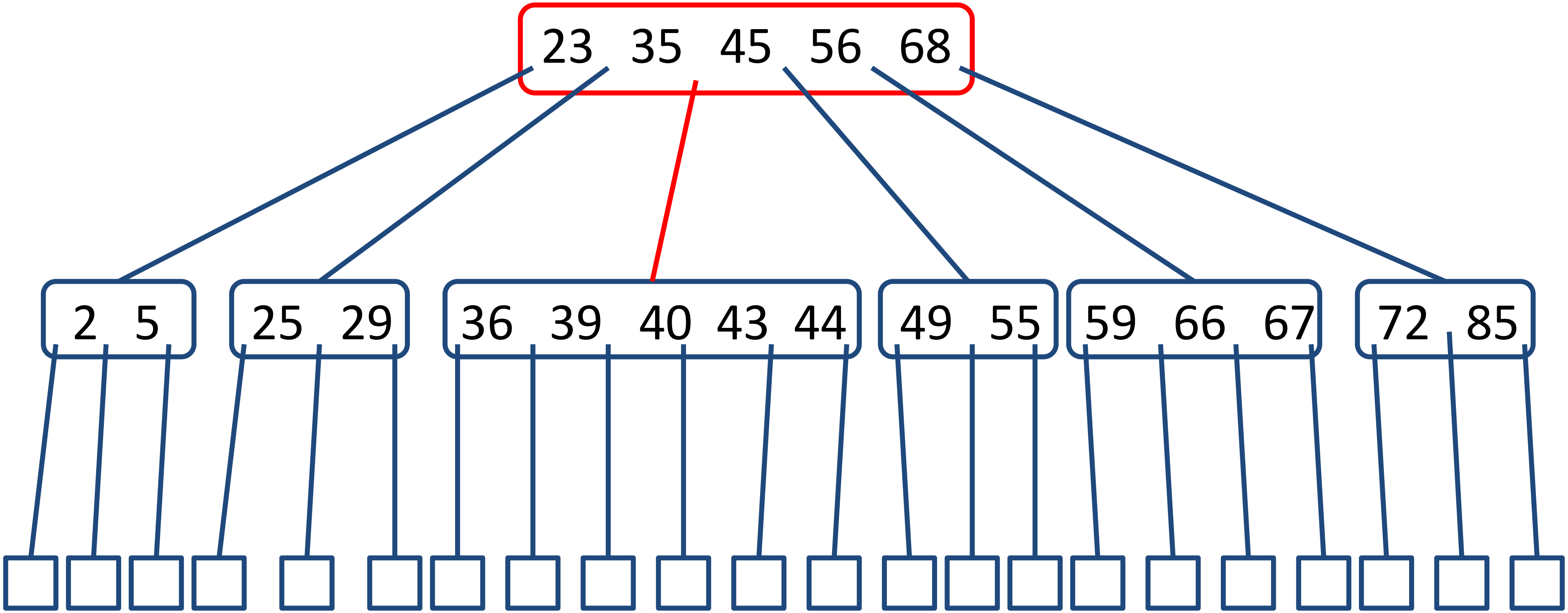
Put 38



B-Trees

B-Tree of order 6

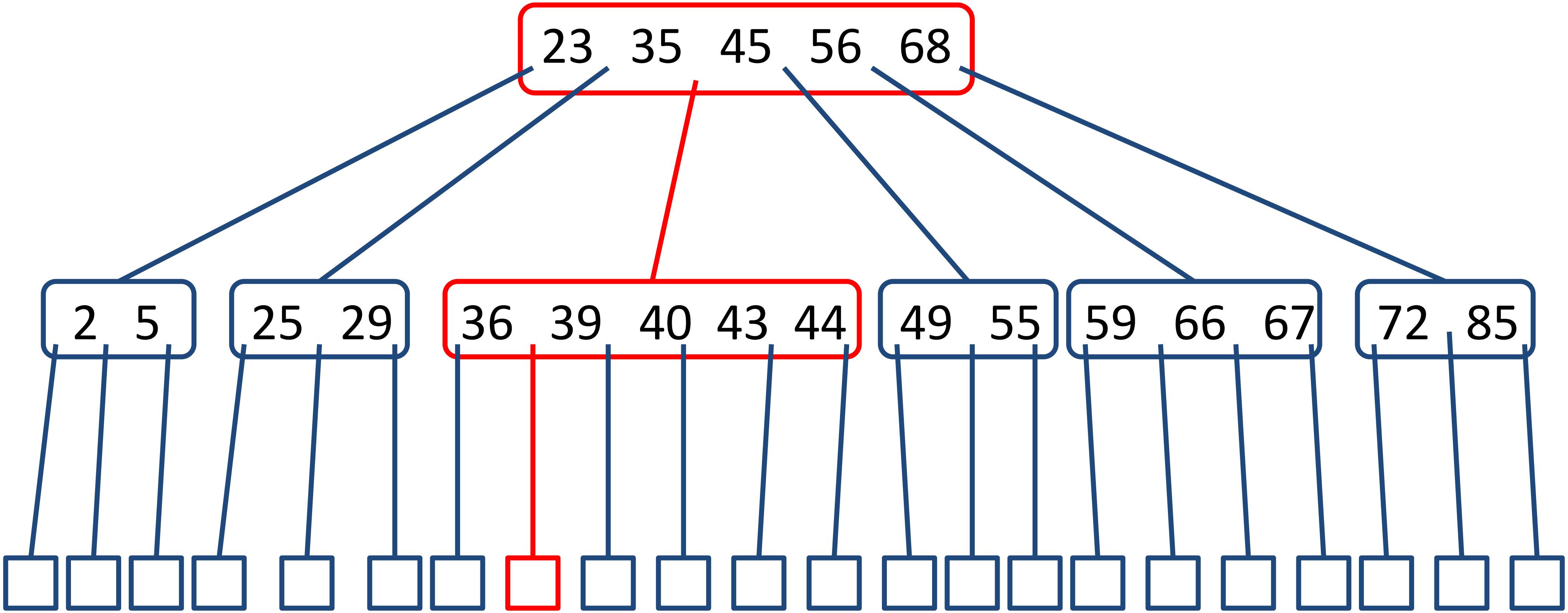
Put 38



B-Trees

B-Tree of order 6

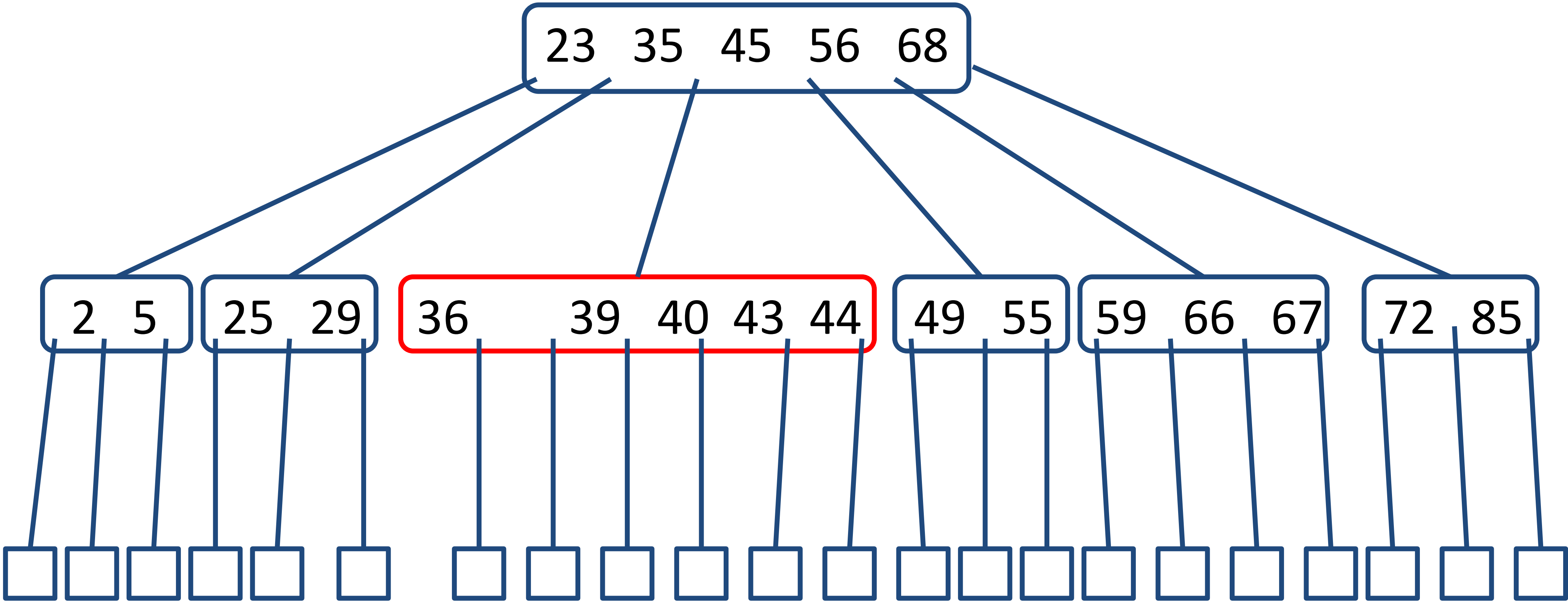
Put 38



B-Trees

B-Tree of order 6

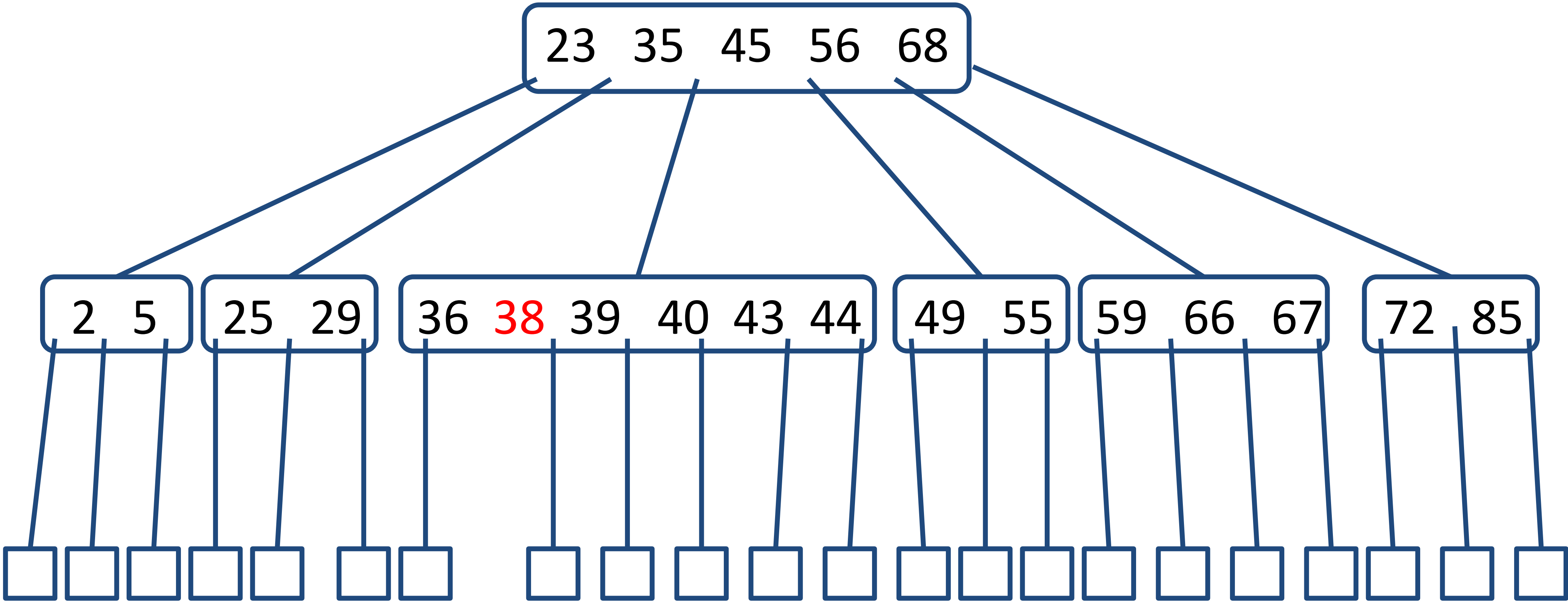
Put 38



B-Trees

B-Tree of order 6

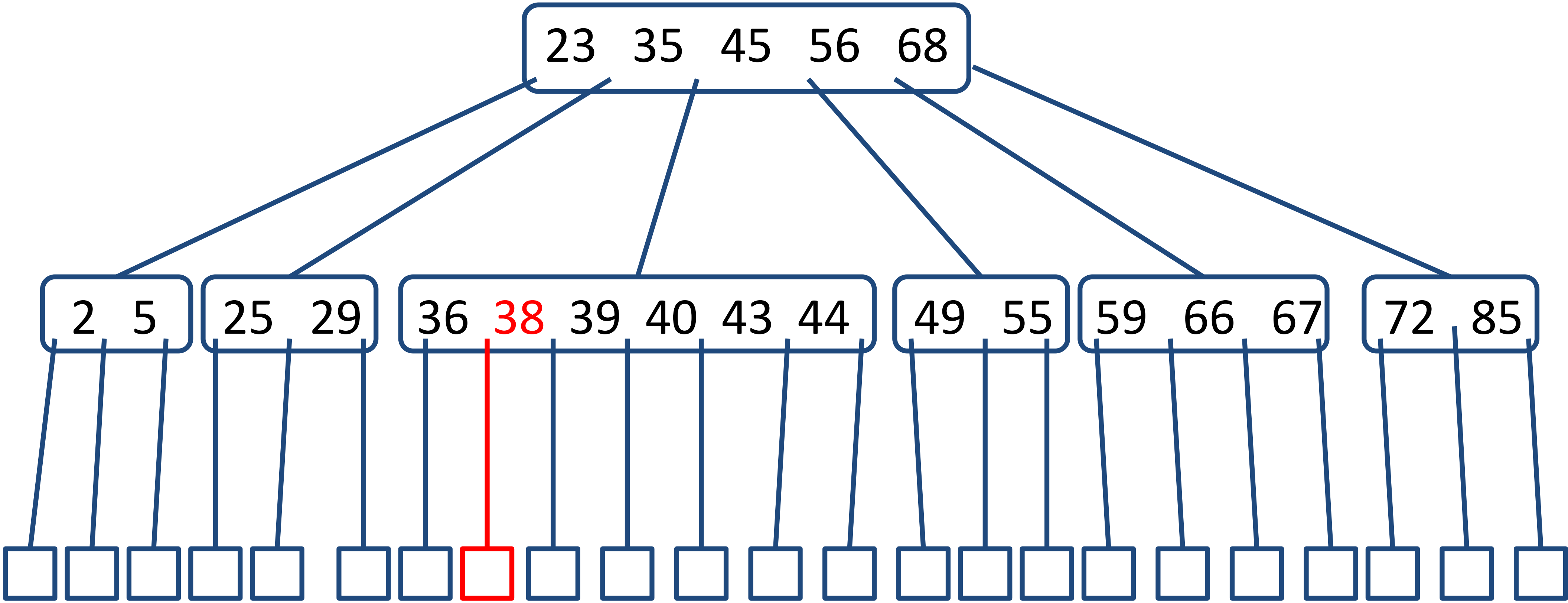
Put 38



B-Trees

B-Tree of order 6

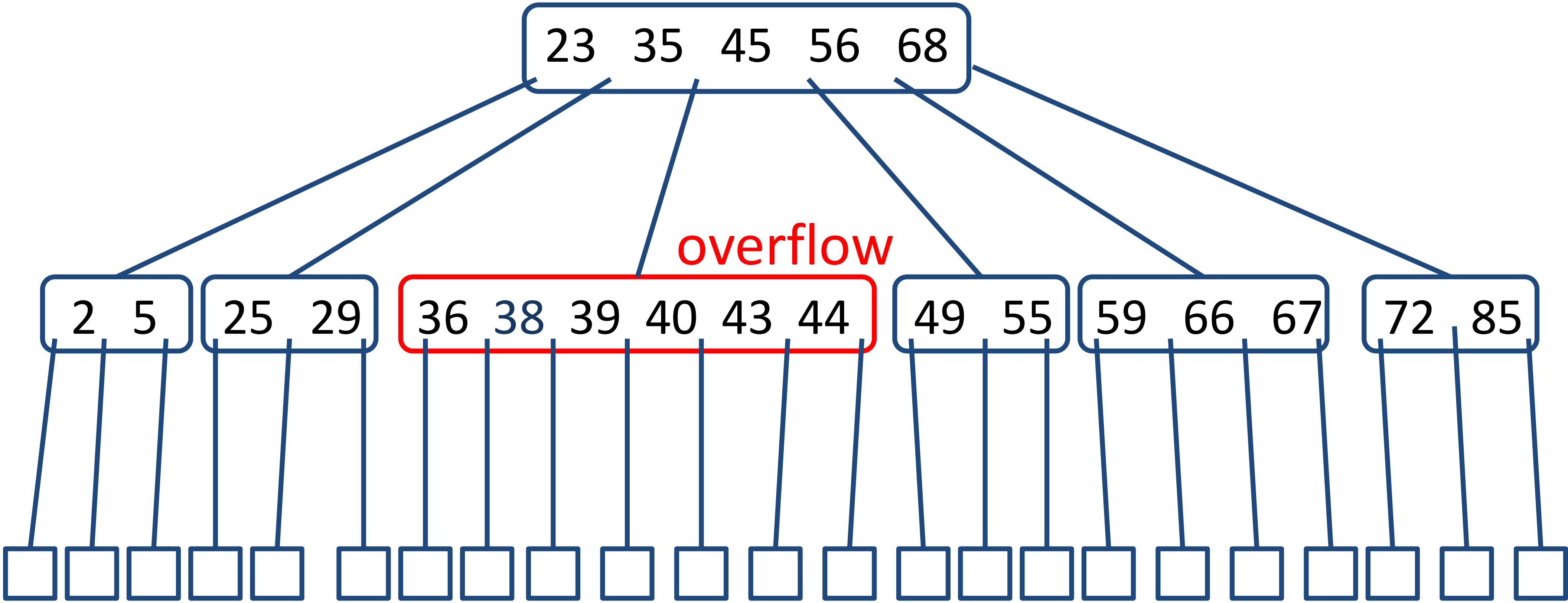
Put 38



B-Trees

B-Tree of order 6

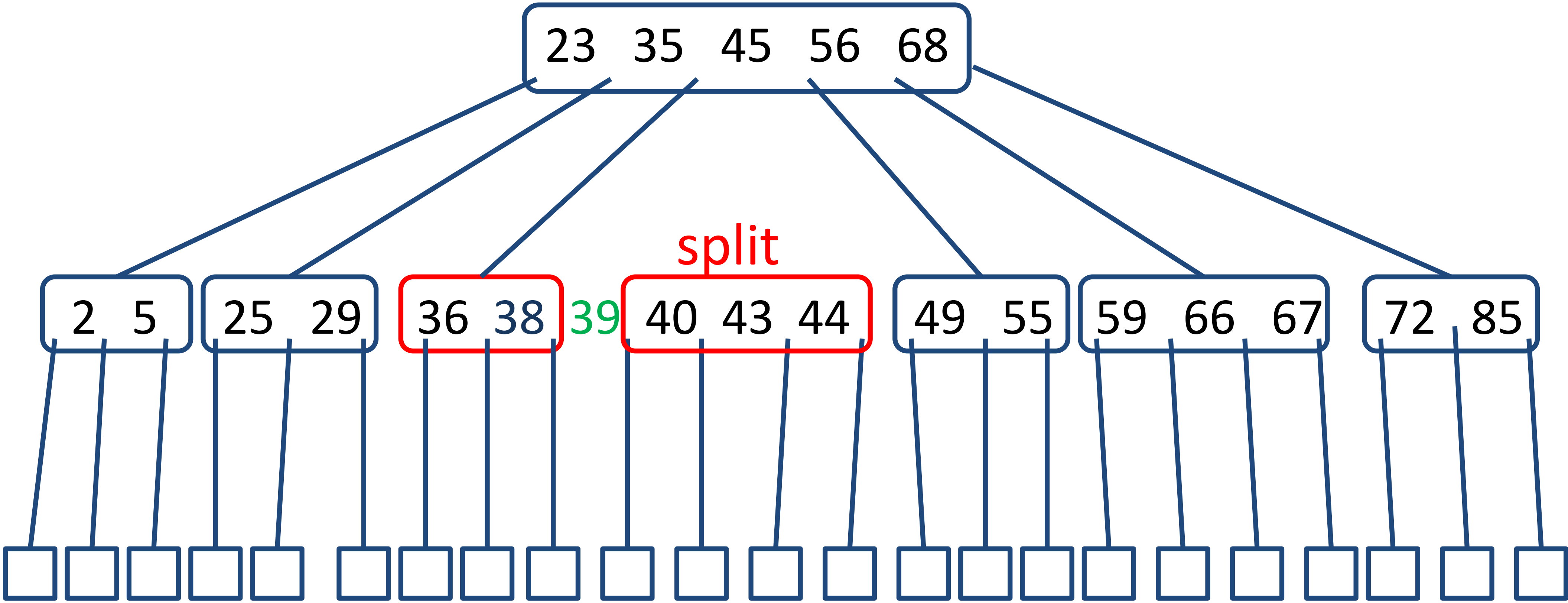
Put 38



B-Trees

B-Tree of order 6

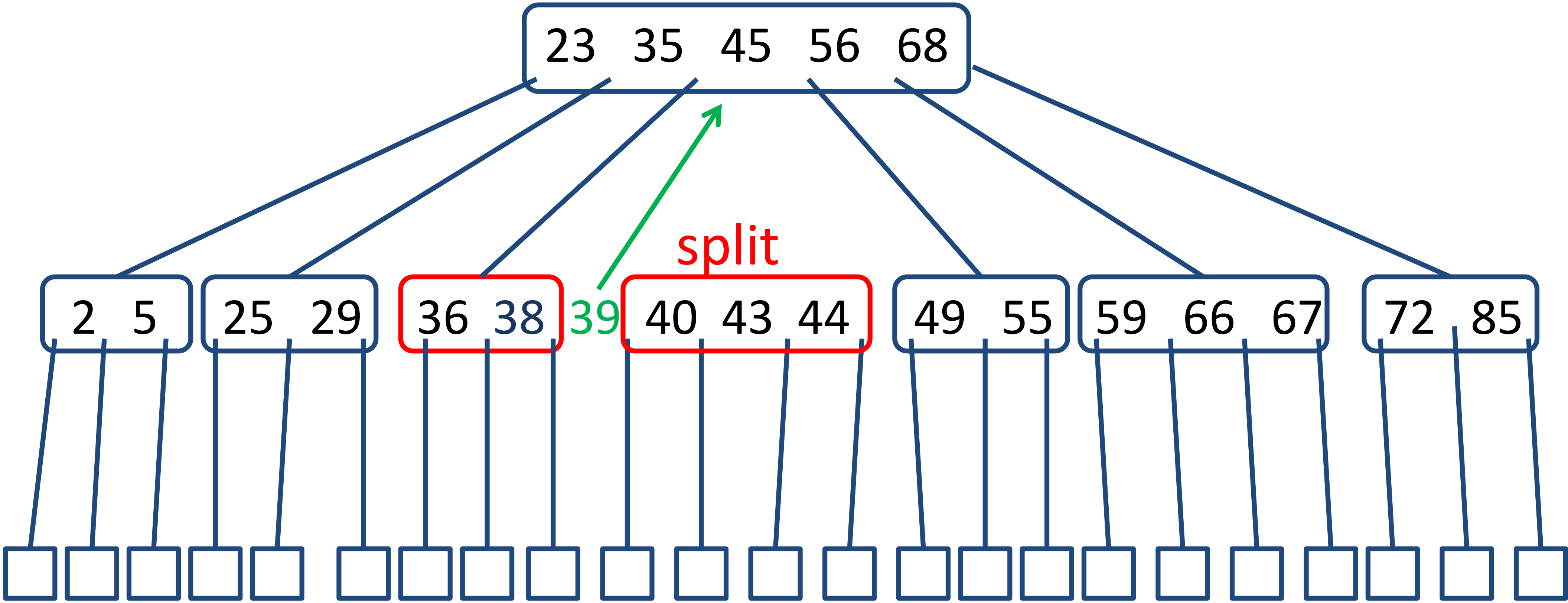
Put 38



B-Trees

B-Tree of order 6

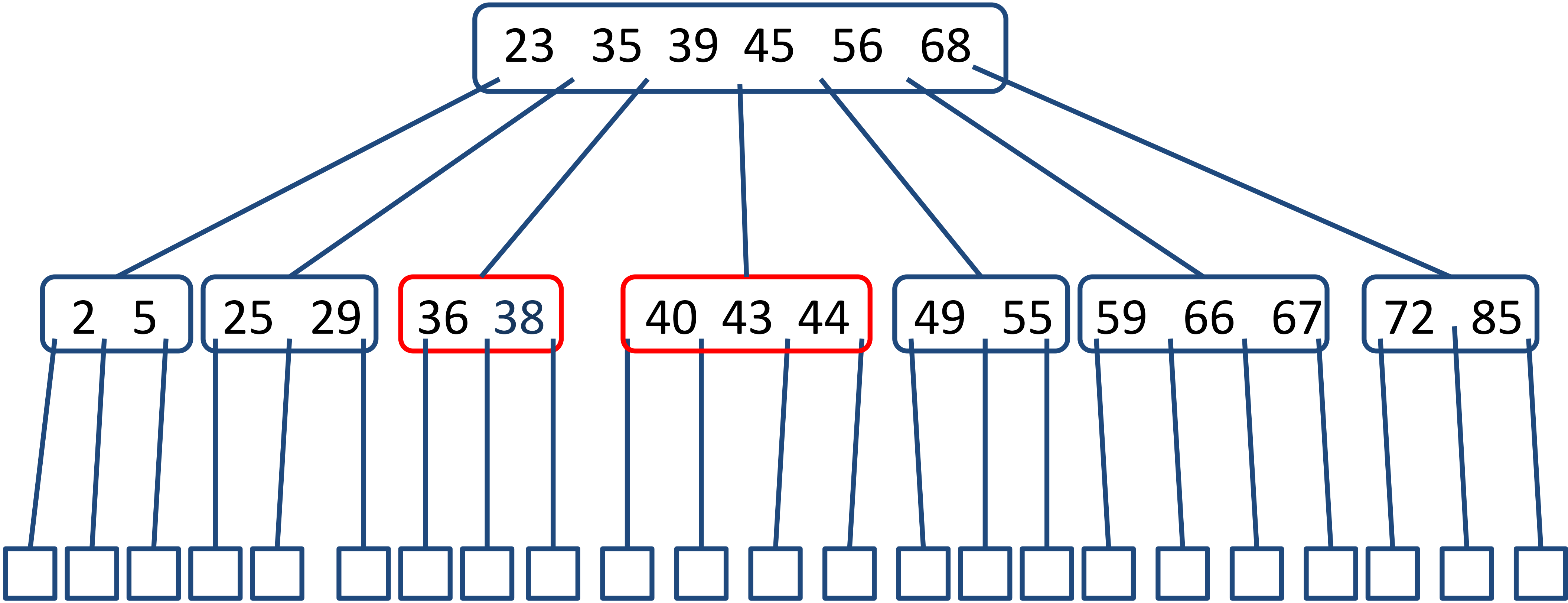
Put 38



B-Trees

B-Tree of order 6

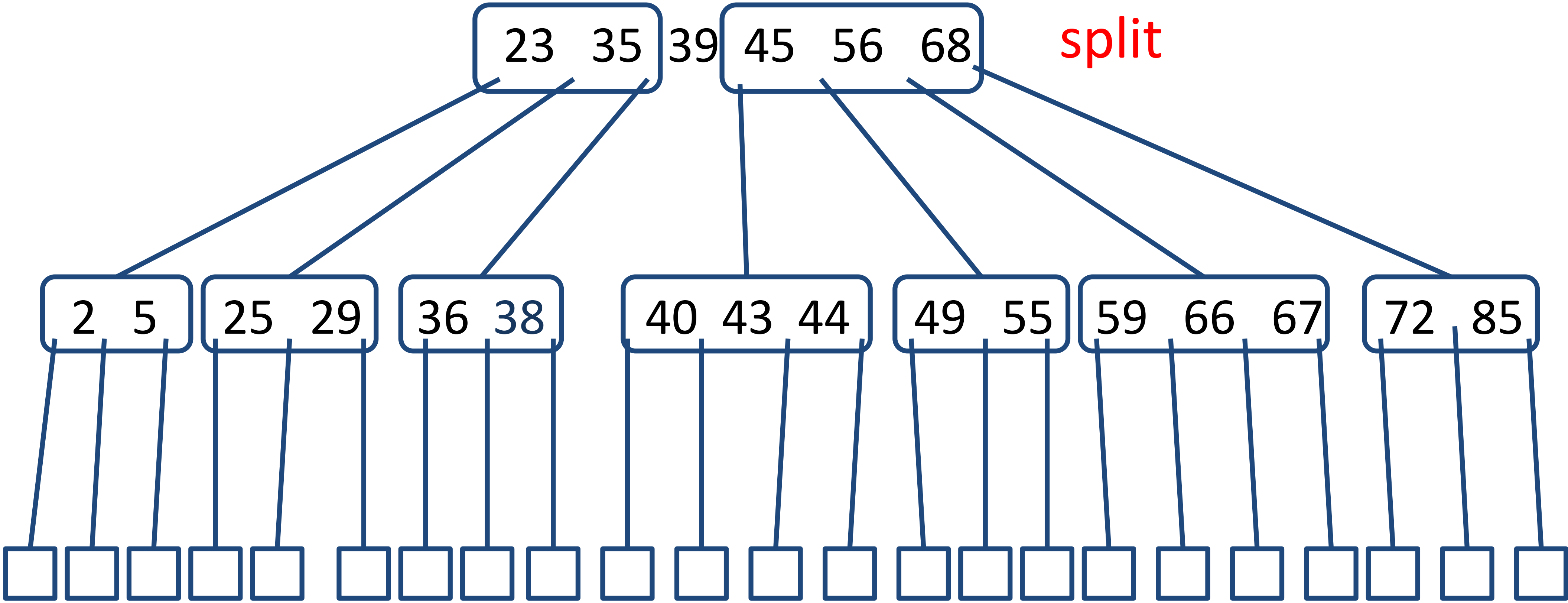
Put 38



B-Trees

B-Tree of order 6

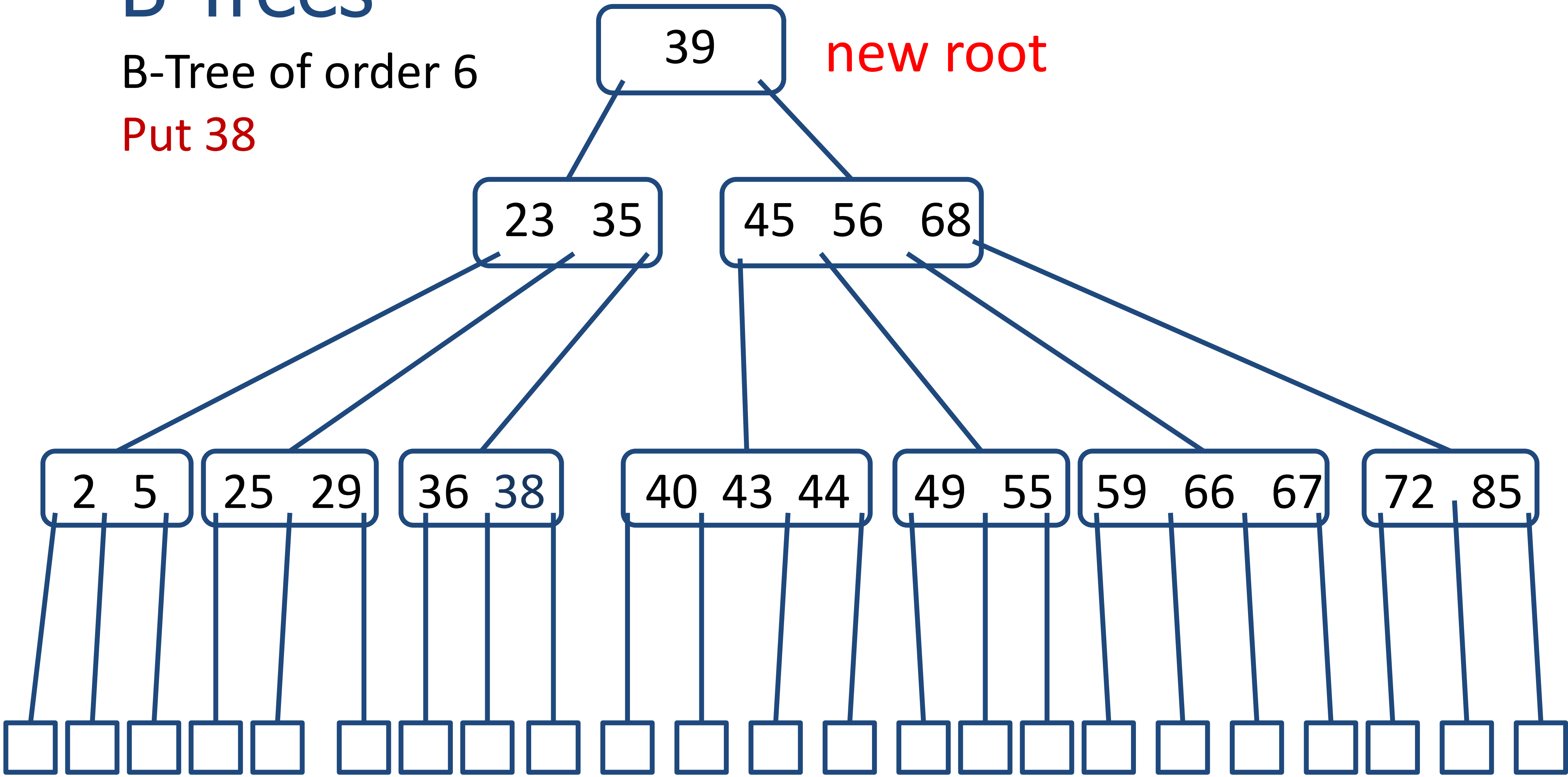
Put 38



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

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

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

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

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

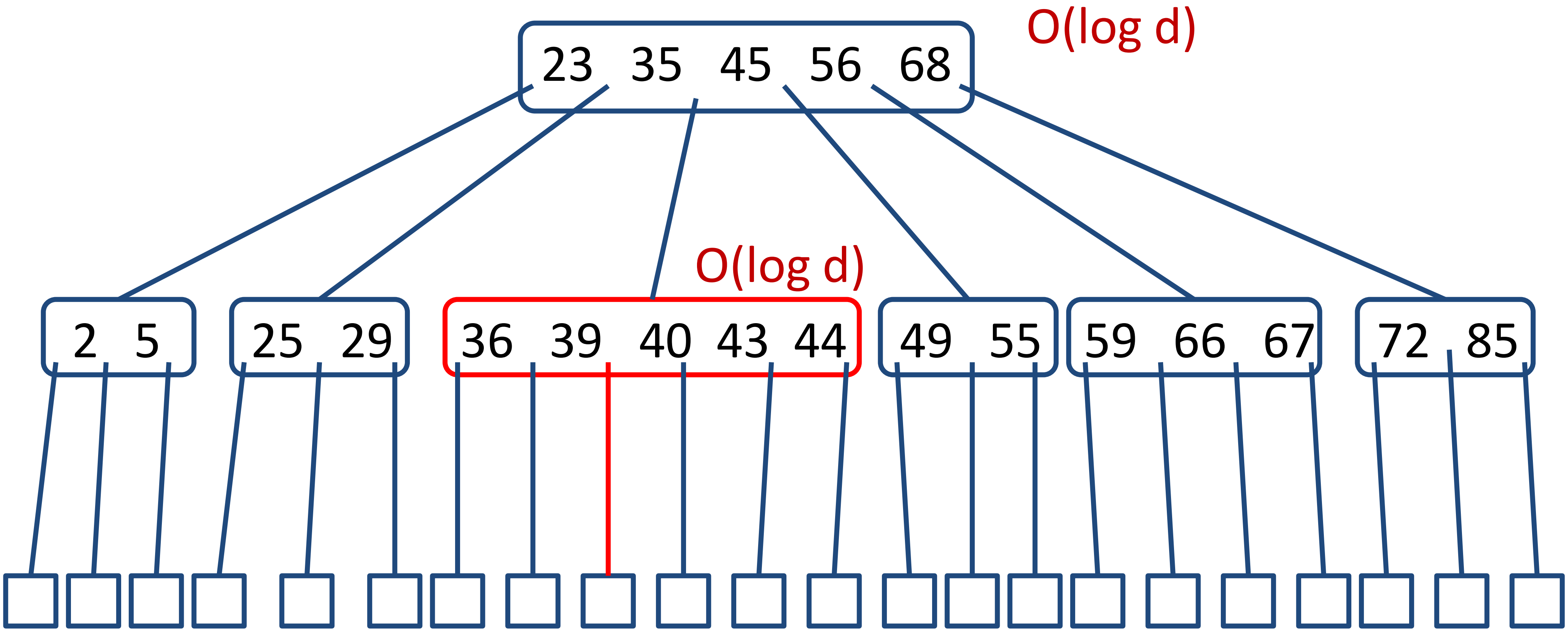
Split $\boldsymbol{v}$ around the middle key $\boldsymbol{k'}$, move $\boldsymbol{k'}$ to parent, and update parent's children

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

}

B-Trees

B-Tree of order 6



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

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

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

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

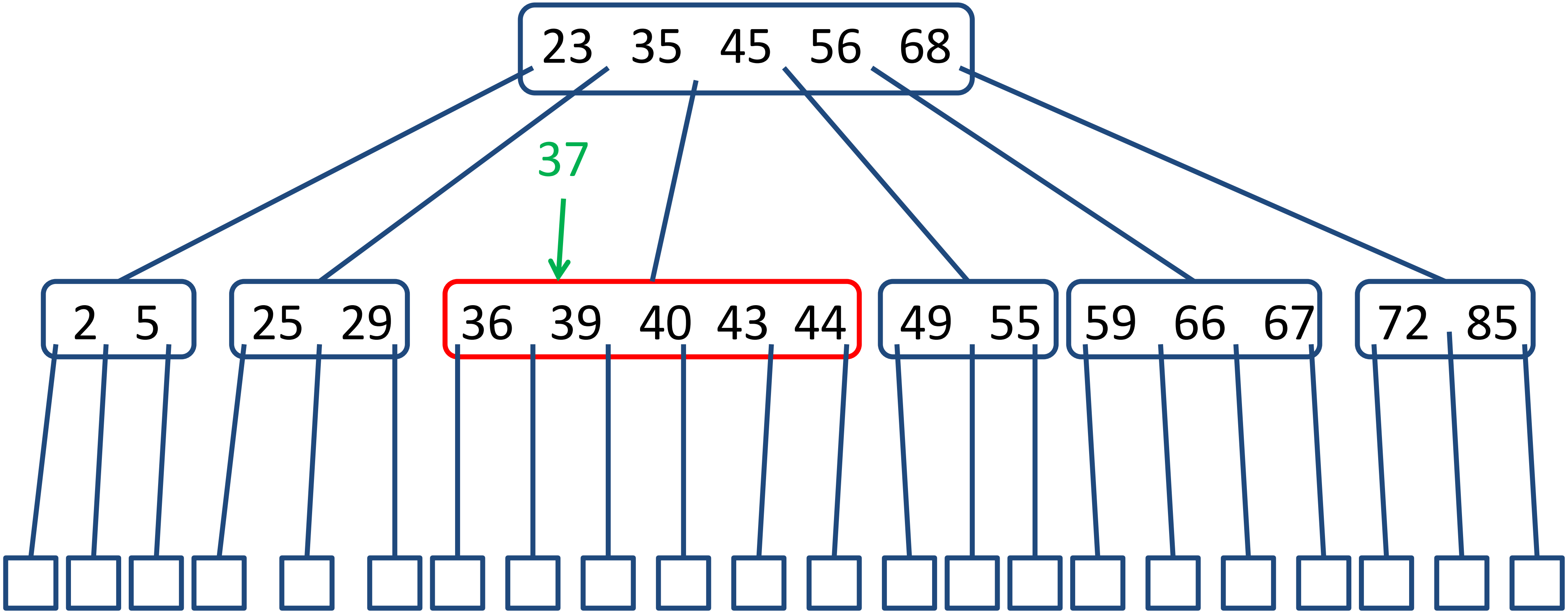
$\boldsymbol{v} \leftarrow$ parent of $\boldsymbol{v}$

}

B-Trees

B-Tree of order 6

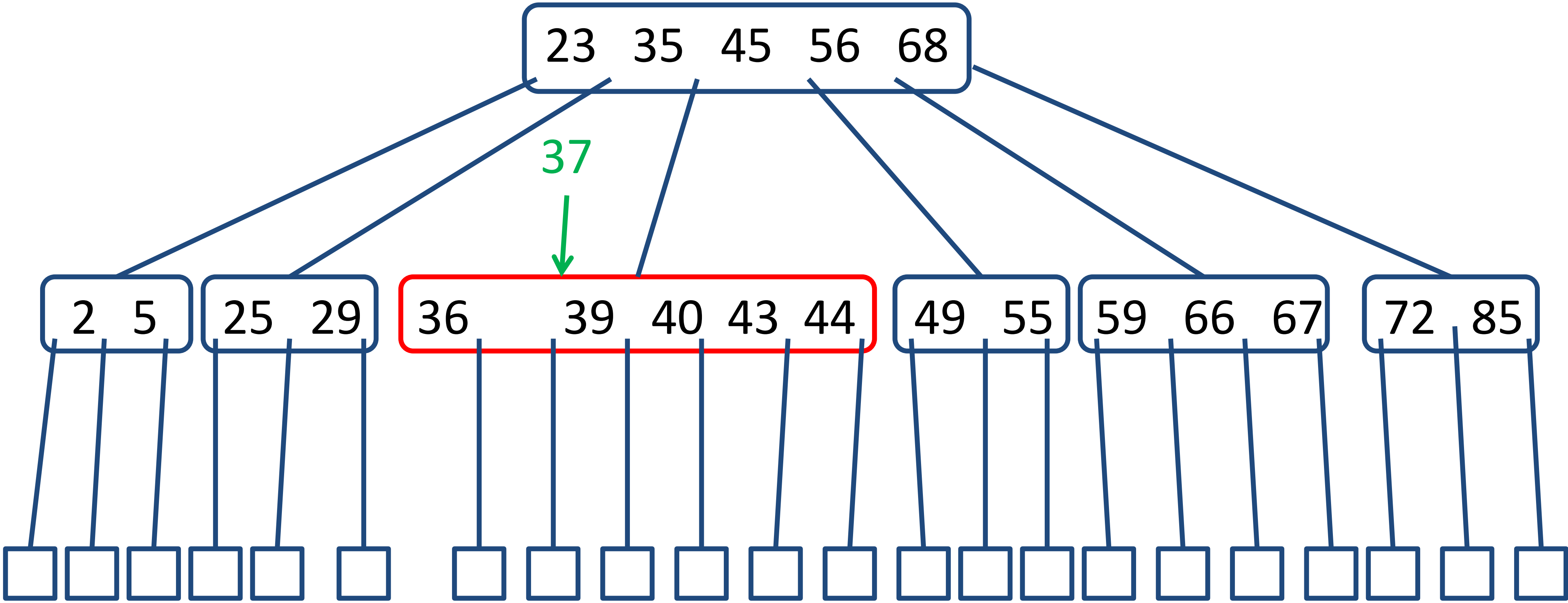
Put 38



B-Trees

B-Tree of order 6

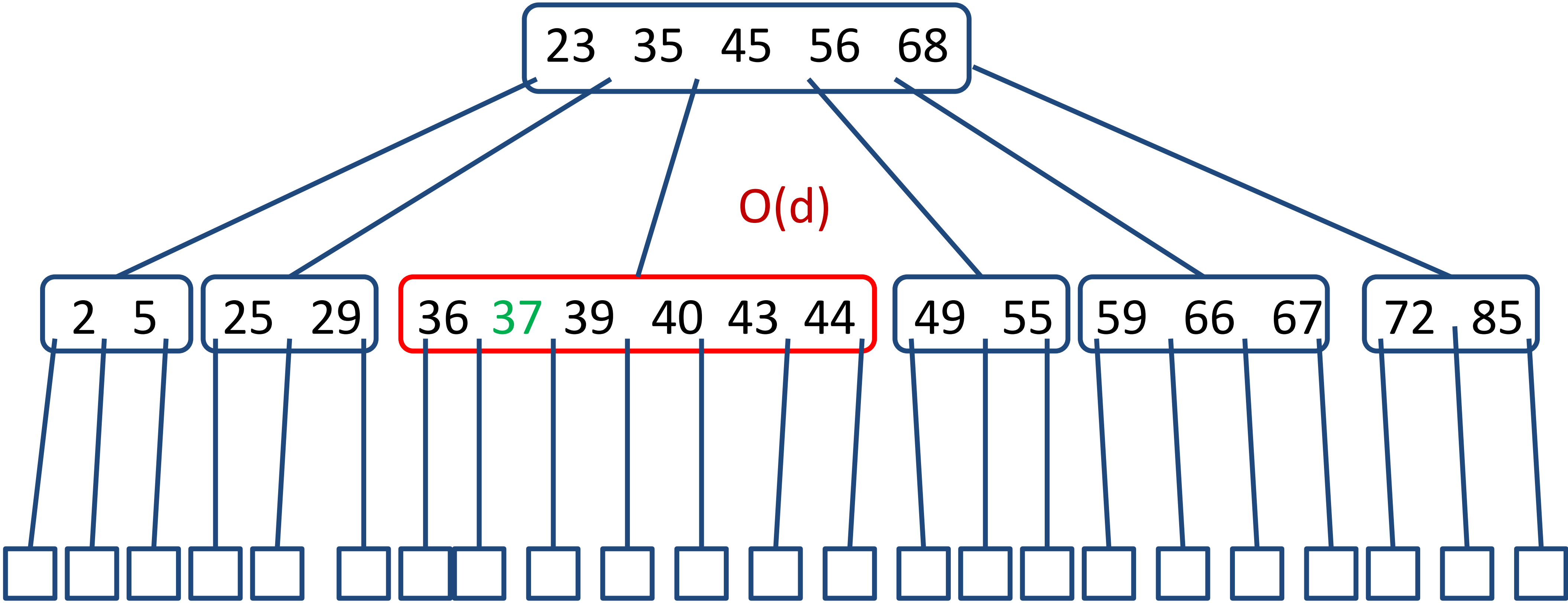
Put 38



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

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 ***overflows*** **do {**

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

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

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 {**

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

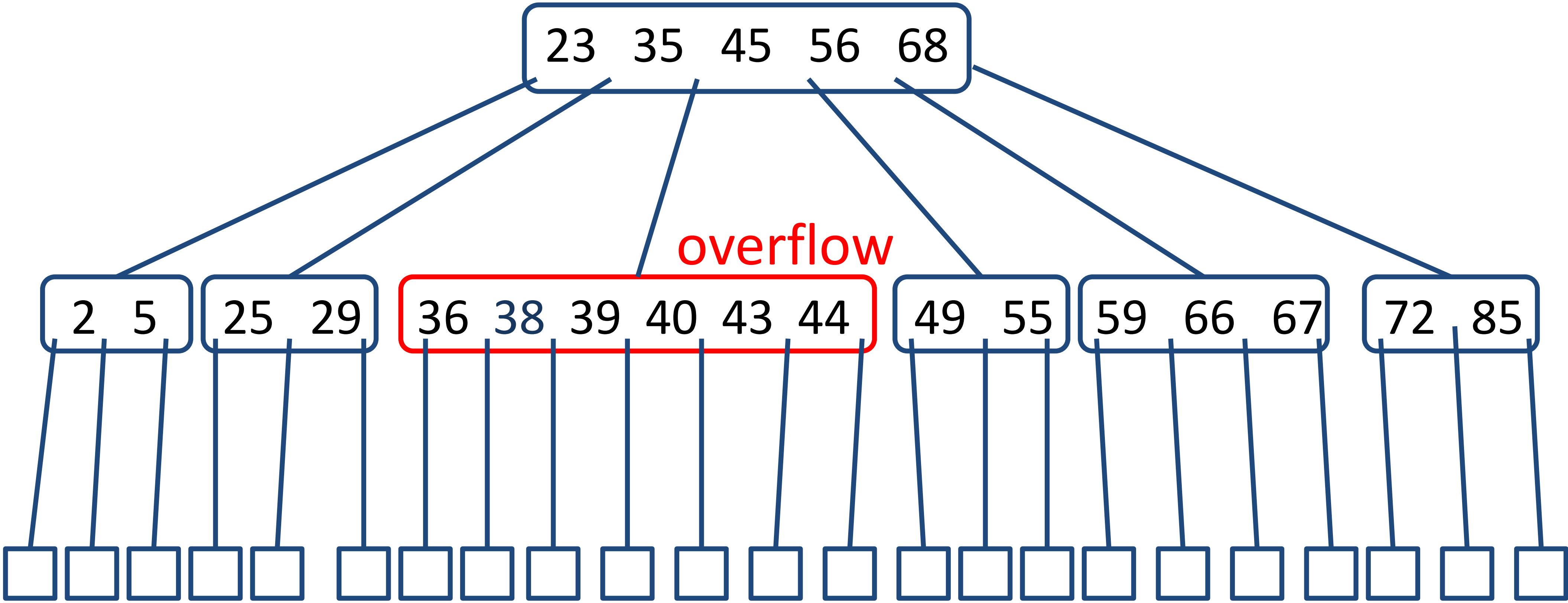
$\boldsymbol{v} \leftarrow$ parent of $\boldsymbol{v}$

}

B-Trees

B-Tree of order 6

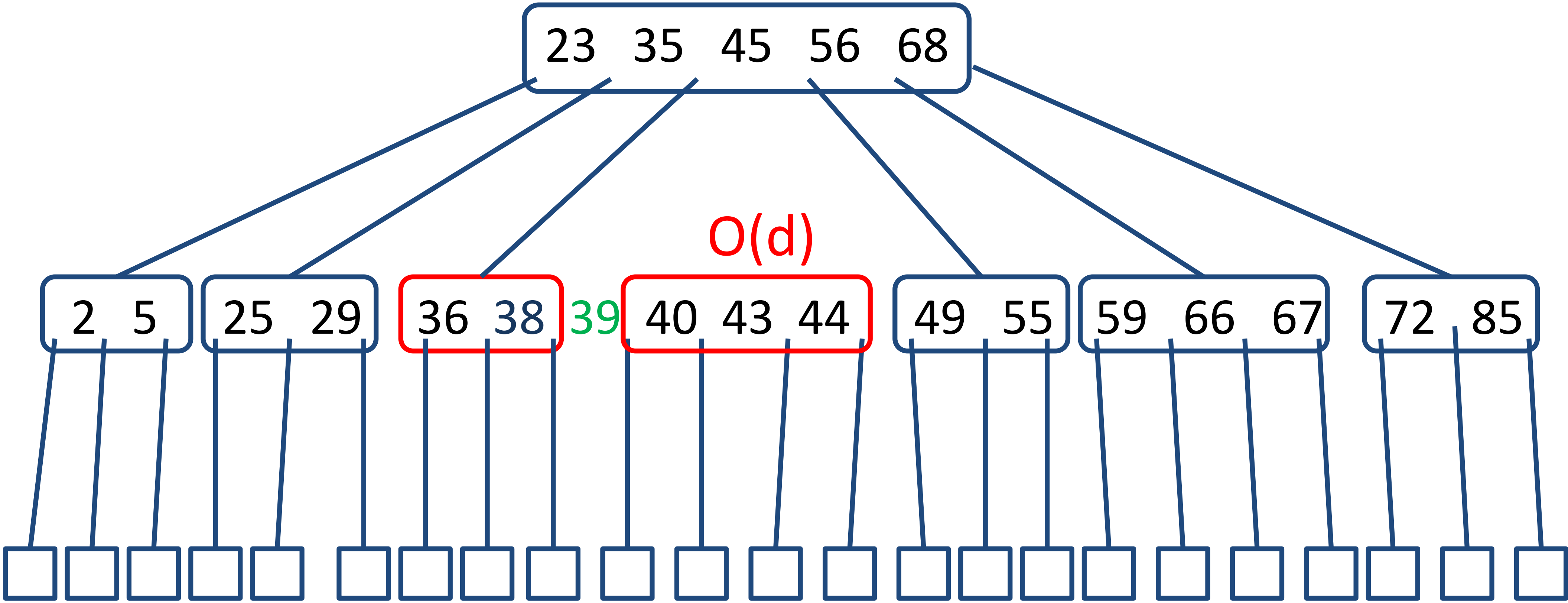
Put 38



B-Trees

B-Tree of order 6

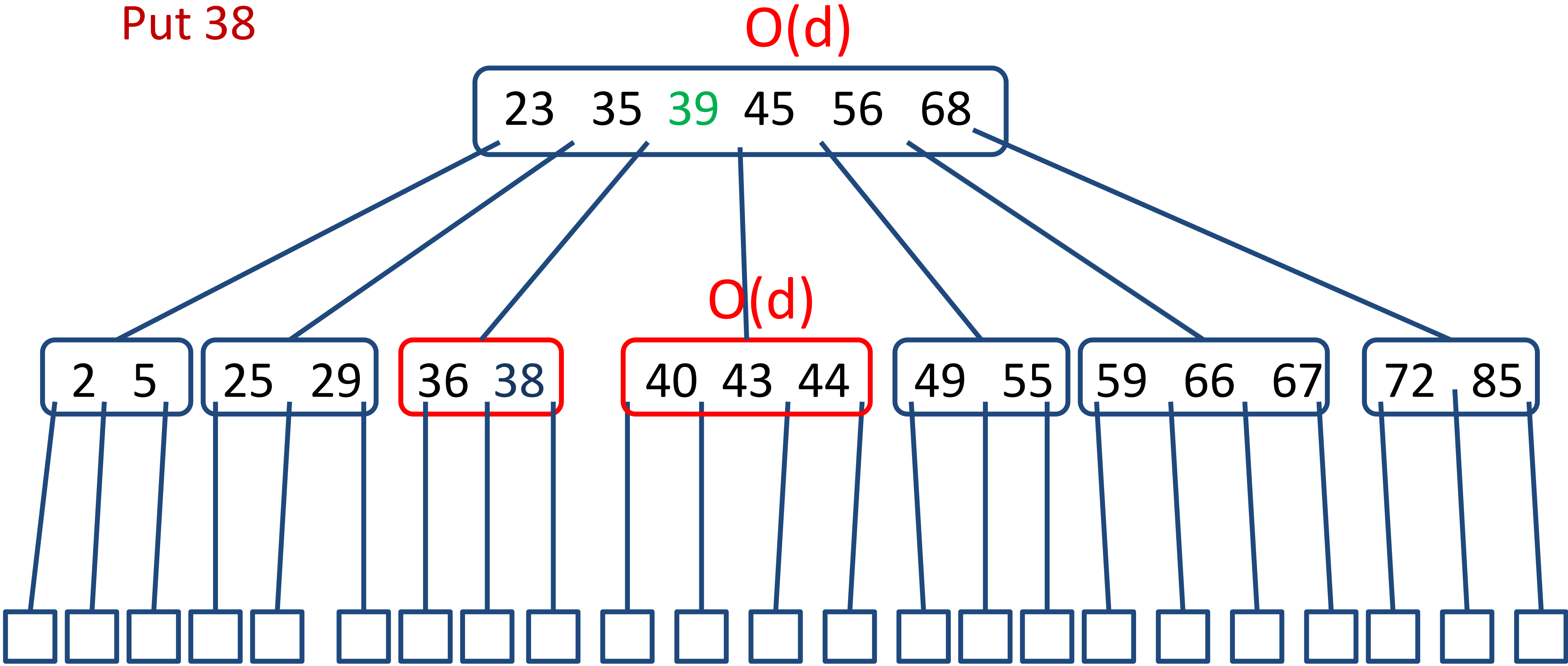
Put 38



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*** $\}$ O(log d × $\log_d$ 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***

}

O(d)

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

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

O(d)

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

O(d)

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

O(d log$_d$ n)

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***

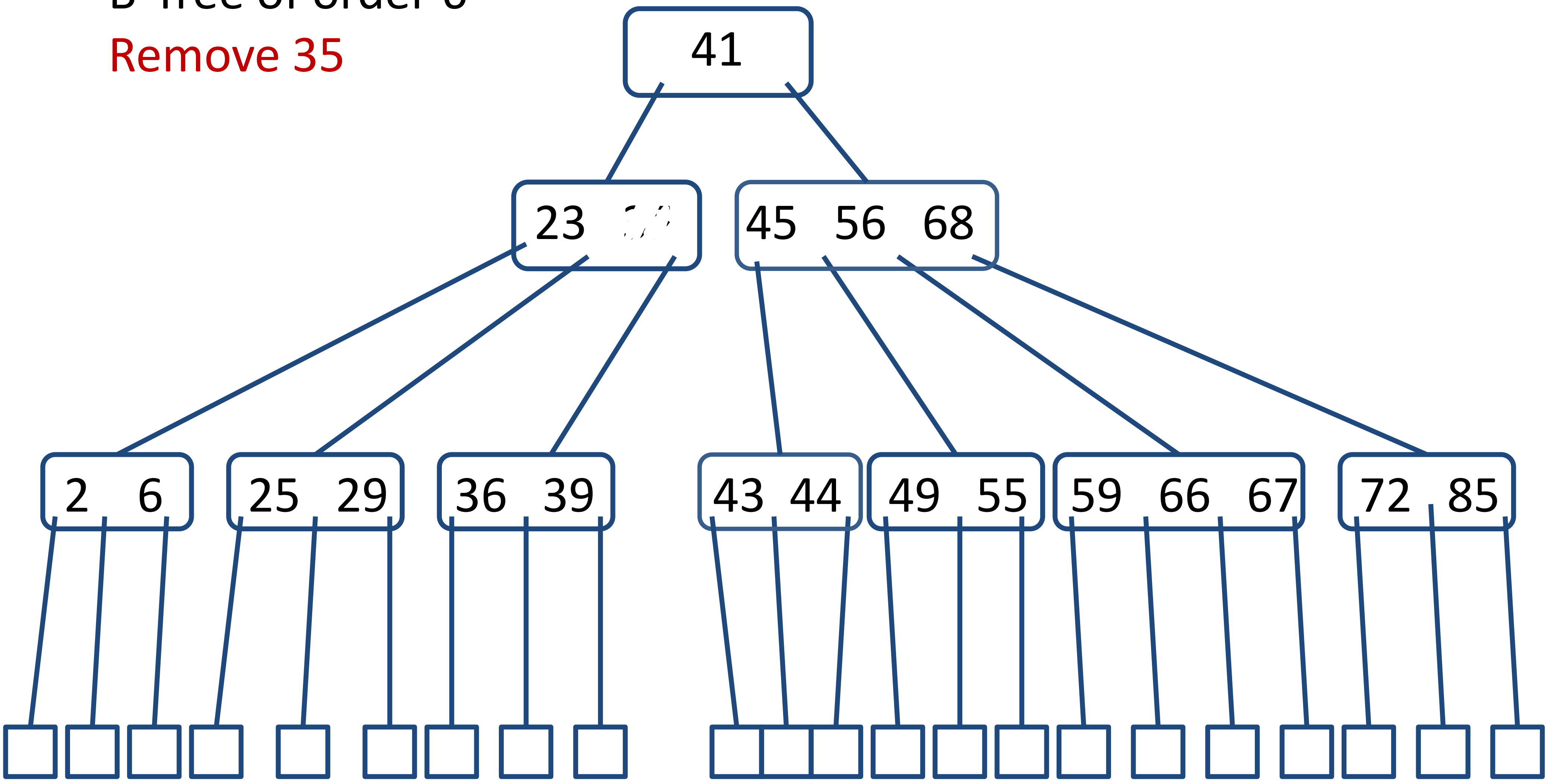
}

Time complexity of put is O(d log$_d$ n)

B-Trees

B-Tree of order 6

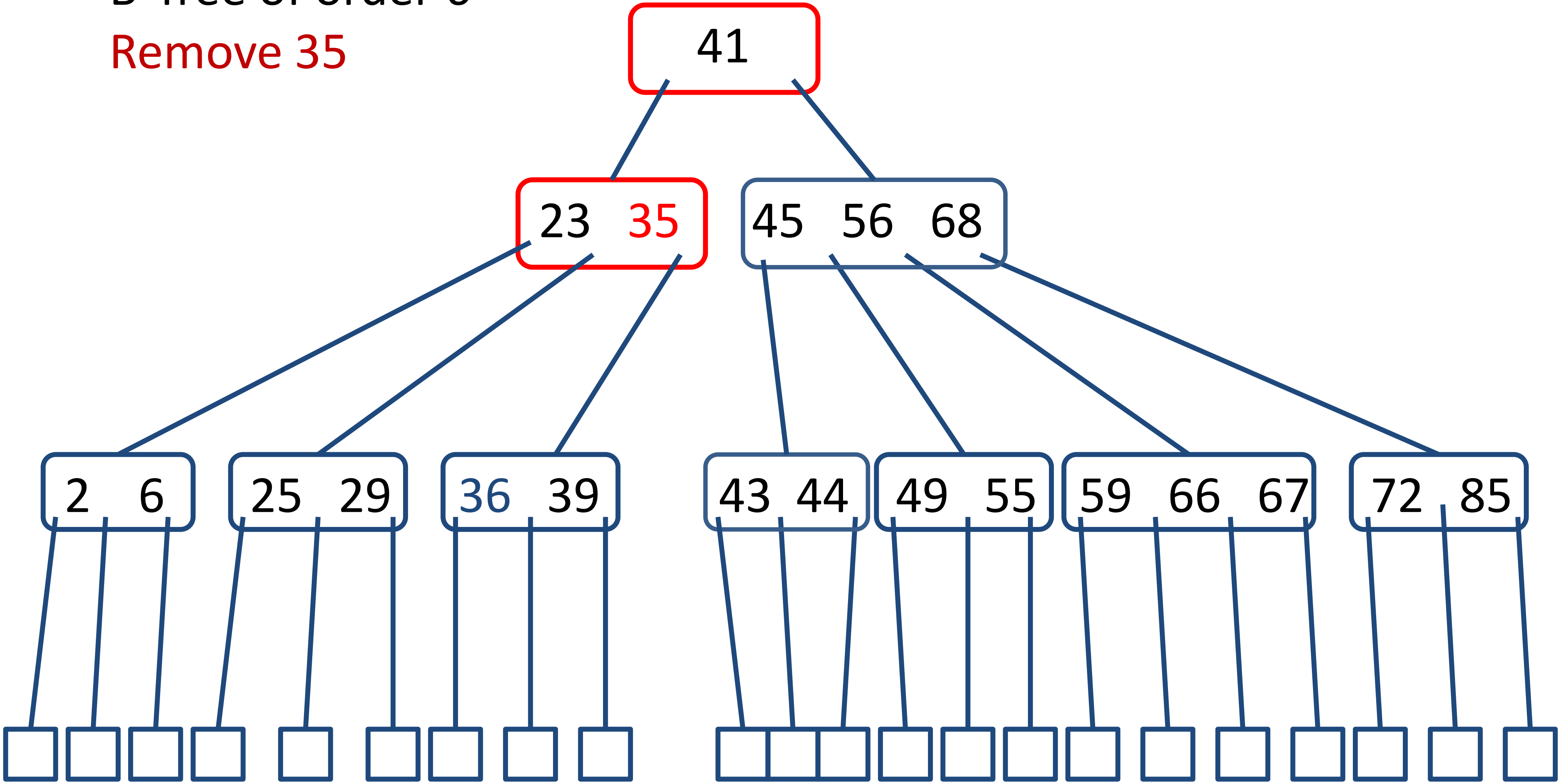
Remove 35



B-Trees

B-Tree of order 6

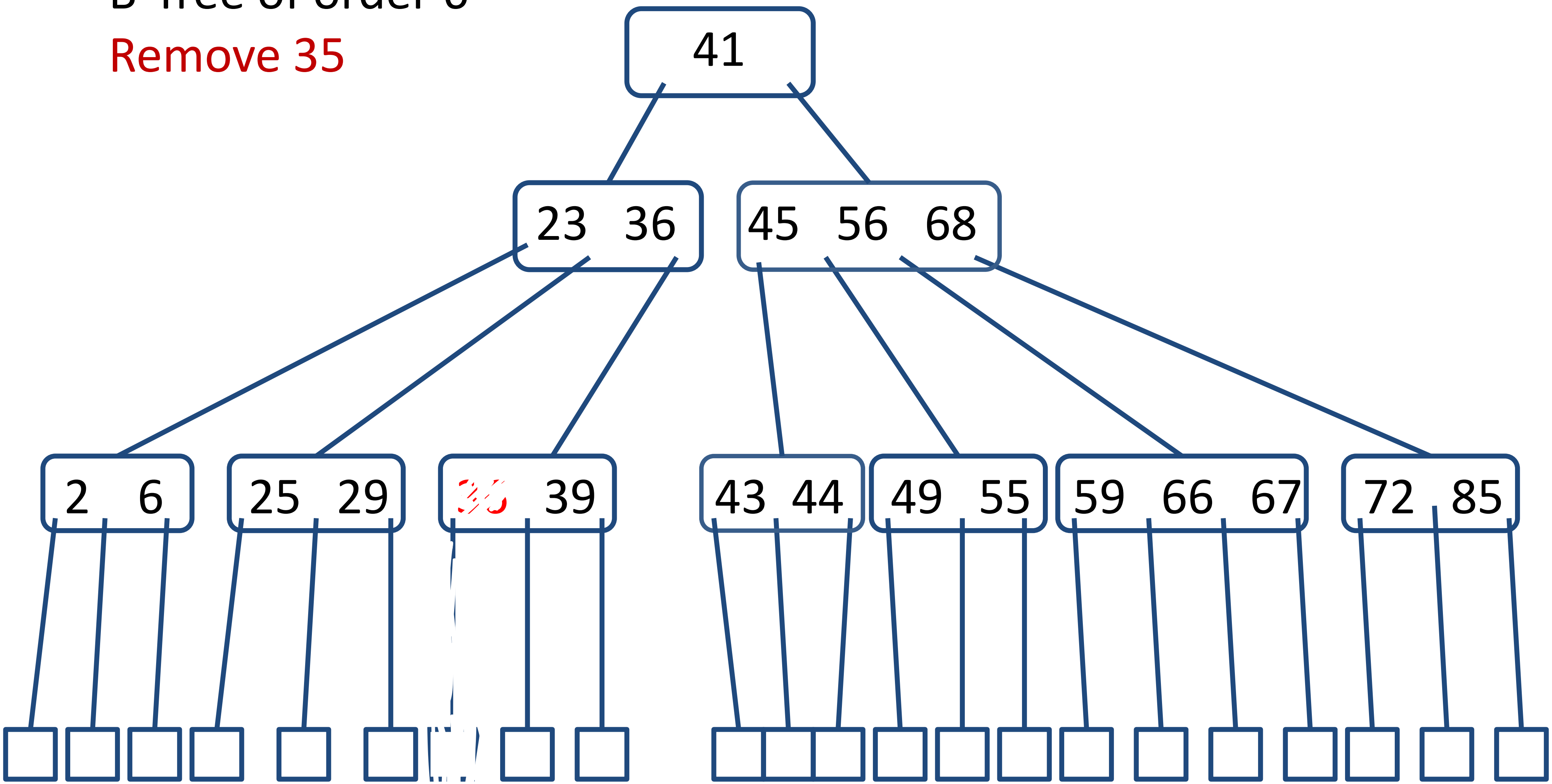
Remove 35



B-Trees

B-Tree of order 6

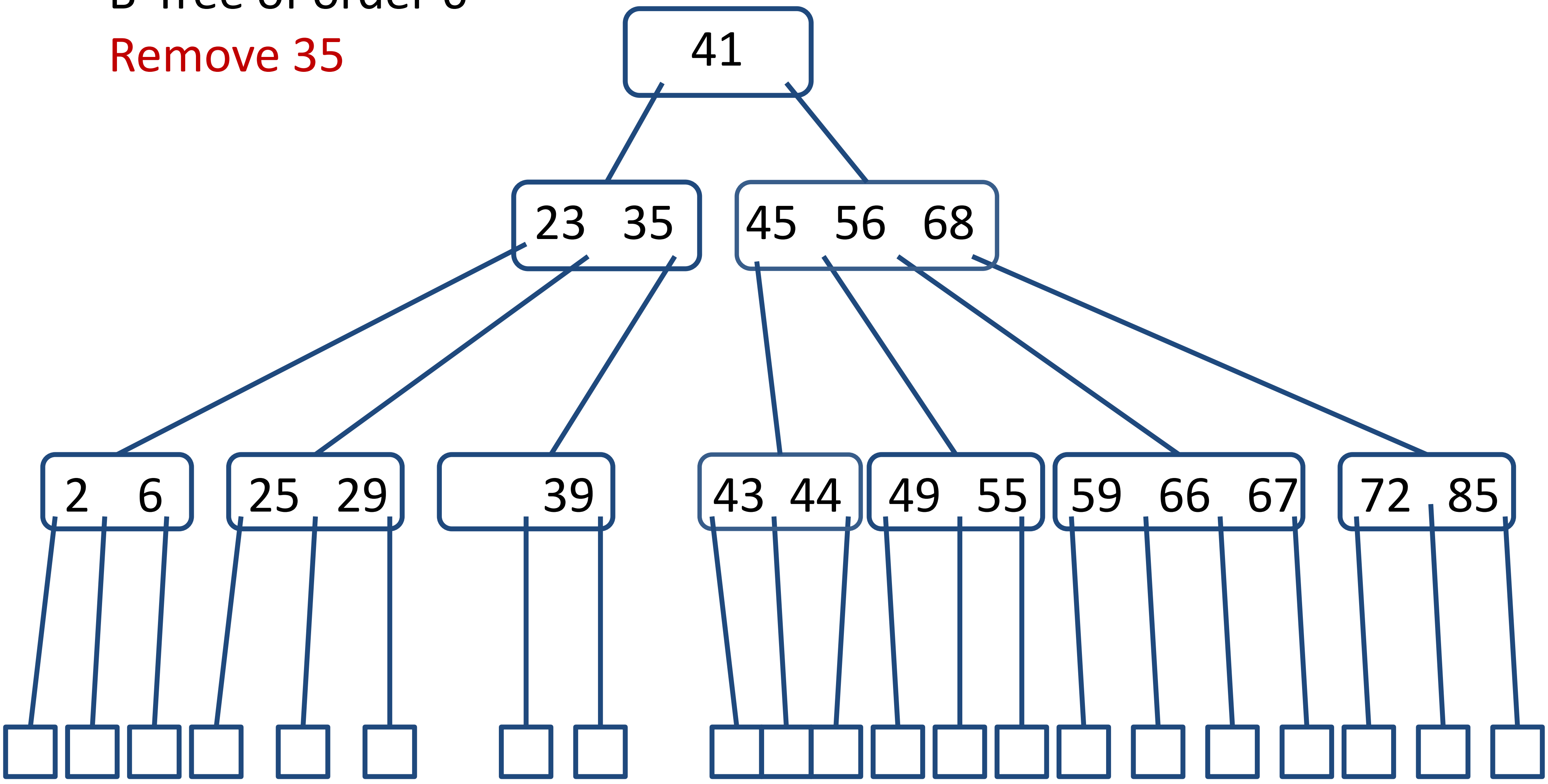
Remove 35



B-Trees

B-Tree of order 6

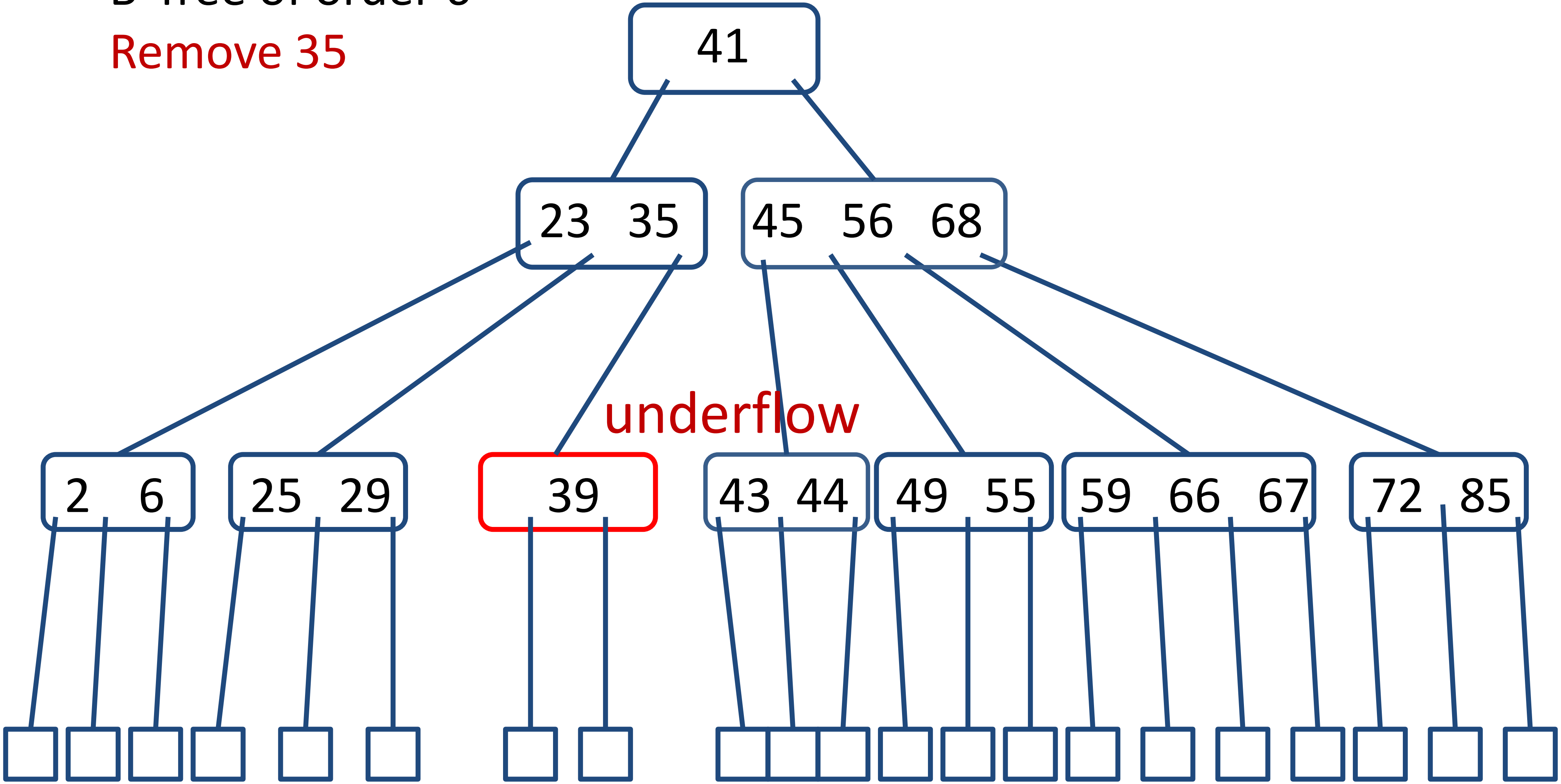
Remove 35



B-Trees

B-Tree of order 6

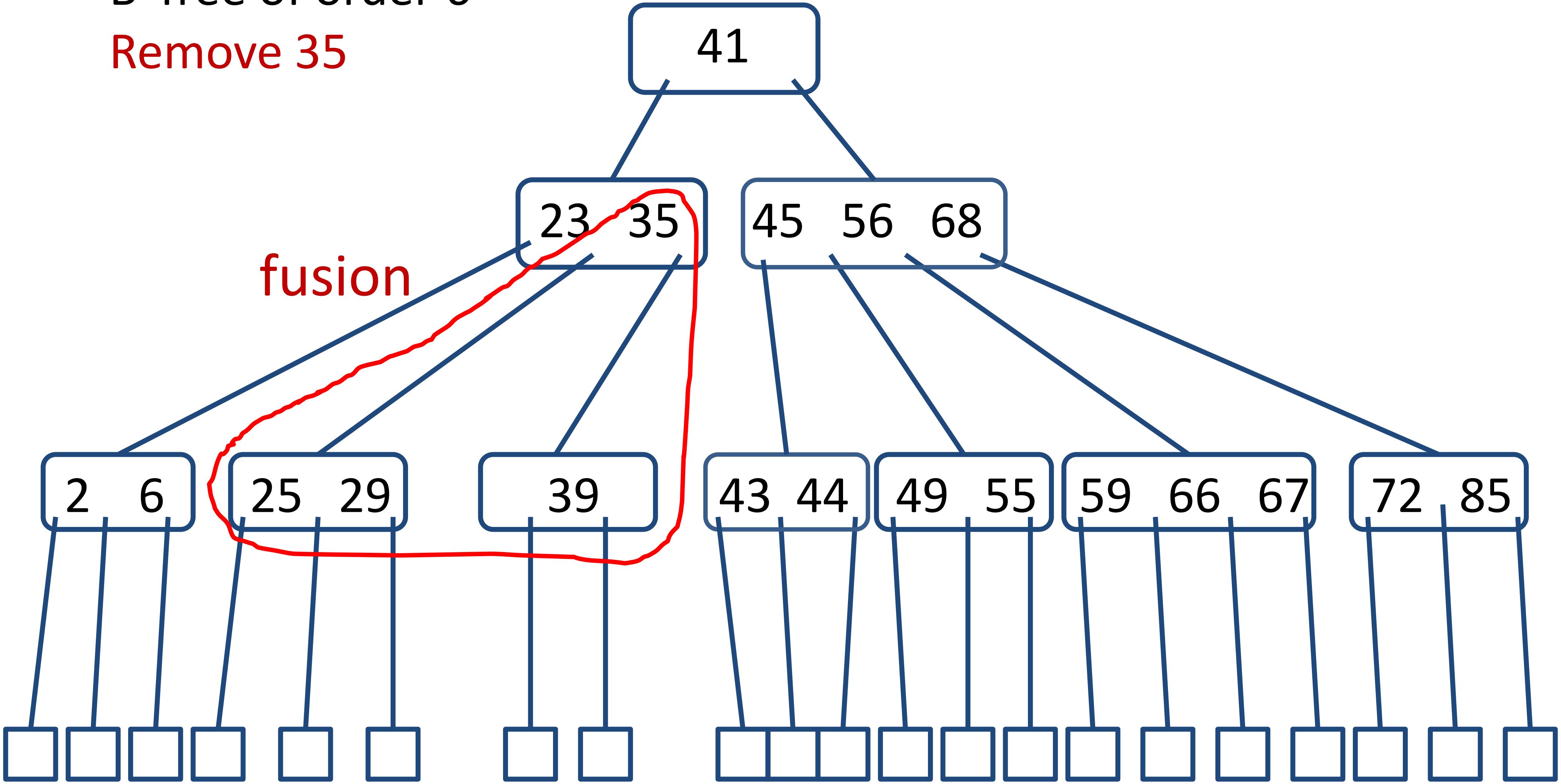
Remove 35



B-Trees

B-Tree of order 6

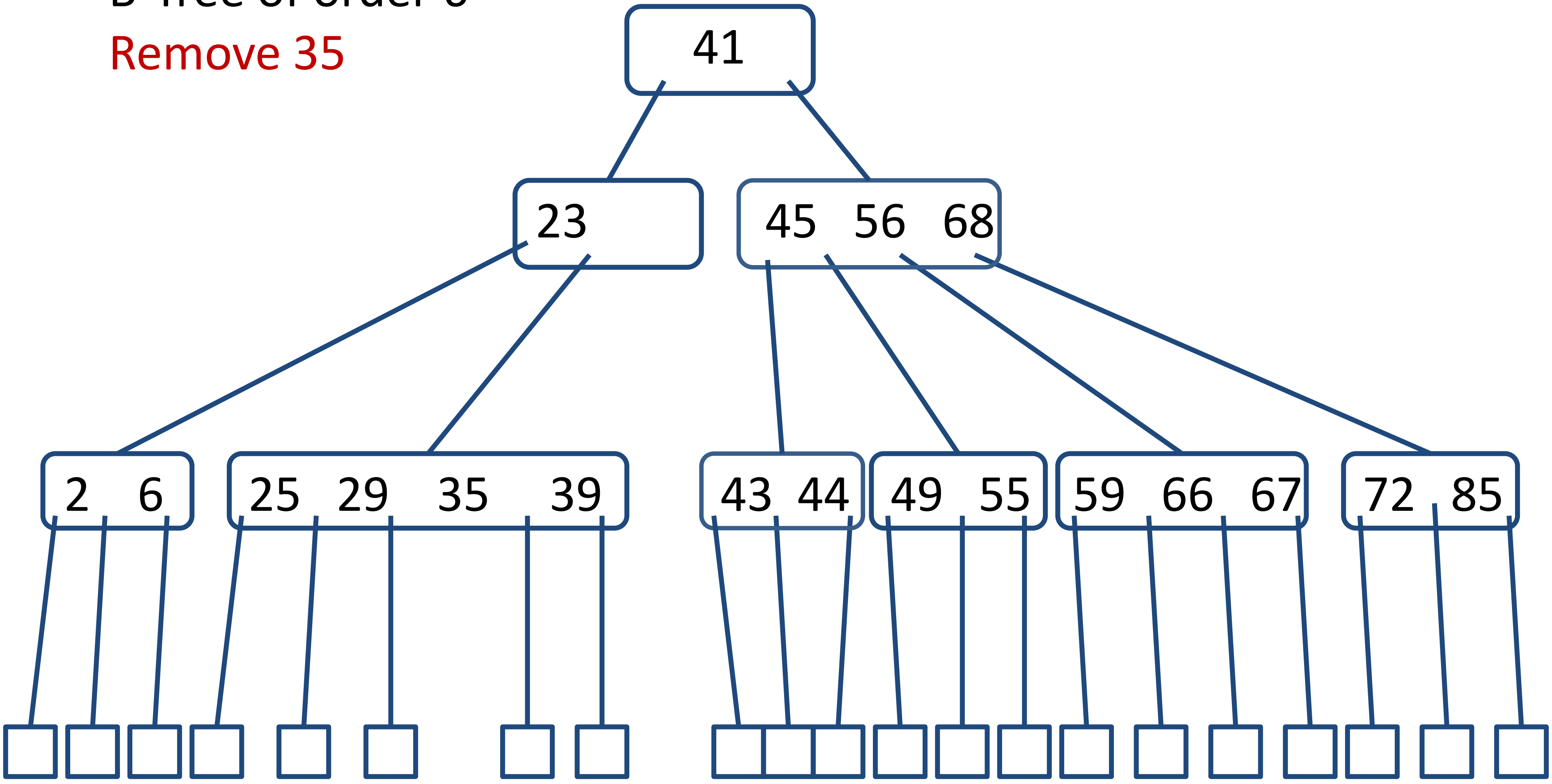
Remove 35



B-Trees

B-Tree of order 6

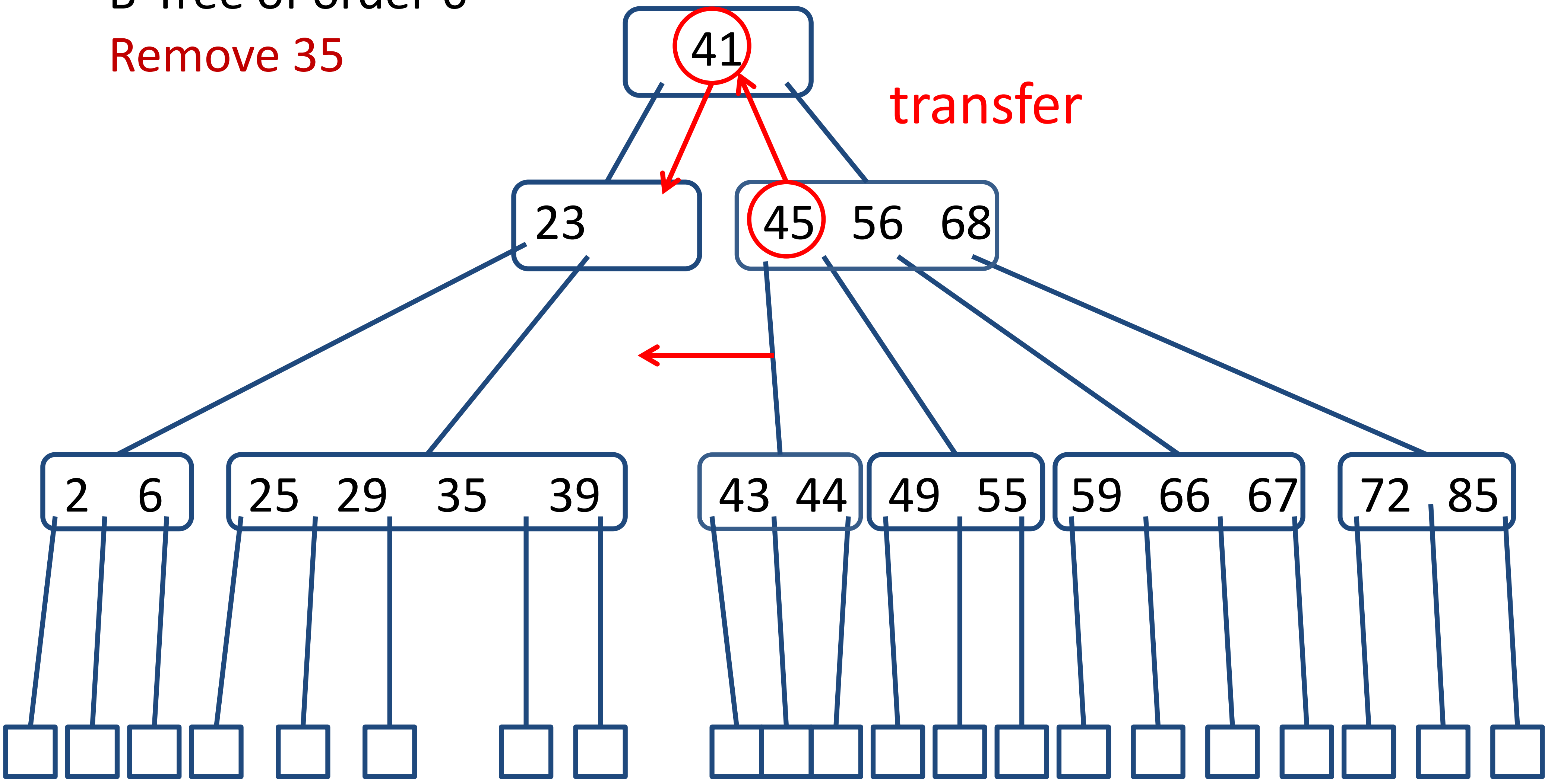
Remove 35



B-Trees

B-Tree of order 6

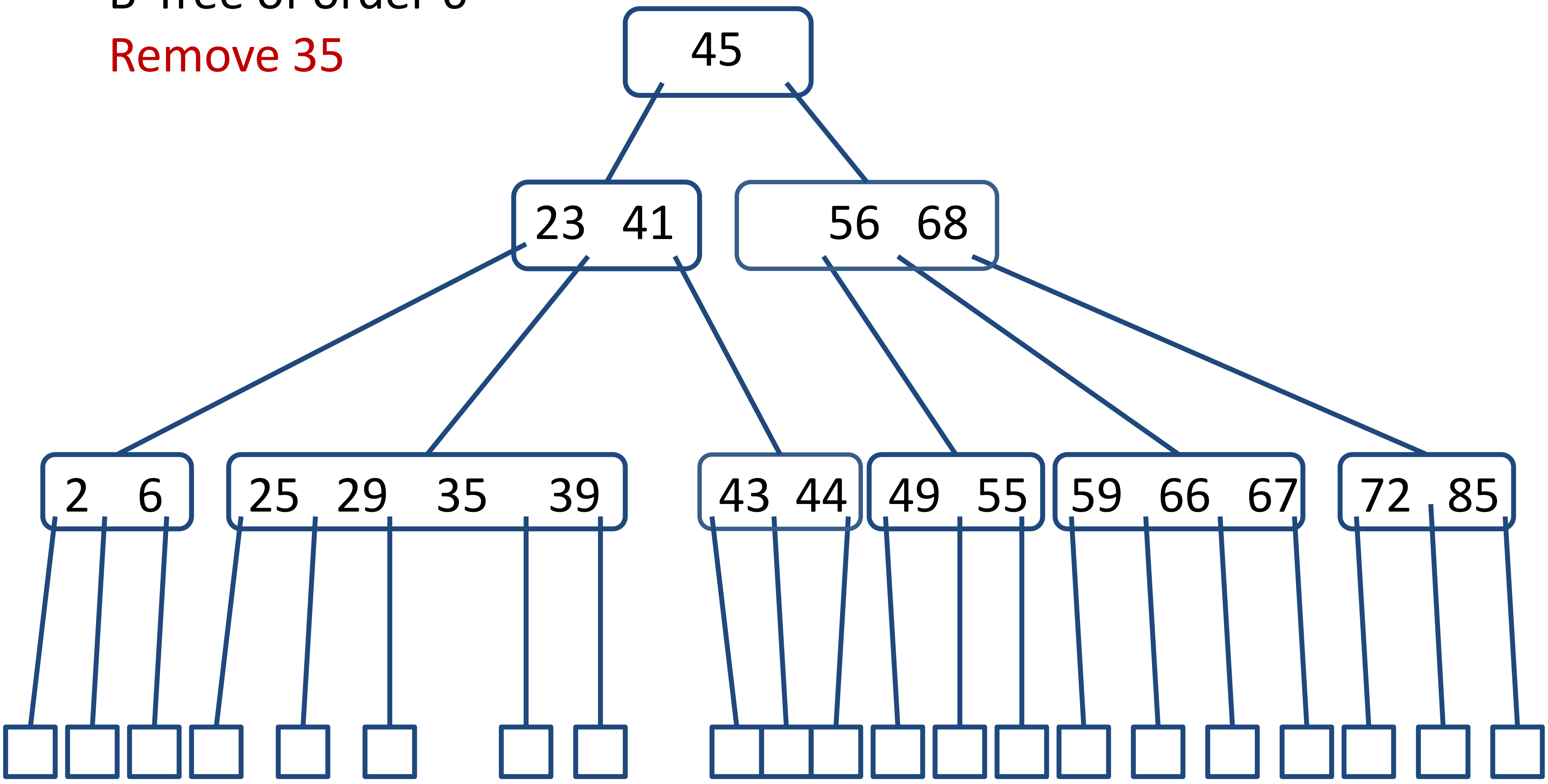
Remove 35



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***

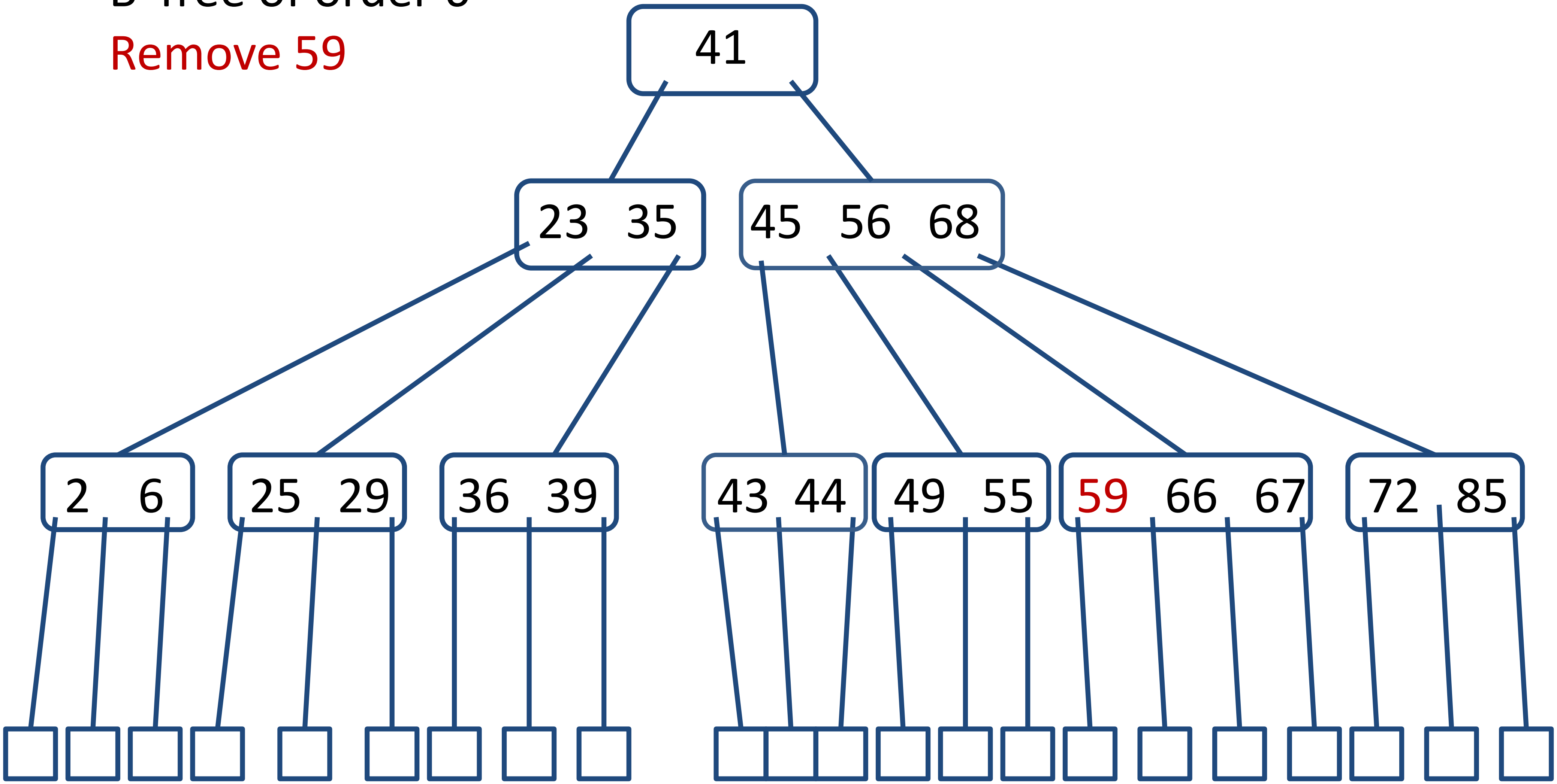
}

}

B-Trees

B-Tree of order 6

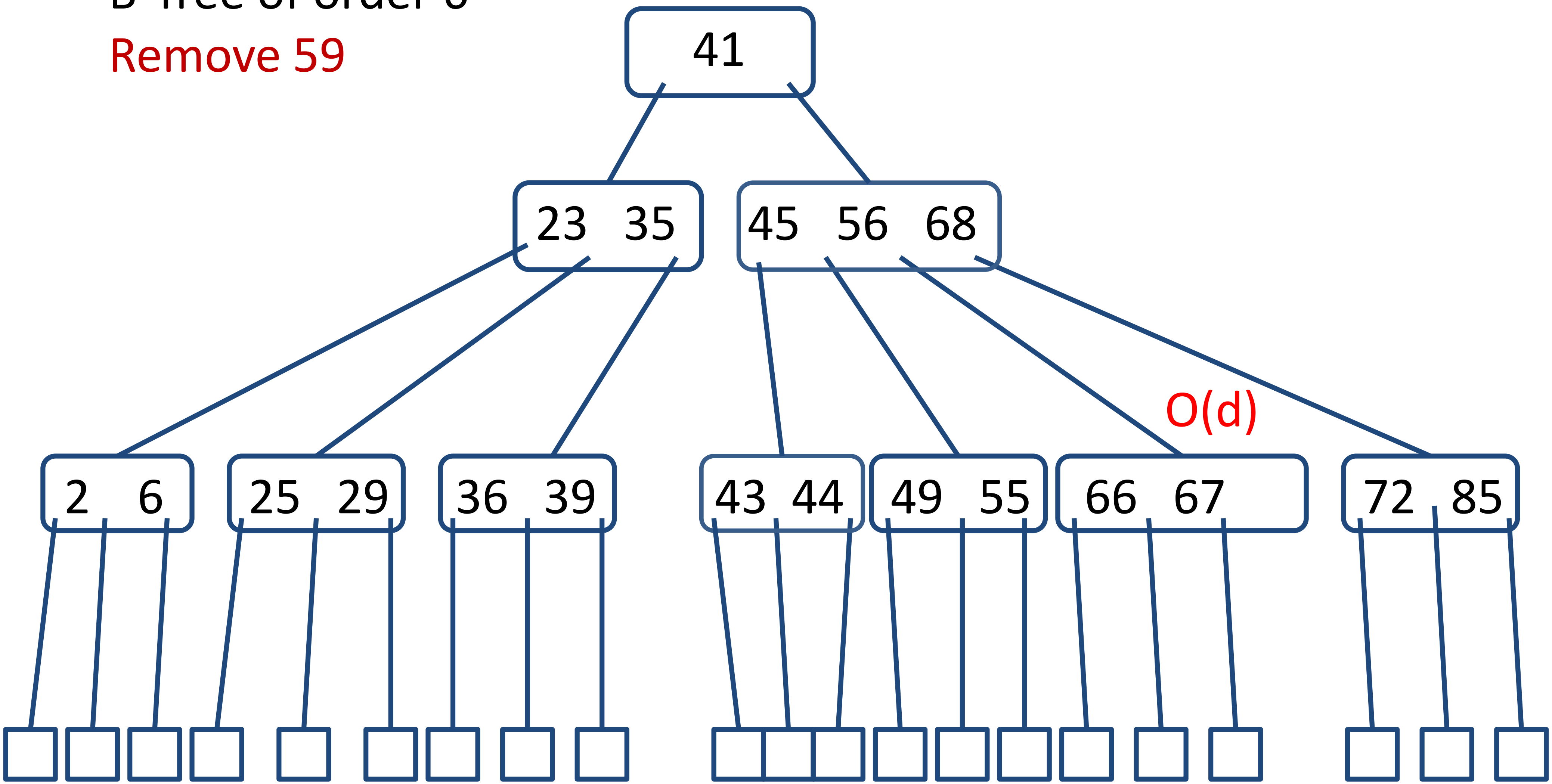
Remove 59



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

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***

}

}

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* ***underflows*** **do** { $O(d + \log d \times \log_d n)$

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***

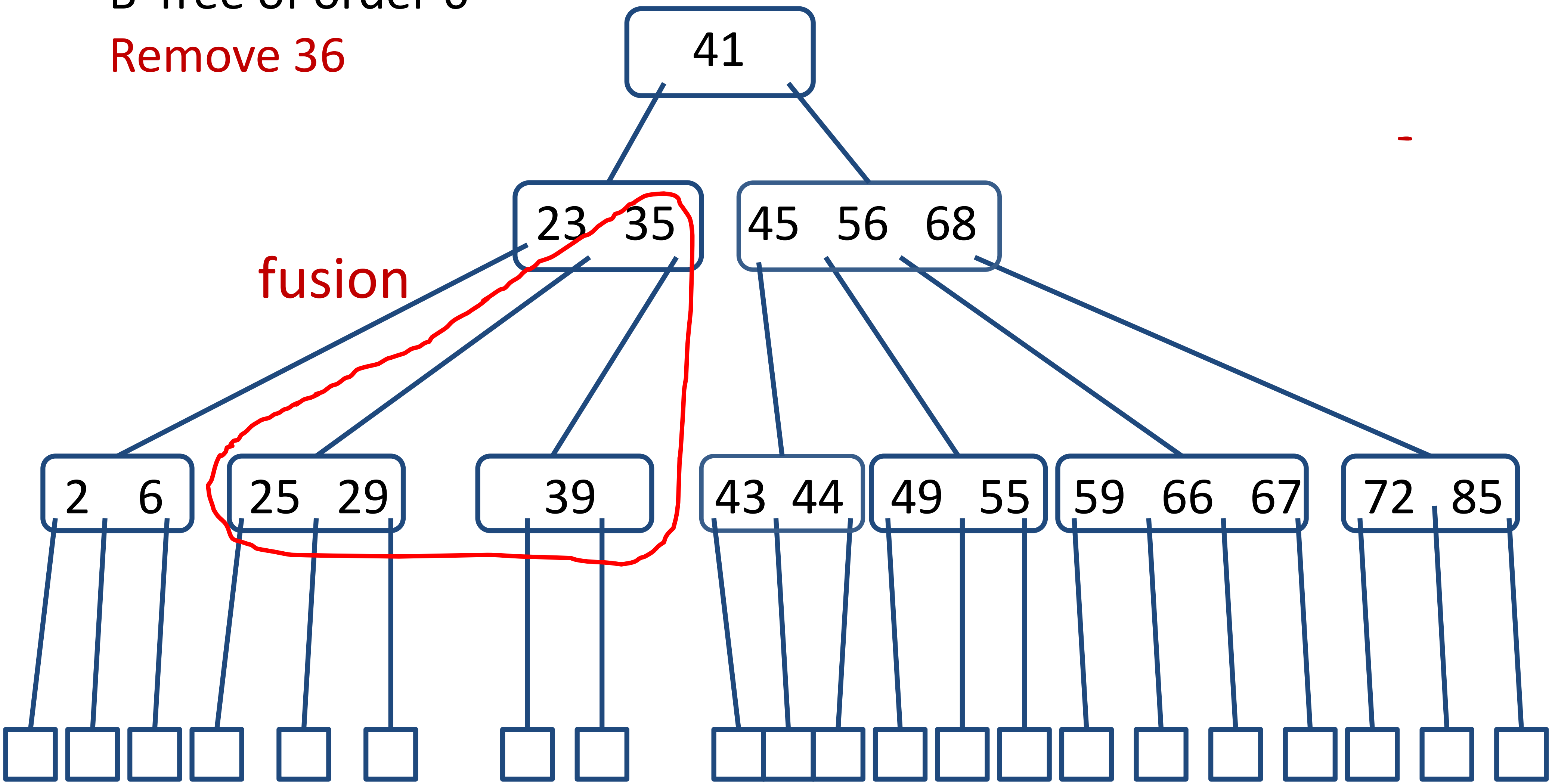
}

}

B-Trees

B-Tree of order 6

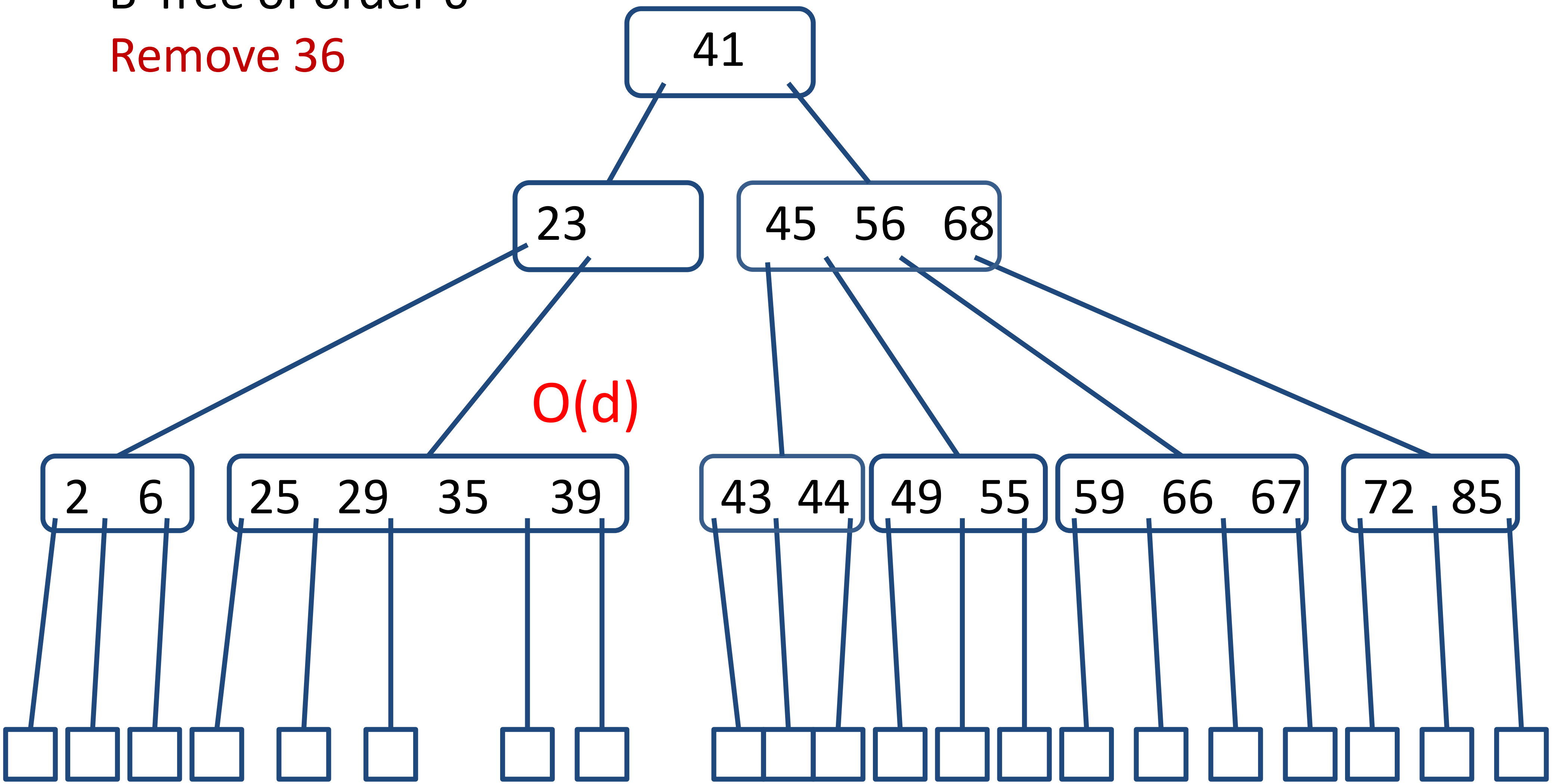
Remove 36



B-Trees

B-Tree of order 6

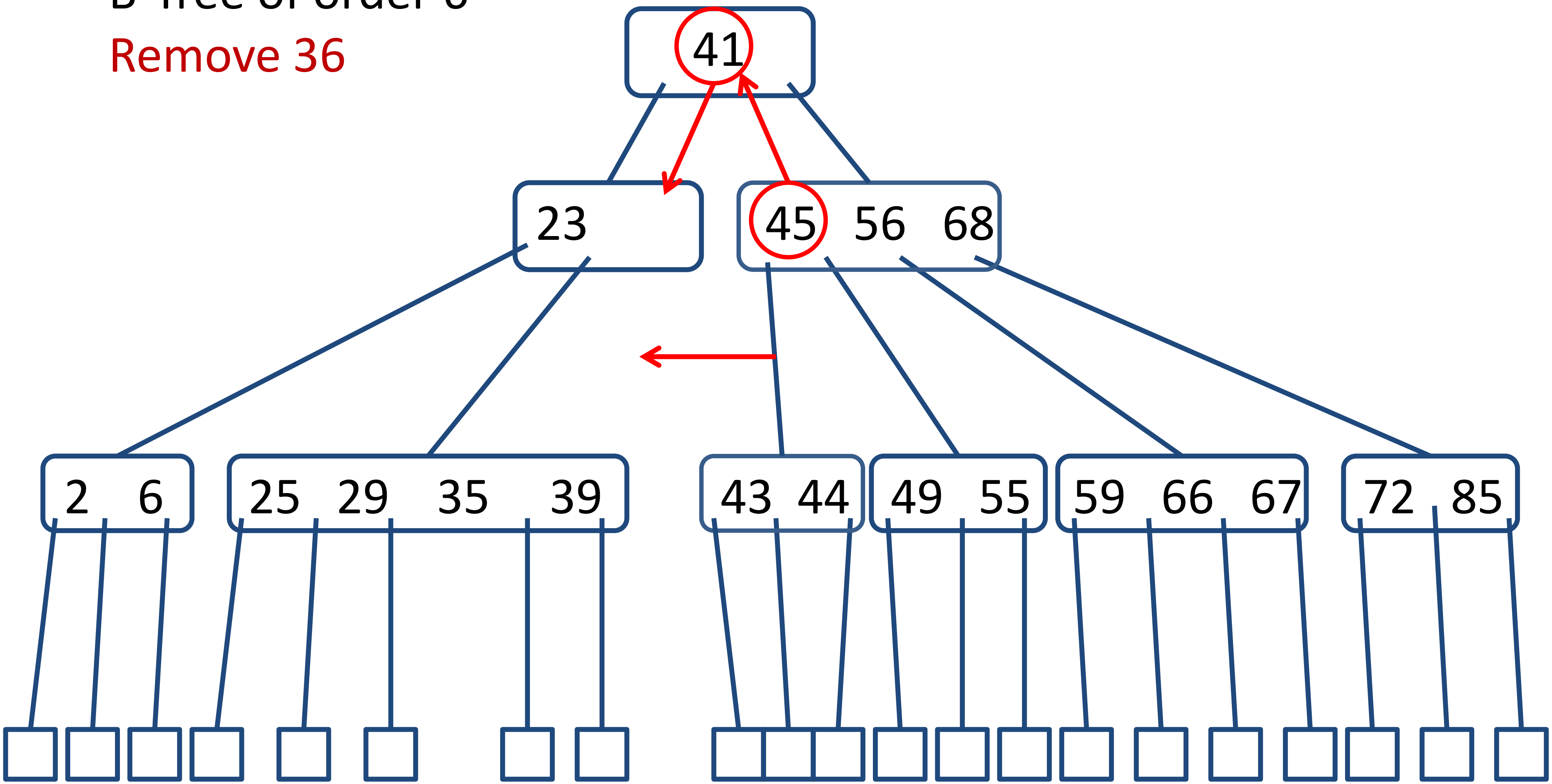
Remove 36



B-Trees

B-Tree of order 6

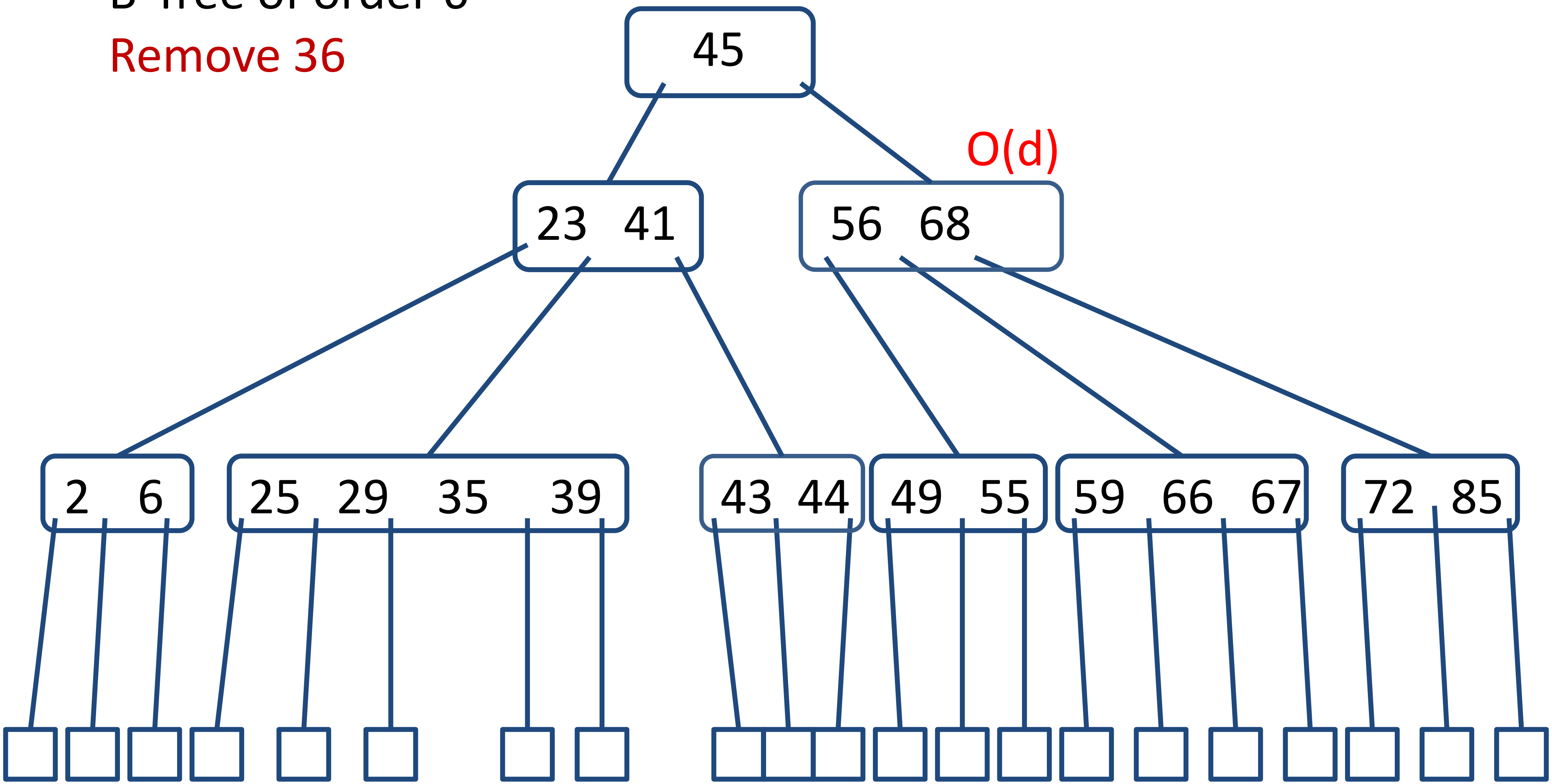
Remove 36



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*** $\Big\}$ O(log d × $\log_d$ n)

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

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***

}

}

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 — $O(d + \log d \times \log_d n)$

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***

}

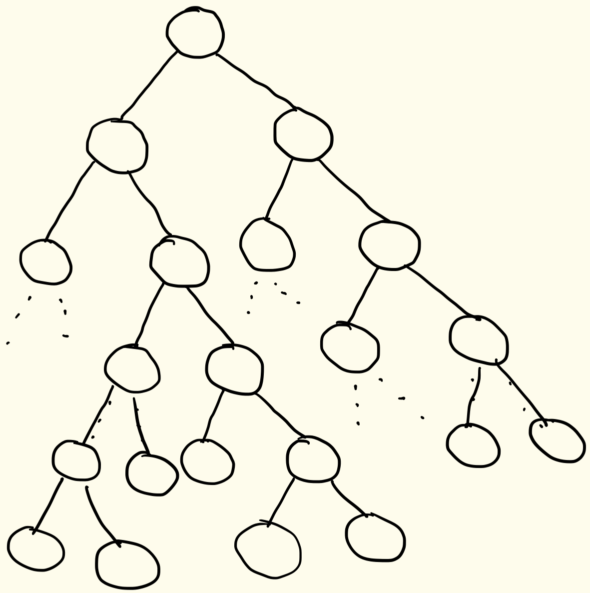
}

$O(d)$

$O(d \log_d n)$

Collection of 10^9 records

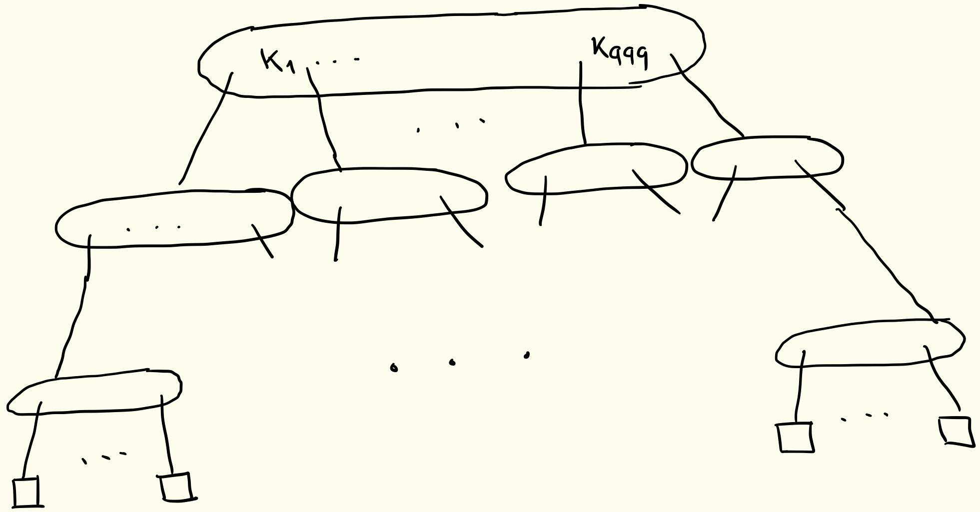
AVL tree



Height: $O(\log_2 n)$

$\log_2 10^9 \approx 30$

B-tree of degree 10^3

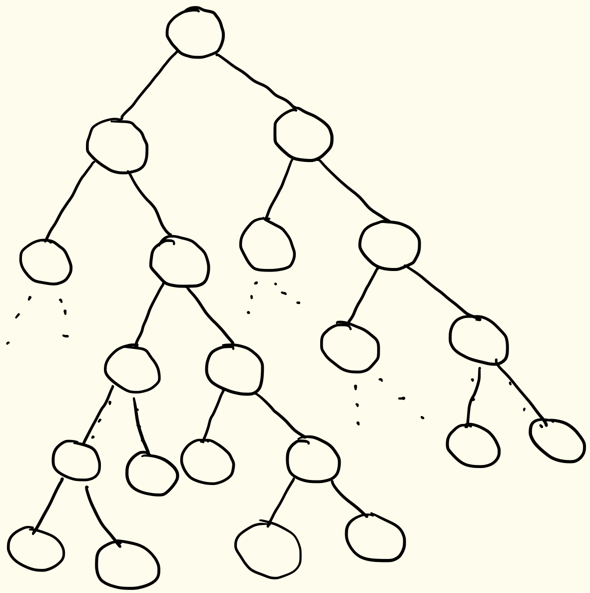


Height: $O(\log_d n)$

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

Collection of 10^9 records

AVL tree



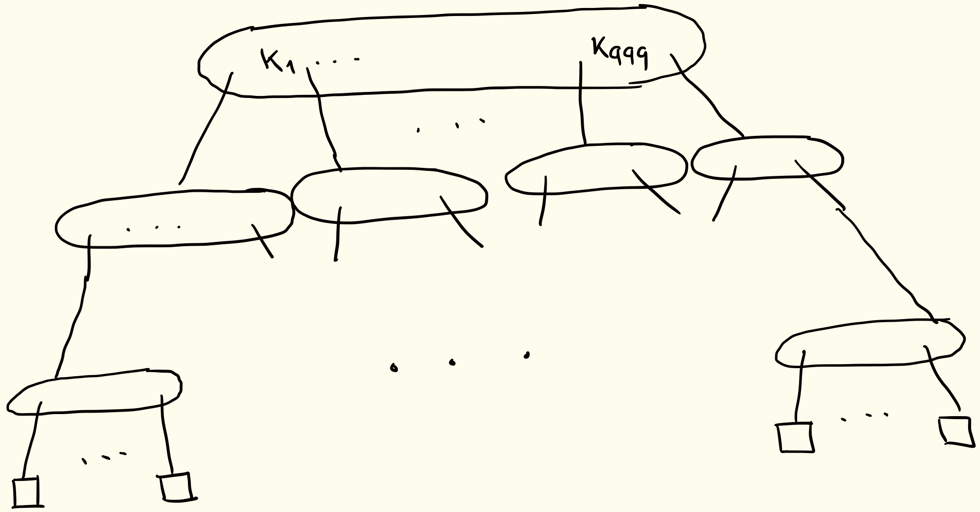
Height: $O(\log_2 n)$

$\log_2 10^9 \approx 30$

B-tree of degree 10^3

Binary search: $O(\log_2 d)$

$\log_2 10^3 \approx 10$



Height: $O(\log_d n)$

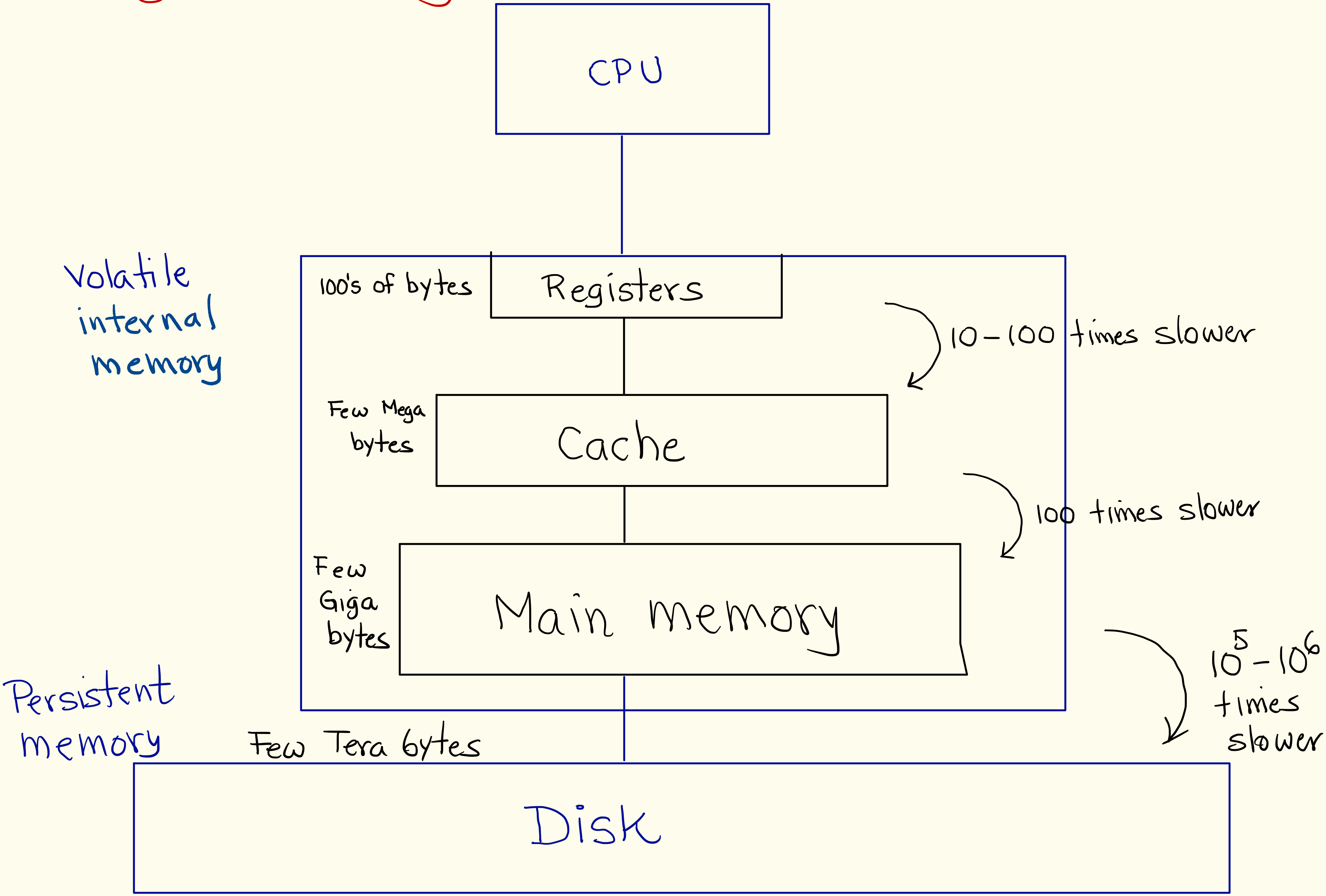
$\log_{1000} 10^9 = 3$

Number of comparisons to find a key

≈ 30

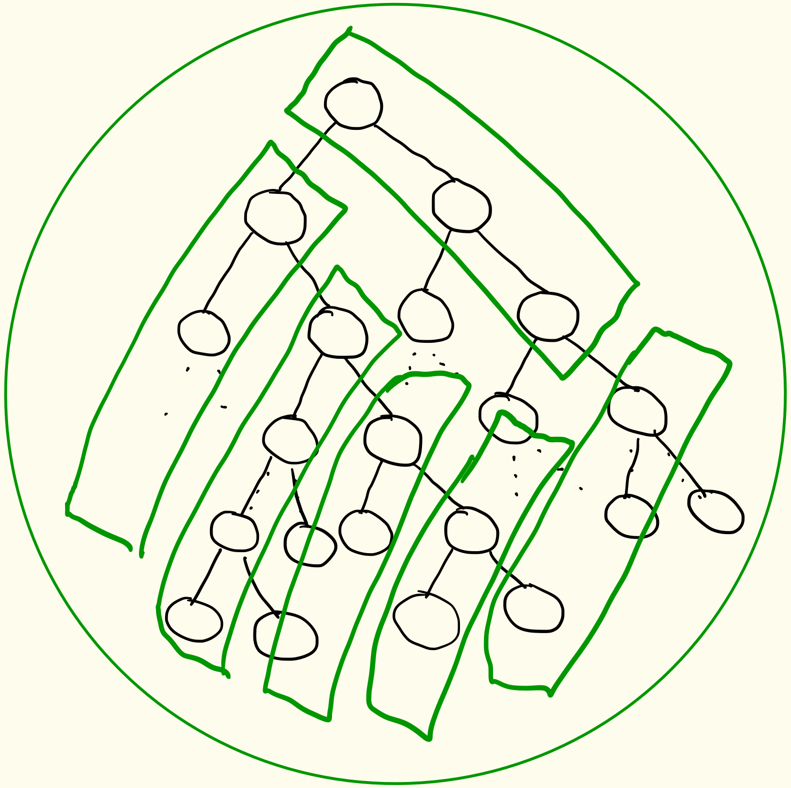
≈ 30

Memory Hierarchy

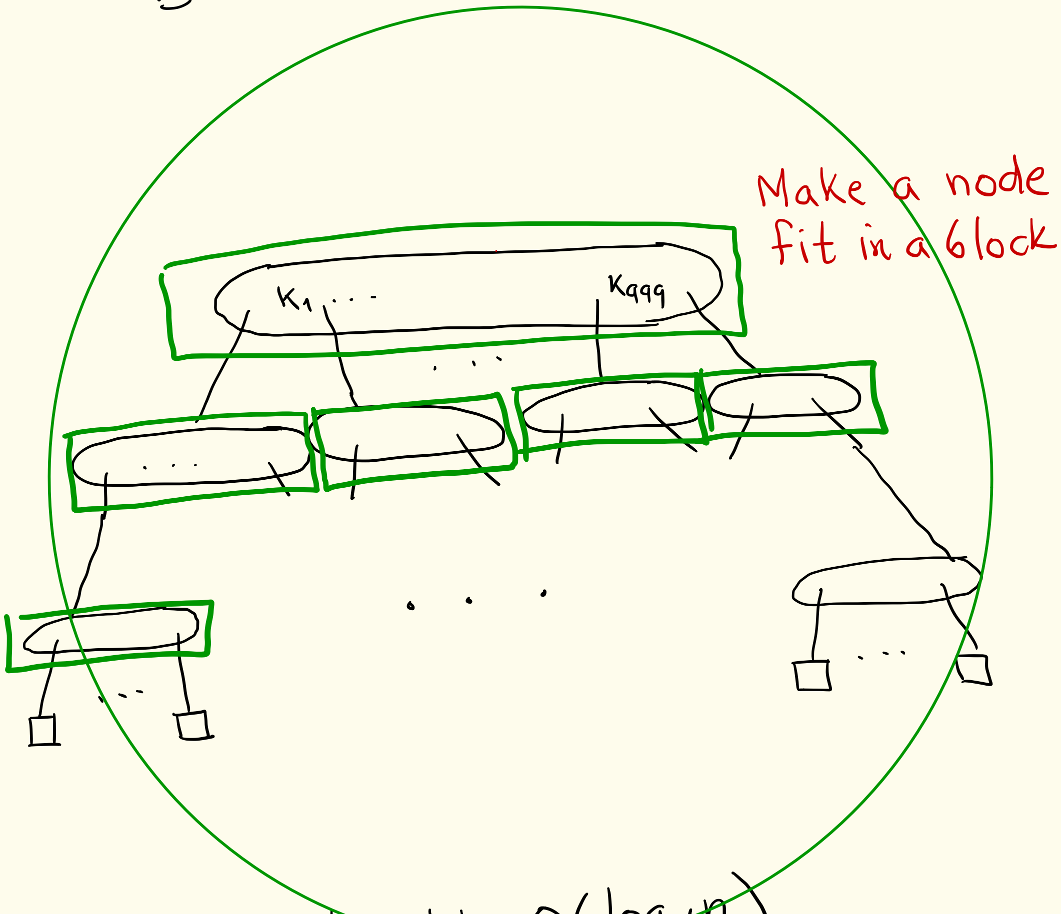


Collection of 10^9 records

AVL tree



B-tree of degree 10^3



Height: $O(\log_2 n)$

$\log_2 10^9 \quad 30$

Height: $O(\log_d n)$

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

	AVL	B-tree
Number of disk accesses:	30	3

So a B-tree is much better.

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.