# CS 240 – Data Structures and Data Management

## Module 11: External Memory


O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo


Winter 2025

# Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - *a*-*b* Trees
    - B-Trees

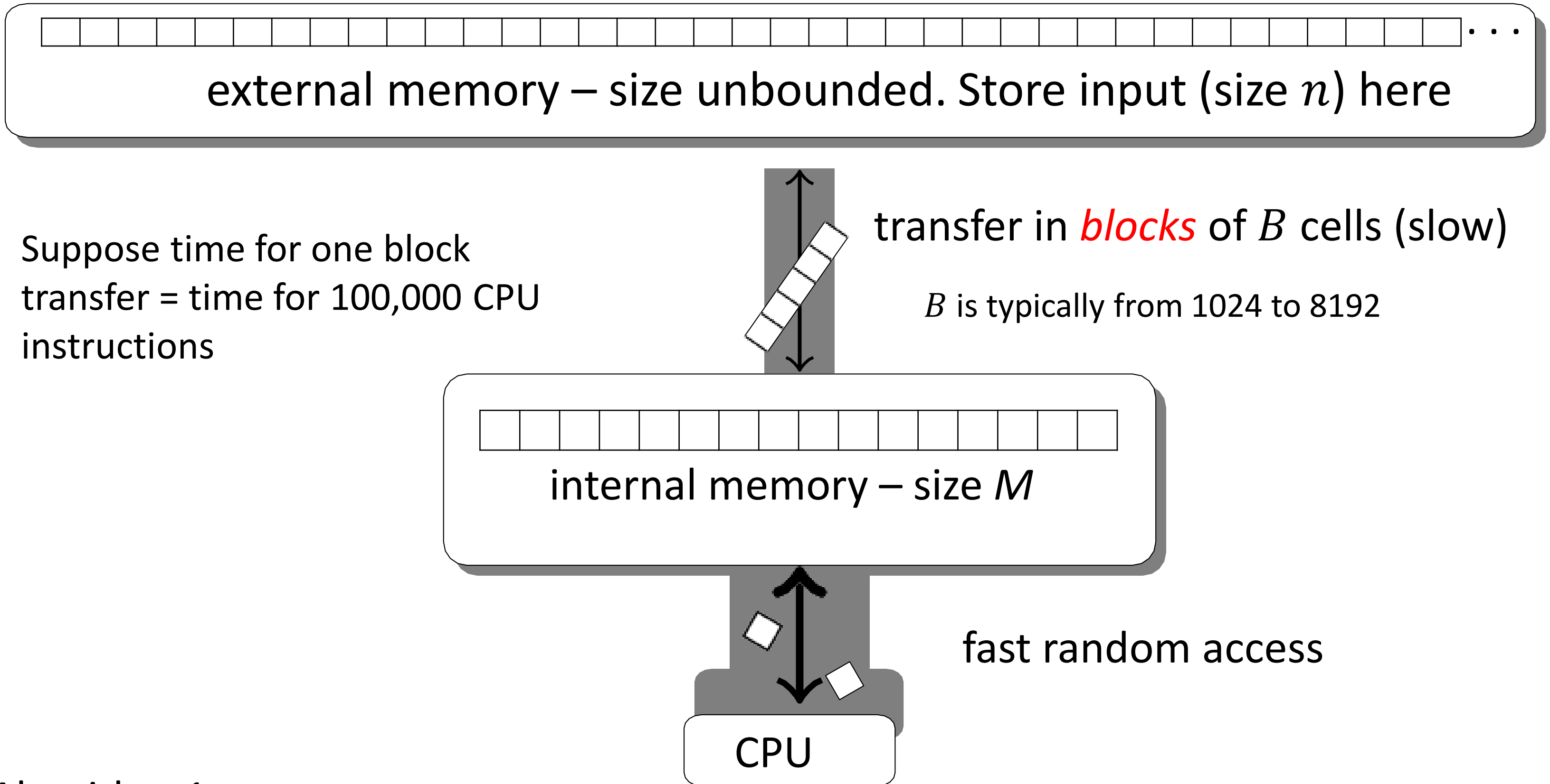
# Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

# Different levels of memory

- RAM model: access to any memory location takes constant time
  - not realistic
- Current architectures
  - registers: super fast, very small
  - cache L1, L2: very fast, less small
  - main memory: fast, large
  - disk or cloud: slow, very large
- How to adapt algorithms to take memory hierarchy into consideration?
  - desirable to minimize transfer between slow/fast memory
- Define computer model that models hierarchy across which must transfer
  - focus on 2 levels of hierarchy: main (internal) memory and disk or cloud (external) memory
  - accessing a single location in external memory automatically loads a whole block (or "page")
    - one block access can take as much time as executing 100,000 CPU instructions
    - need to care about the number of block accesses

# External-Memory Model (EMM)




Suppose time for one block transfer = time for 100,000 CPU instructions

- Algorithm 1
  ~~1,000 CPU instructions~~ + 1,000 block transfers = ~~1,000~~+1,000·100,000 = ~~$10^3$~~ + $10^8$
- Algorithm 2
  ~~10,000 CPU instructions~~ + 10 block transfers = ~~10,000~~+10·100,000 = ~~$10^4$~~ + $10^6$

dominating factors

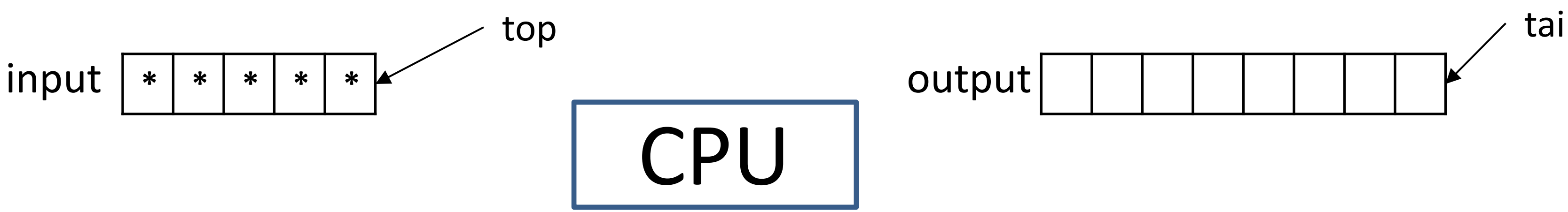
- **New cost of computation:** number of blocks transferred (or 'probes', 'disk transfers', 'page loads') between internal and external memory
- We will revisit ADTs/problems with the objective of minimizing block transfers

# Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

# Stream Based Algorithms in Internal Memory

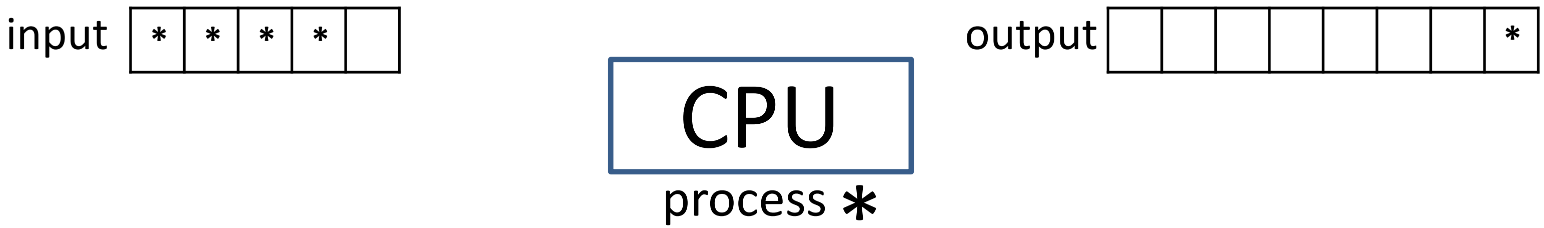
- Studied algorithms that handle input/output with streams
  - access only top item in input stream, append only to tail of output stream




- Repeat
  1. take item off top of the input
  2. process item
  3. put the result of processing at the tail of output

# Stream Based Algorithms in Internal Memory

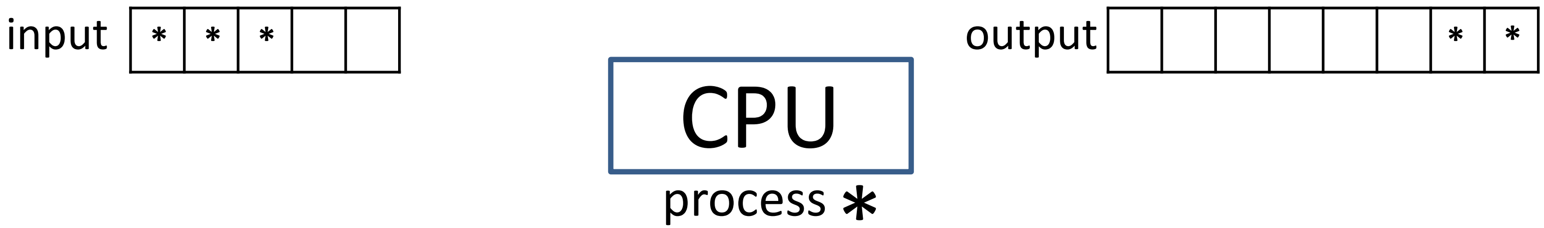
- Studied algorithms that handle input/output with streams
  - access only top item in input stream, append only to tail of output stream




- Repeat
  1. take item off top of the input
  2. process item
  3. put the result of processing at the tail of output

# Stream Based Algorithms in Internal Memory

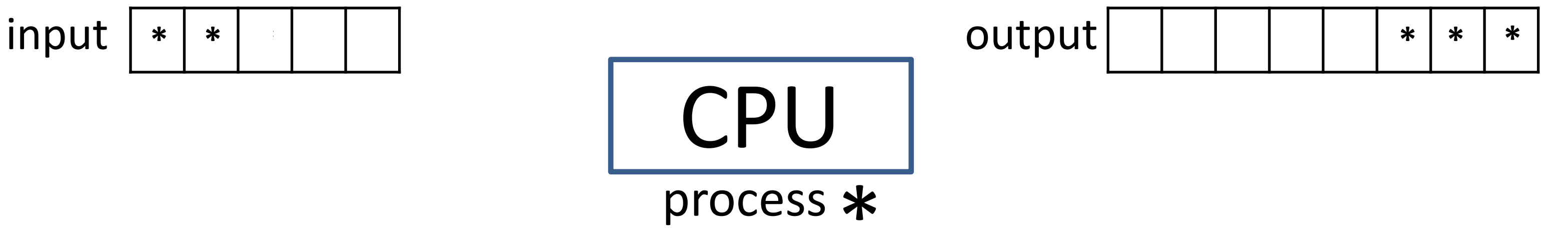
- Studied algorithms that handle input/output with streams
  - access only top item in input stream,  append only to tail of output stream




- Repeat
  1. take item off top of the input
  2. process item
  3. put the result of processing at the tail of output

# Stream Based Algorithms in Internal Memory

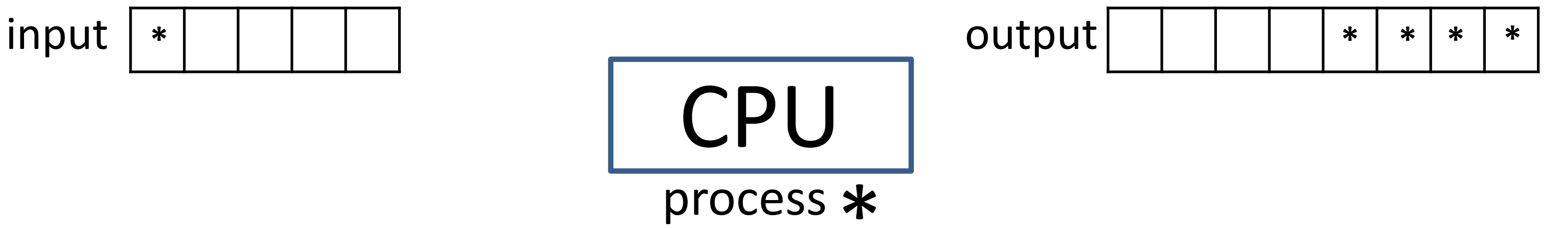
- Studied algorithms that handle input/output with streams
  - access only top item in input stream,  append only to tail of output stream




- Repeat
  1. take item off top of the input
  2. process item
  3. put the result of processing at the tail of output

# Stream Based Algorithms in Internal Memory

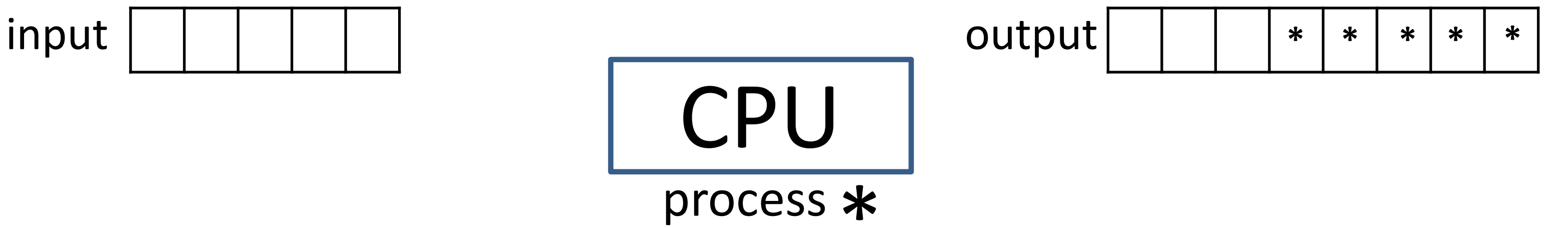
- Studied algorithms that handle input/output with streams
  - access only top item in input stream, append only to tail of output stream




- Repeat
  1. take item off top of the input
  2. process item
  3. put the result of processing at the tail of output

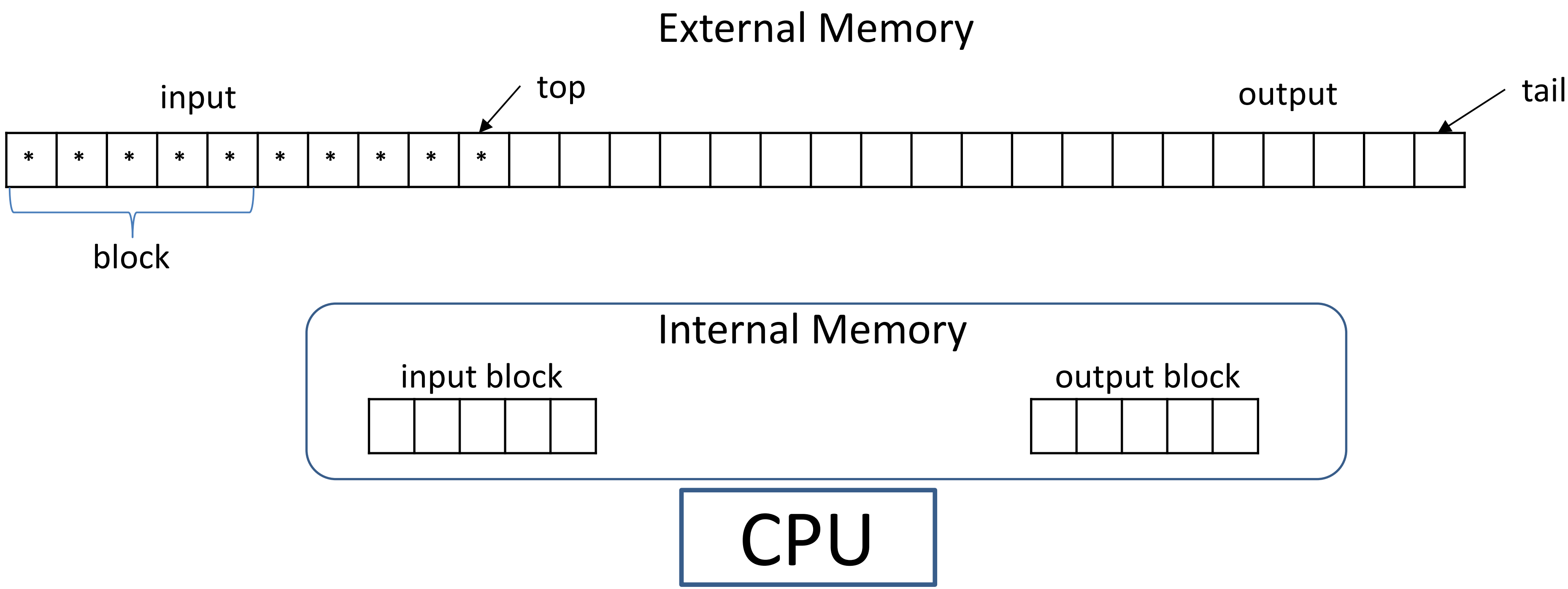
# Stream Based Algorithms in Internal Memory

- Studied algorithms that handle input/output with streams
  - access only top item in input stream,  append only to tail of output stream



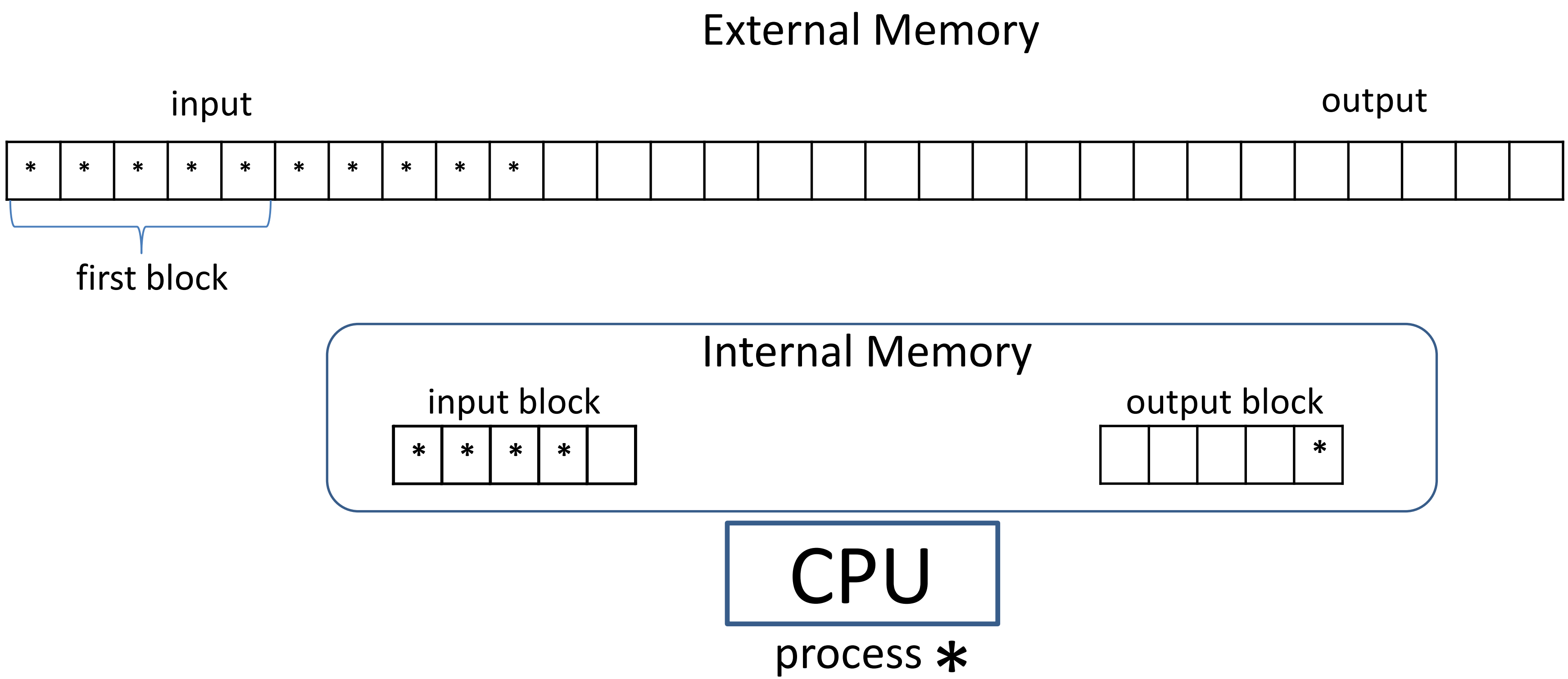

- Repeat
  1. take item off top of the input
  2. process item
  3. put the result of processing at the tail of output

# Stream Based Algorithms in External Memory

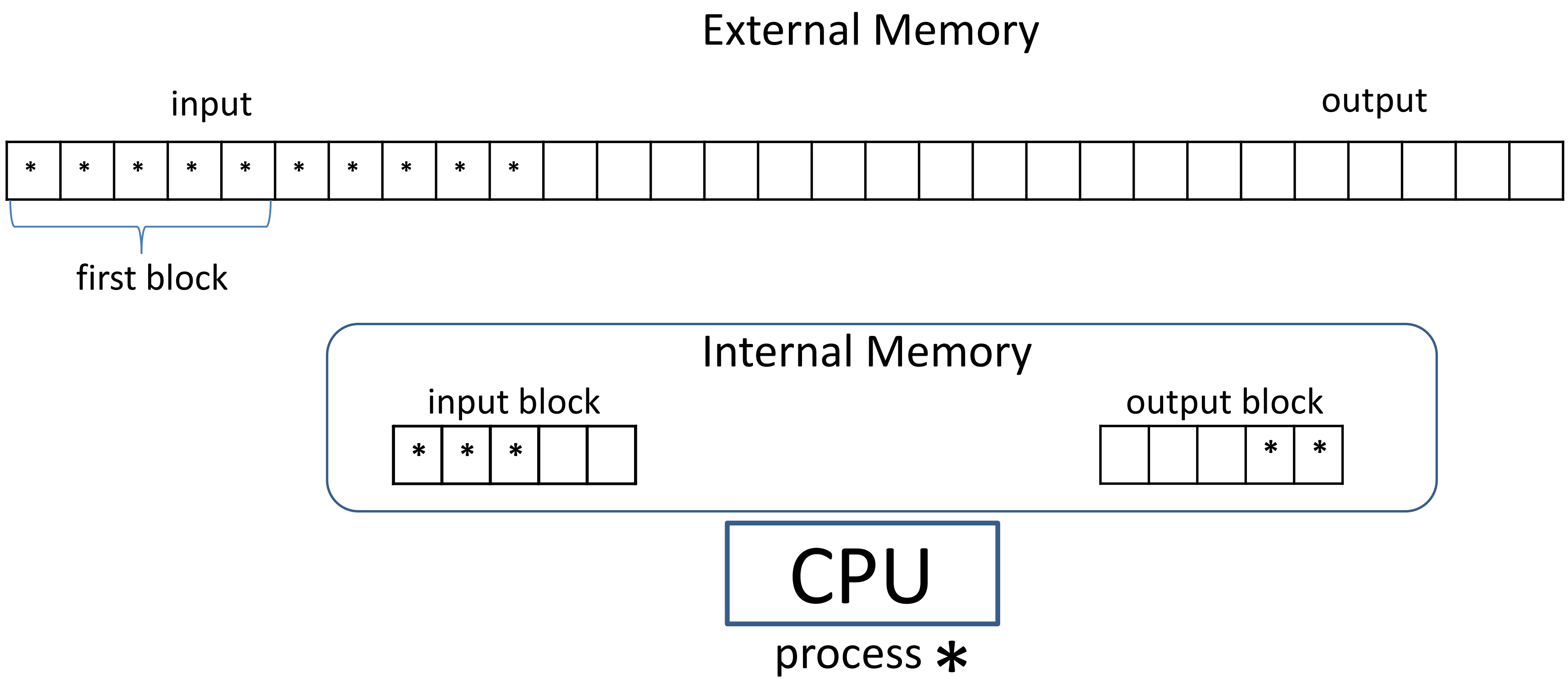



- Data in external memory has to be placed in internal memory before it can be processed
- Idea: perform the same algorithm as before, but in “block-wise” manner
  - have one block for input, one block for output in internal memory
  - transfer a block (size $B$) to internal memory, process it as before, store result in output block
  - when output stream is of size $B$ (full block), transfer it to external memory
  - when current block is in internal memory is fully processed, transfer next unprocessed block from external memory

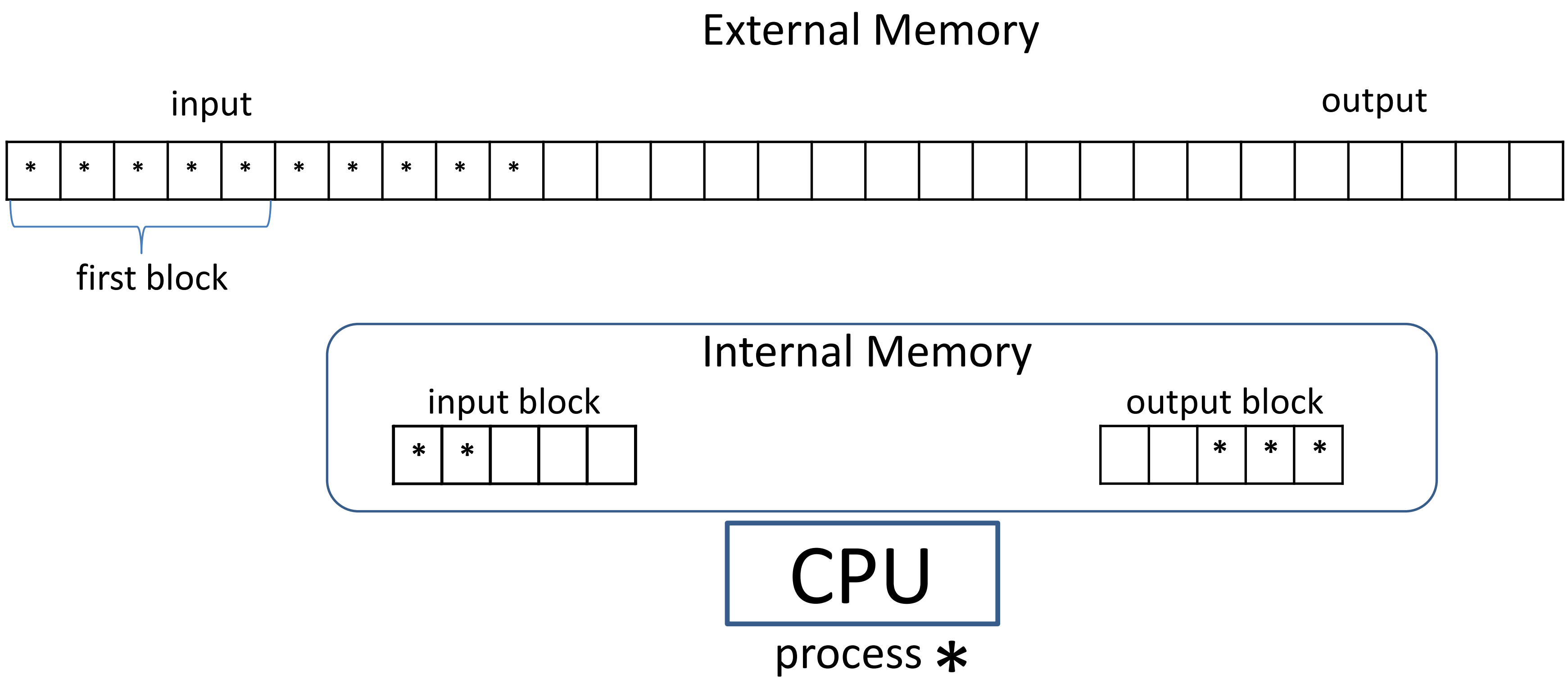
# Stream Based Algorithms in External Memory



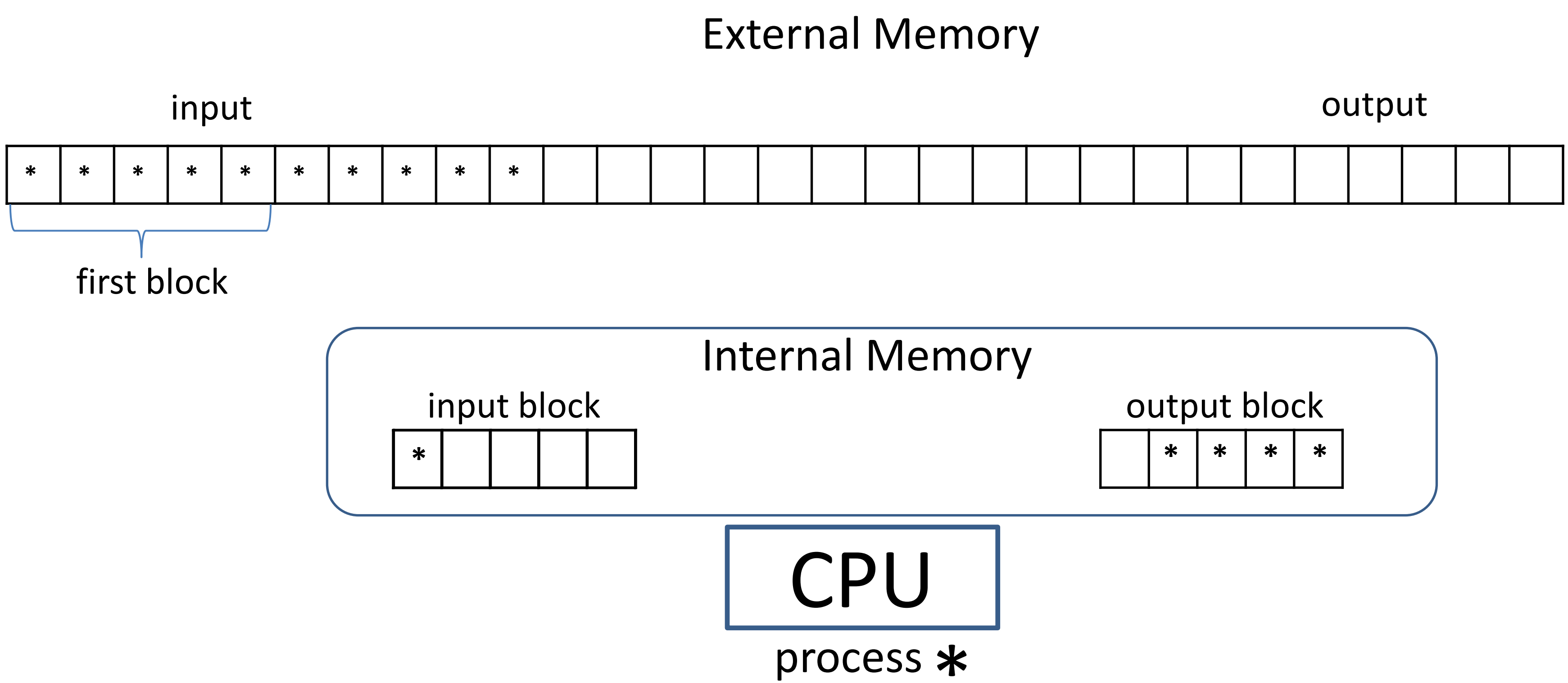
# Stream Based Algorithms in External Memory



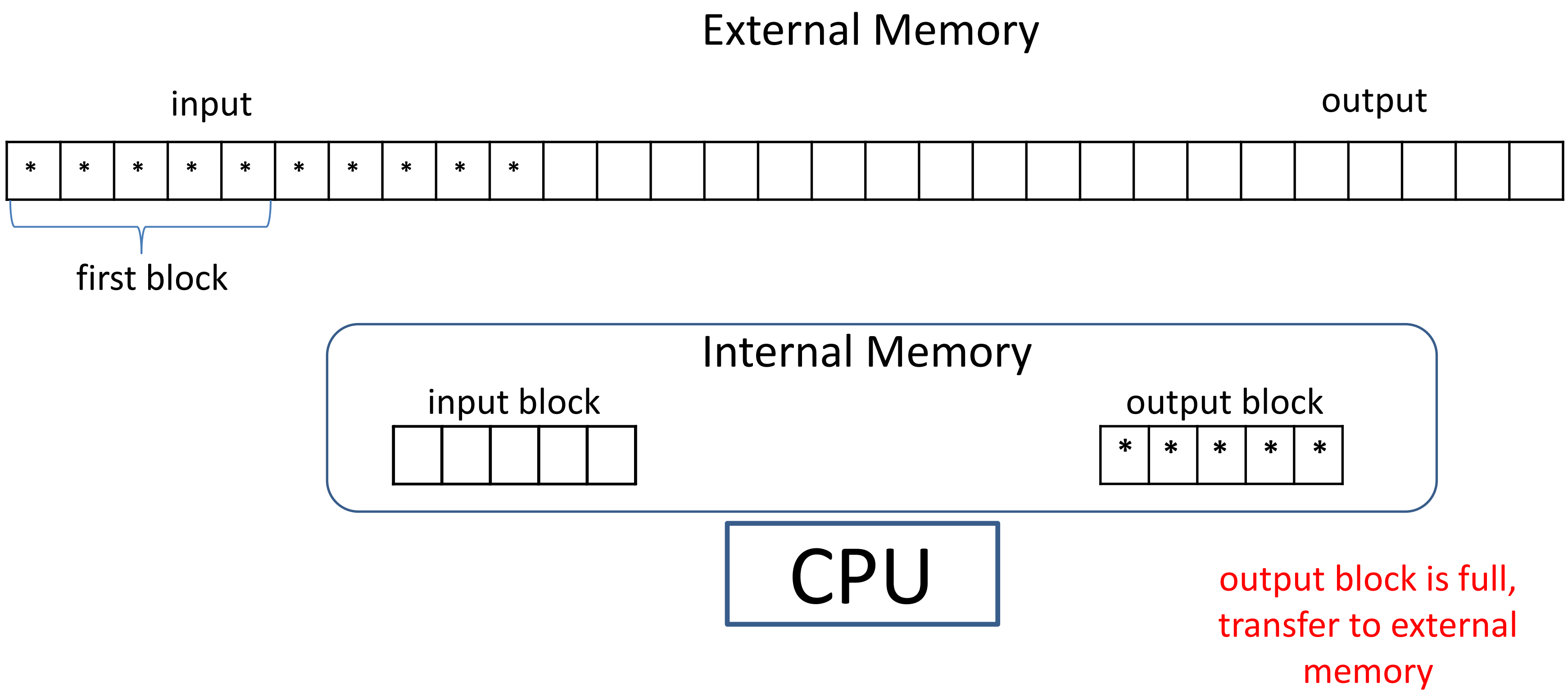
# Stream Based Algorithms in External Memory



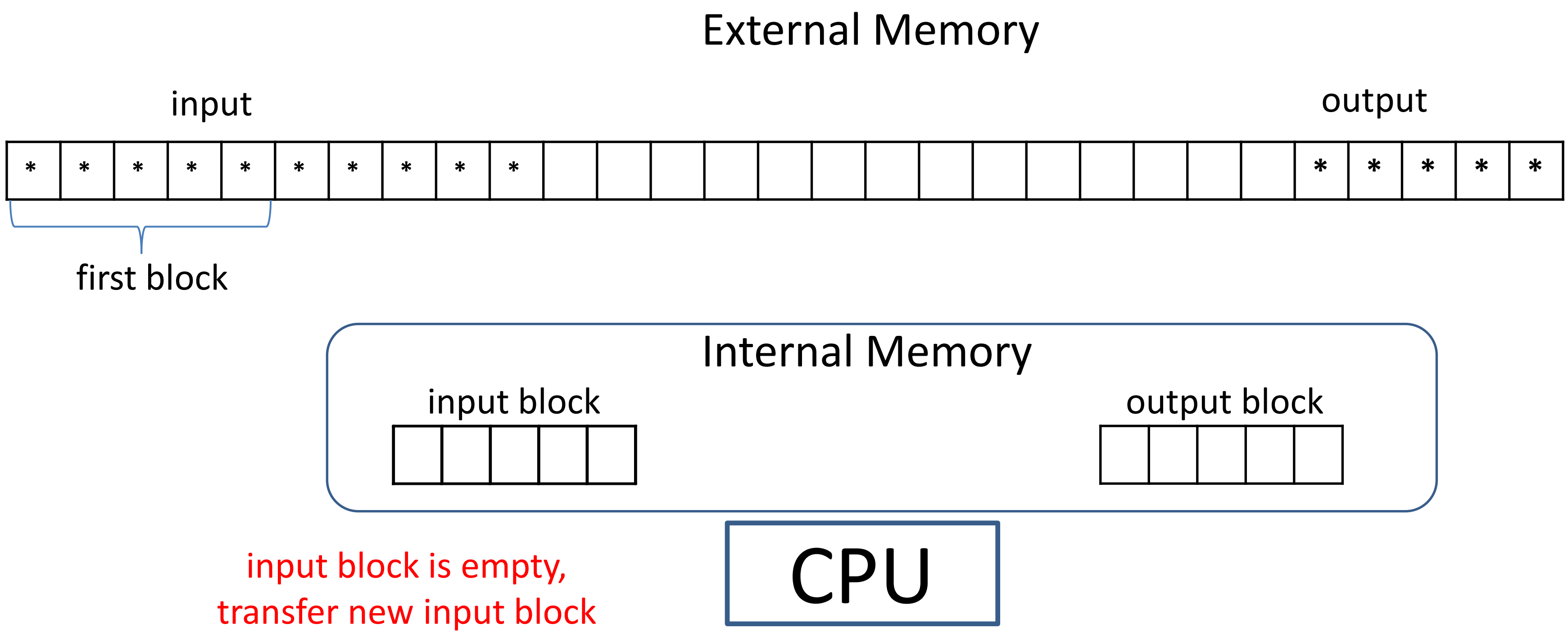
# Stream Based Algorithms in External Memory



# Stream Based Algorithms in External Memory

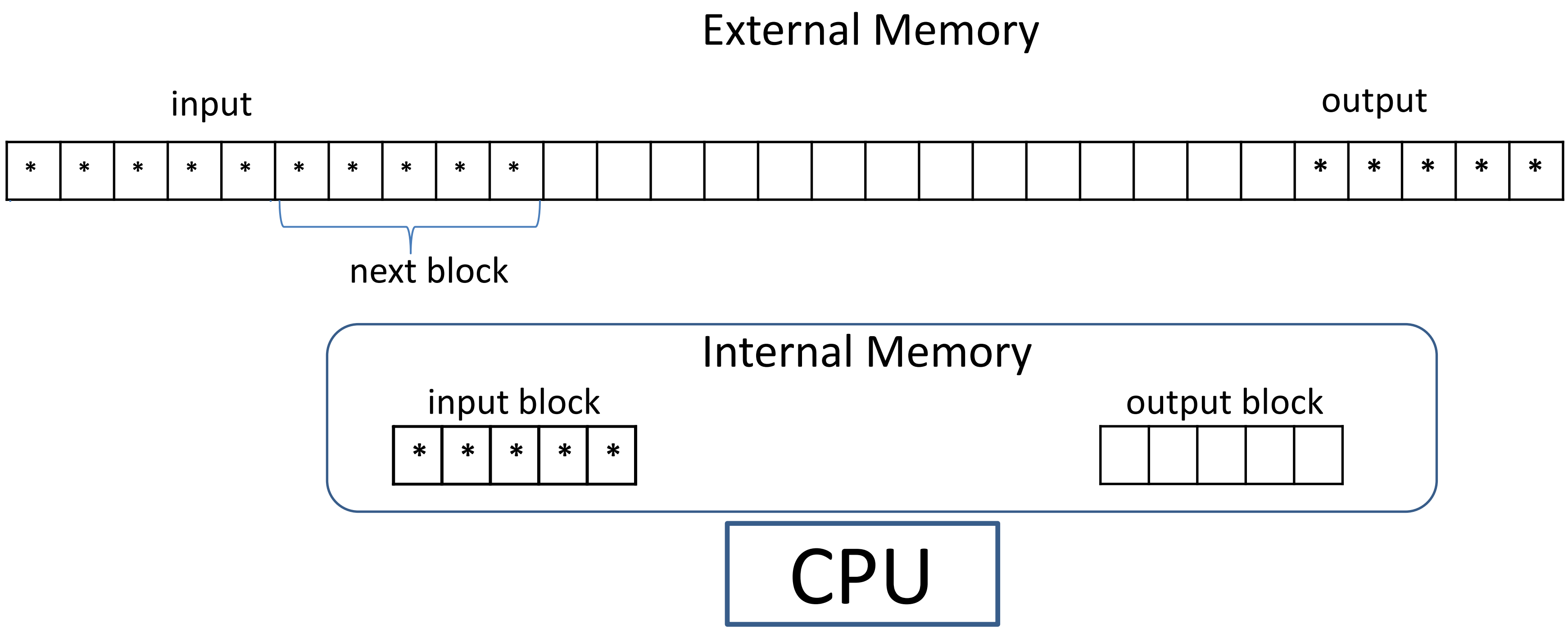


# Stream Based Algorithms in External Memory

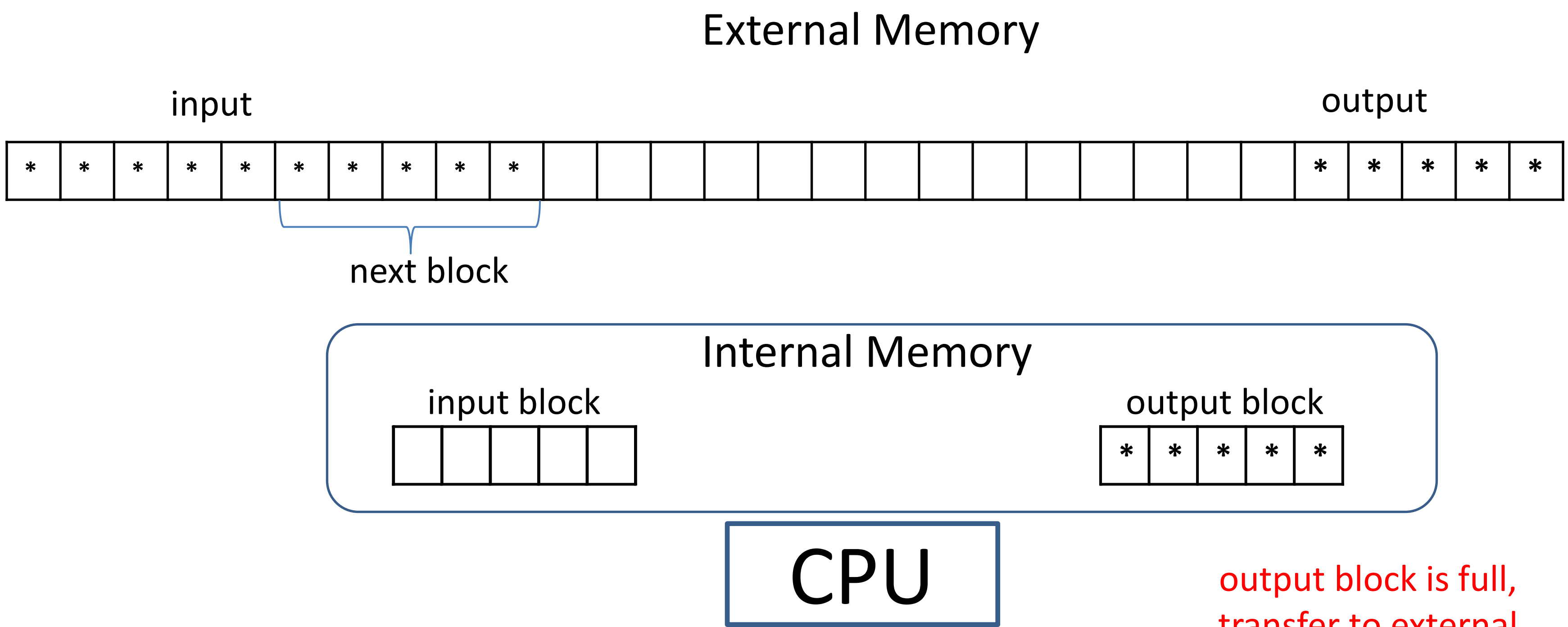



input block is empty,
transfer new input block
from external memory

# Stream Based Algorithms in External Memory

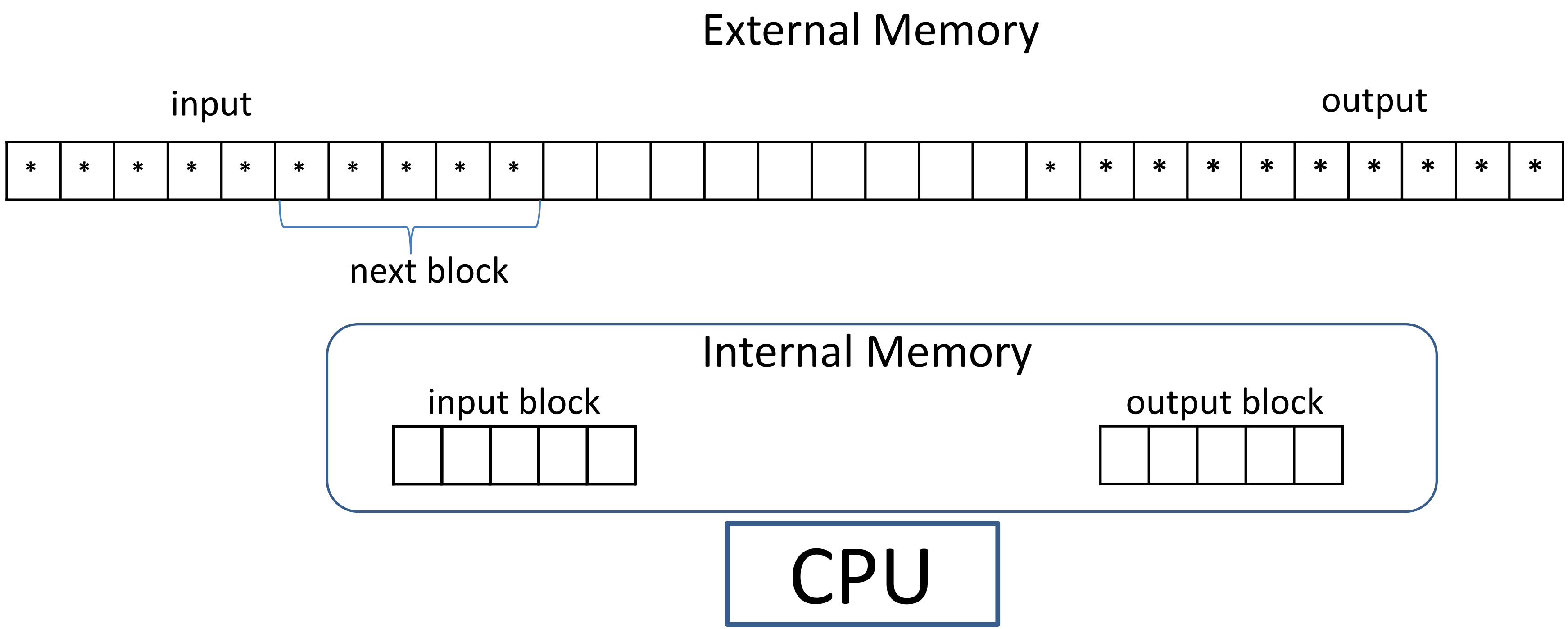


# Stream Based Algorithms in External Memory




output block is full, transfer to external memory

# Stream Based Algorithms in External Memory




- Running time (recall that we only count the block transfers now)
  - input stream: $\frac{n}{B}$ block transfers to read input of size $n$
  - output stream: $\frac{s}{B}$ block transfers to write output of size $s$
- Running time is *automatically* as efficient as possible for external memory
  - any algorithm needs at least $\frac{n}{B}$ block transfers to read input of size $n$ and $\frac{s}{B}$ block transfers to write output of size $s$

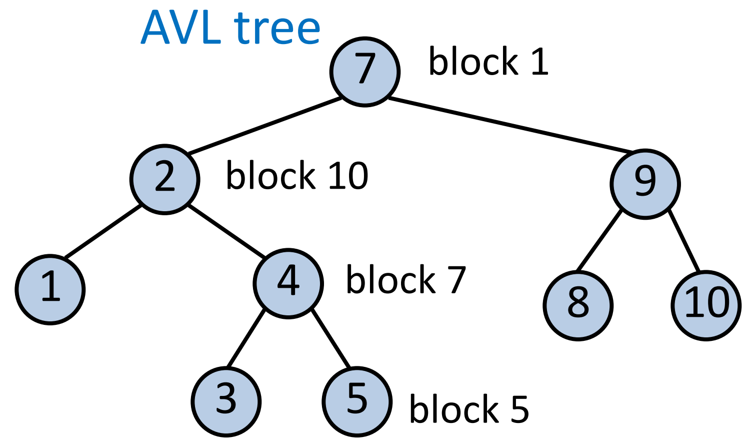
# Stream Based Algorithms in External Memory

- Methods below use stream input/output model, therefore need $\Theta\left(\frac{n}{B}\right)$ block transfers, assuming output size is $O(n)$
  - Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore
    - assuming pattern $P$ fits into internal memory
  - Text compression: Huffman, run-length encoding, Lempel-Ziv-Welch
  - Sorting: *merge-sort* can be implemented with $O\left(\frac{n}{B}\log n\right)$ block transfers
  - Bzip2 cannot be streamed as we described
    - can compress in 'blocks'
      - not as good as the whole text compression, but better than nothing

# Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

# Dictionaries in External Memory: Motivation



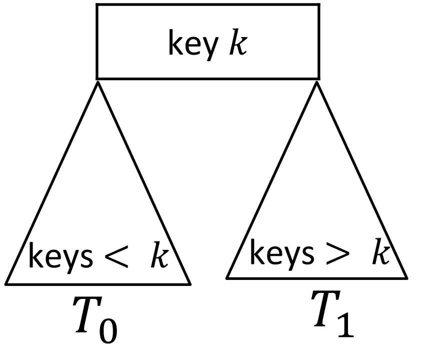

- AVL tree based dictionary implementations have poor *memory locality*
  - 'nearby' tree nodes are unlikely to be in the same block

- In an AVL tree $\Theta(\log n)$ blocks are loaded in the worst case
- Idea: allow trees that have many children per node
- Many children per node $\Longrightarrow$ smaller height $\Longrightarrow$ fewer block transfers
  - suppose store $n = 2^{50}$ items total, and $B = 2^{15}$ children per node
  - tree height is $\log_B n \;= \dfrac{\log_2 n}{\log_2 B} = \dfrac{50}{15}$
  - 15 times less block transfers
- First consider a special case: *2-4 trees*
  - 2-4 trees also used for dictionaries in internal memory
  - may be even faster than AVL-trees

# Outline

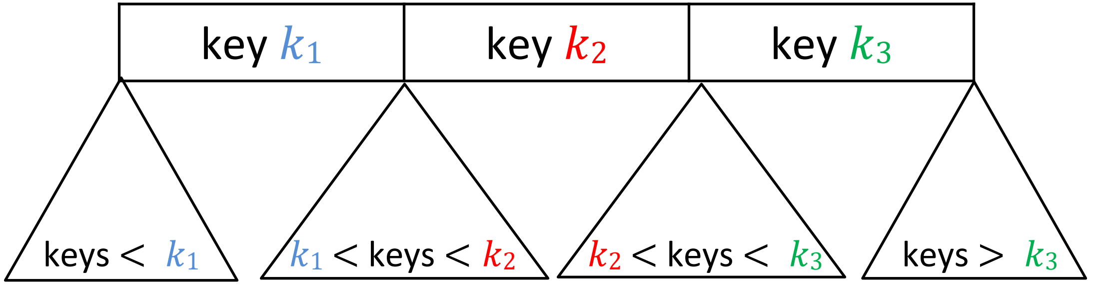
- External Memory
  - Motivation
  - Stream based algorithms
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

# 2-4 Trees Motivation



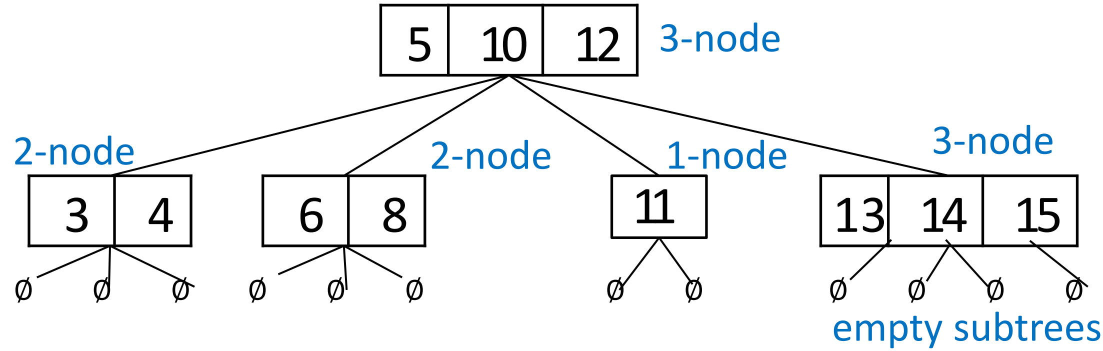

- Binary Search Tree supports efficient search with special key ordering

- Need nodes that store more than one key
    - how to support efficient search?

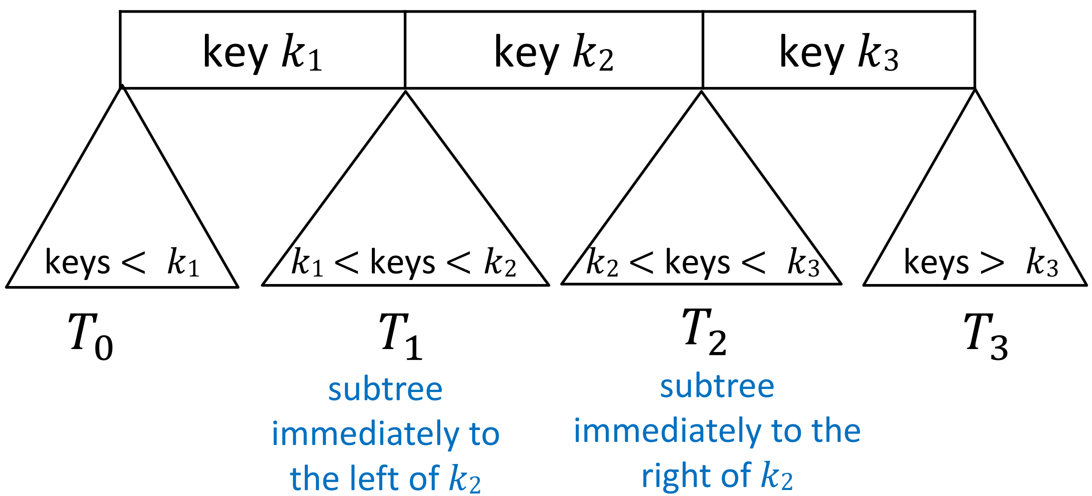



- Need additional properties to ensure tree is balanced and therefore *insert*, *delete* are efficient

# 2-4 Trees



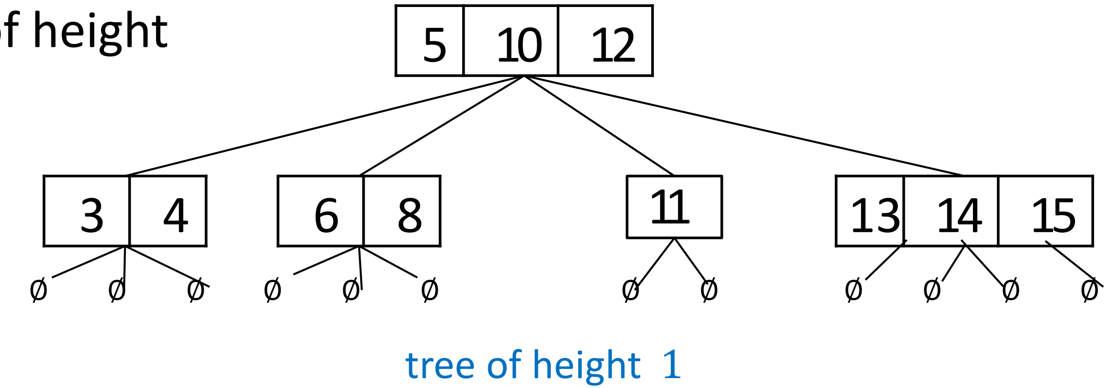

- **Structural properties**
  - Every node is either
    - 1-node: *one KVP* and *two subtrees* (possibly empty), or
    - 2-node: *two KVPs* and *three subtrees* (possibly empty), or
    - 3-node: *three KVPs* and *four subtrees* (possibly empty)
    - allowing 3 types of nodes simplifies insertion/deletion
  - All empty subtrees are at the same level
    - necessary for ensuring height is logarithmic in the number of KVP stored
- **Order property:** keys at any node are between the keys in the subtrees



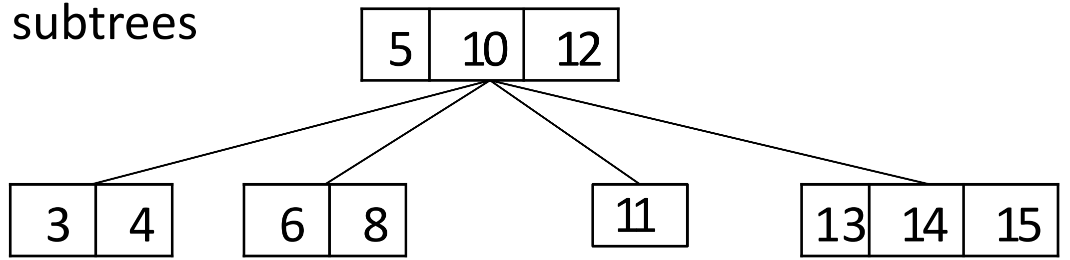

key-subtree list of the node
$< T_0, k_1, T_1, k_2, T_2, k_3, T_3 >$

# 2-4 Tree Example

- Empty subtrees are not part of height computation



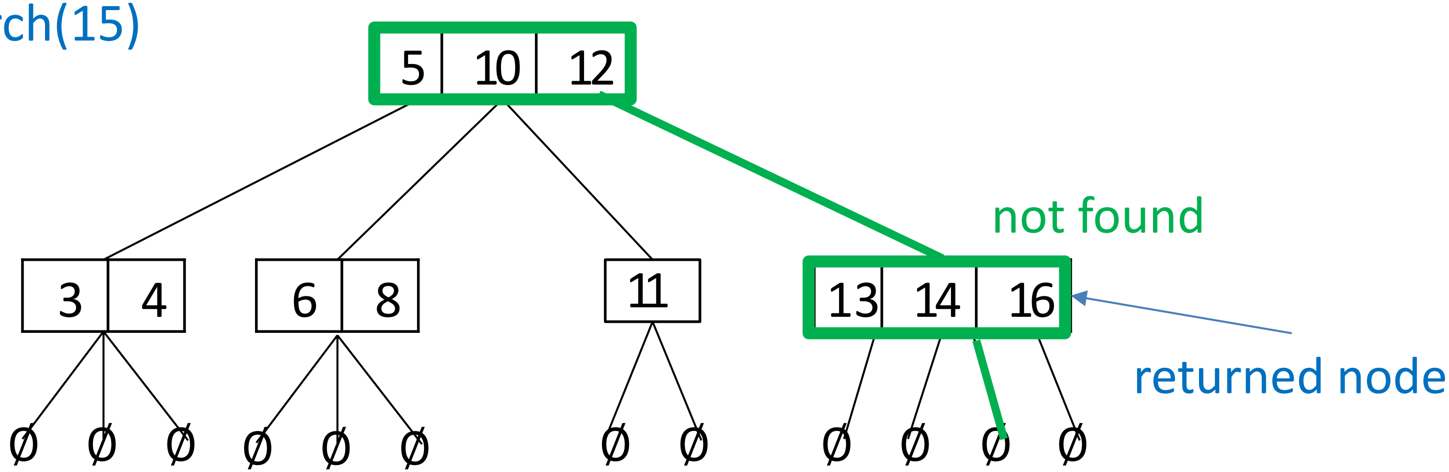

- Often do not even show empty subtrees




- Will prove height is $O(\log n)$ later, when we talk about (a,b)-trees
  - 2-4 tree is a special type of (a,b)-tree

# 2-4 Tree: Search Example

- **Search**
  - similar to search in BST
  - search($k$) compares key $k$ to $k_1$, $k_2$ , $k_3$, and either finds $k$ among $k_1$, $k_2$ , $k_3$ or figures out which subtree to recurse into
  - if key is not in tree, search returns parent of empty tree where search stops
    - key can be inserted at that node
  - search(15)



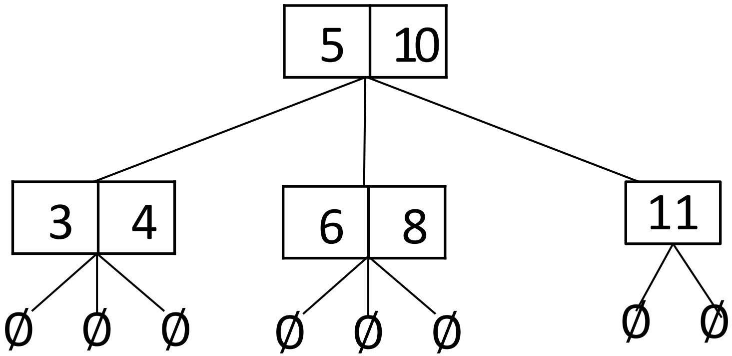
# 2-4 Tree operations

```
24Tree::search(k, v ←root, p ←empty subtree)
k: key to search, v: node where we search; p: parent of v
        if v represents empty subtree
                return "not found, would be in p"
        let < T_0, k_1, . . . , k_d , T_d > be key-subtrees list at v
        if k ≥ k_1
                i ← maximal index such that k_i ≤ k
                if k_i = k
                        return "at ith key in v "
                else 24Tree::search(k, T_i , v )
        else 24Tree::search(k, T_0, v )
```

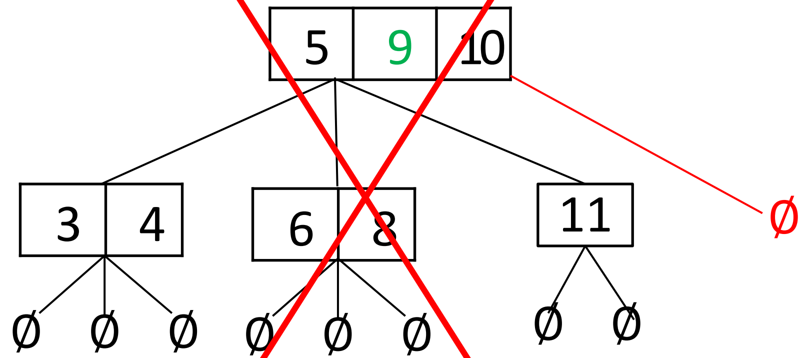
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(9)

node can hold one more item, so it's tempting to insert 9 in it



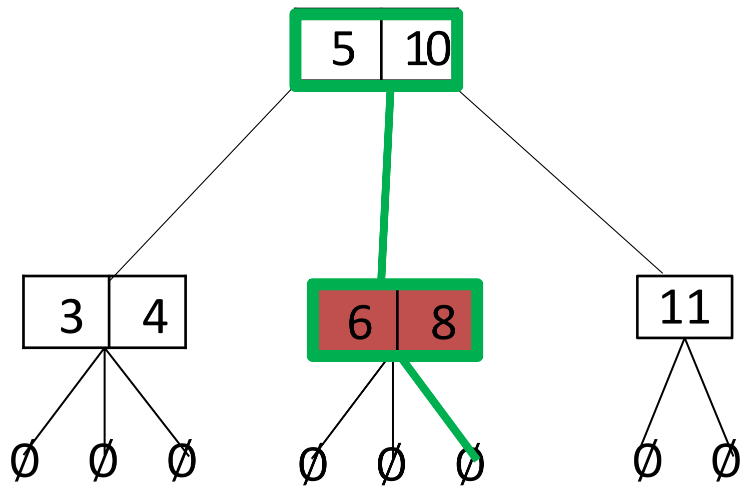

however, need 1 more subtree, since node has 3 keys now!




adding an empty subtree as the 4th subtree does not work, as all empty subtrees must be at the same level

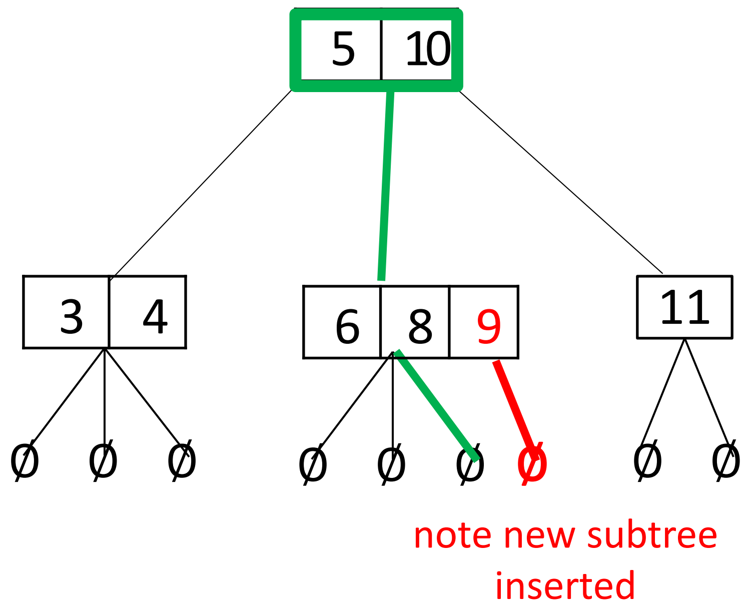
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(9)
  - first step: 24Tree::search(9)



# Example: 2-4 tree Insert

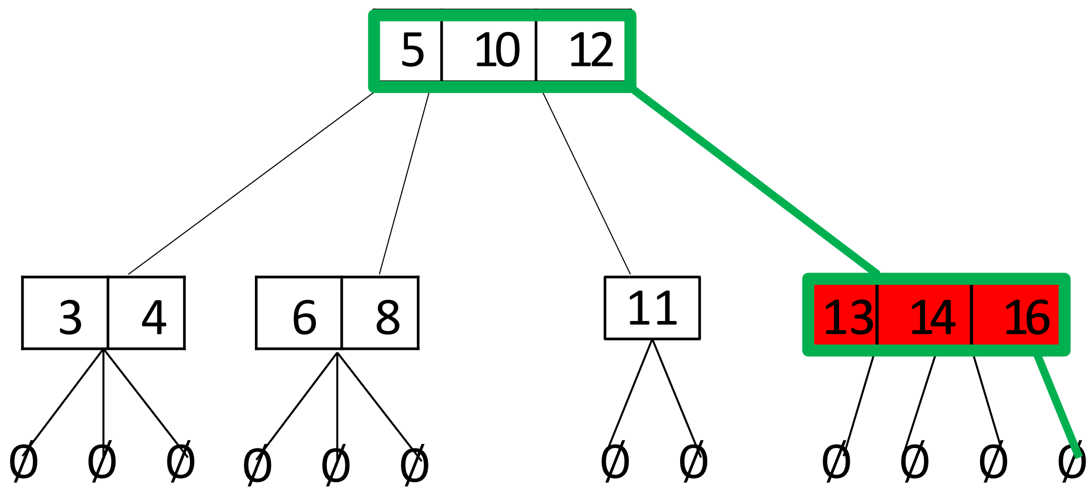
- **Example**: *24TreeInsert*(9)
  - first step: 24Tree::search(9)
  - second step: insert at the leaf node returned by search




  - adding an empty subtree at the last level causes no problems
  - order properties are preserved
  - node stays valid, it now has 3 KVPs, which is allowed

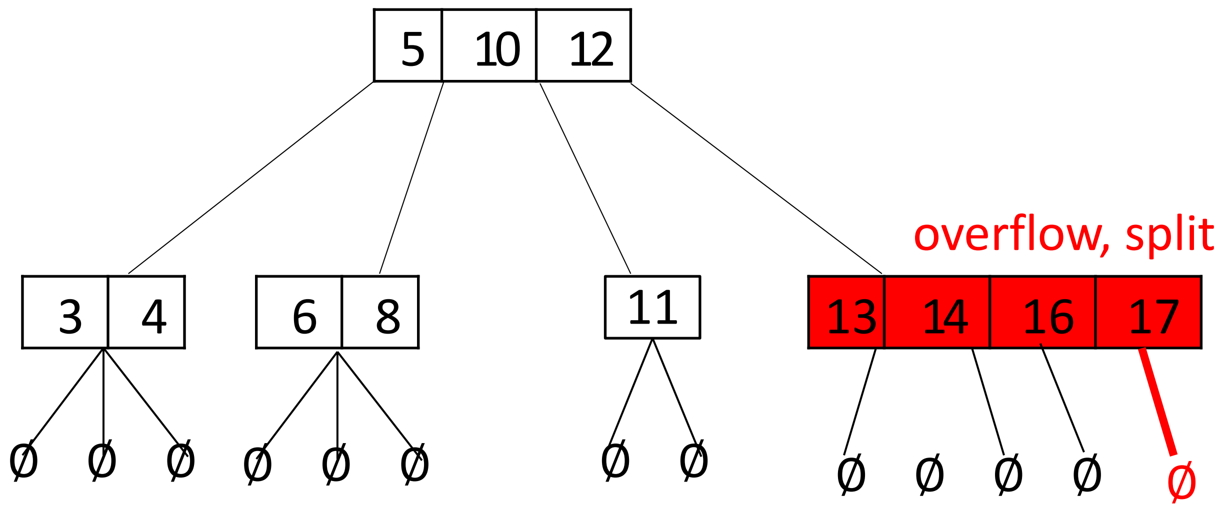
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
    - first step is 24Tree::search(17)
    - insert at the leaf node returned by search



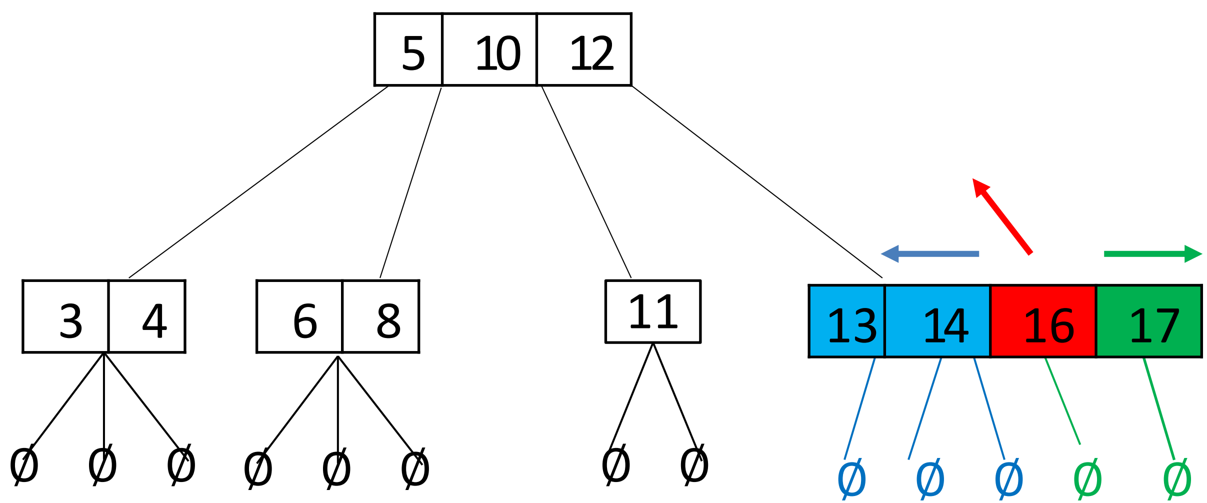
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
  - now leaf has 4 KVPs, not allowed, have to fix this



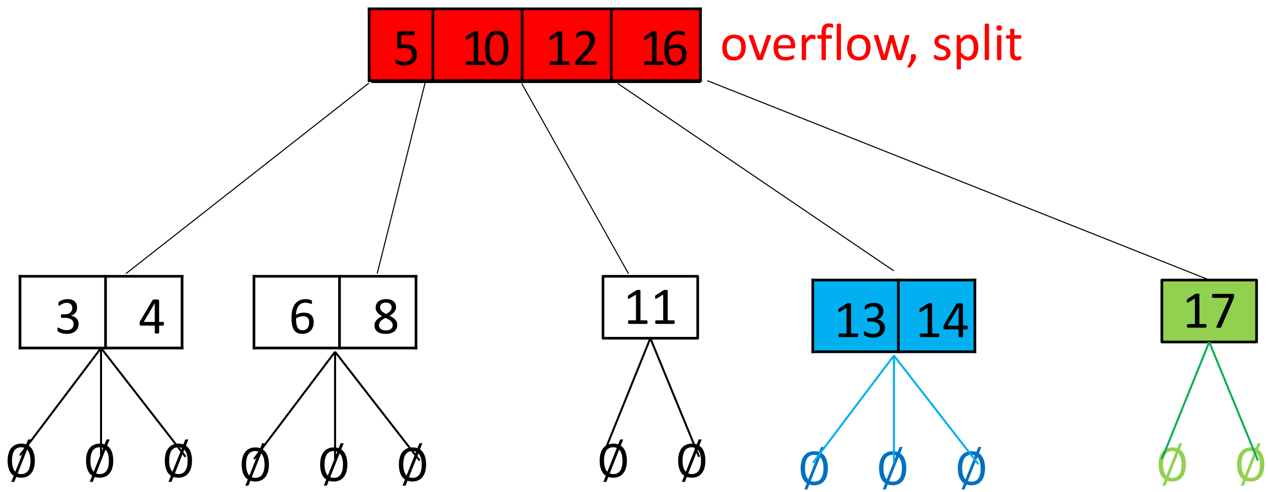
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
    - now leaf has 4 KVPs, not allowed, have to fix this



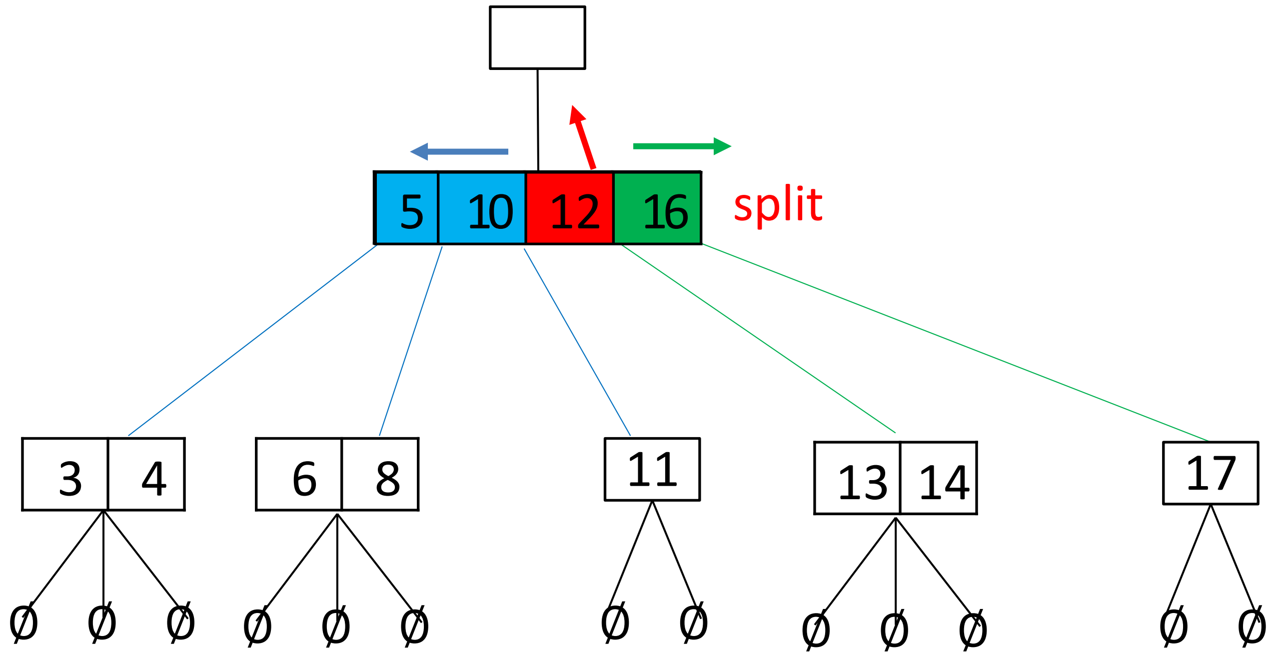
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
    - splitting is possible because we allow variable node size
        - split 3-node into 1-node and 2-node
    - order property is preserved after a split
    - overflow can propagate to the parent of split node



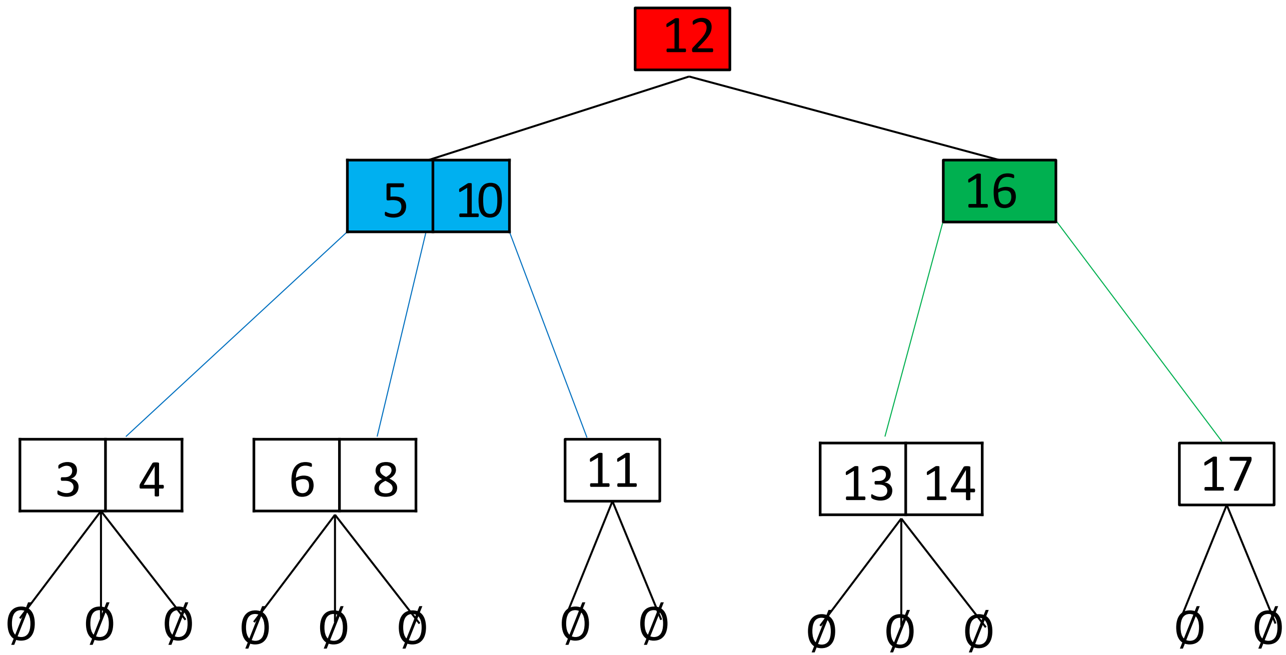
# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
  - when splitting the root node, need to create new root



# Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)



# 2-4 Tree Insert Pseudocode

*24Tree::insert*$(k)$

$v \leftarrow$ *24Tree::search*$(k)$ //leaf where *k* should be

add $k$ and an empty subtree in key-subtree-list of $v$

**while** $v$ has 4 keys (overflow → node split)

let $< T_0, k_1, \ldots, k_4, T_4 >$ be key-subtrees list at $v$

**if** $v$ has no parent

create an empty parent of $v$

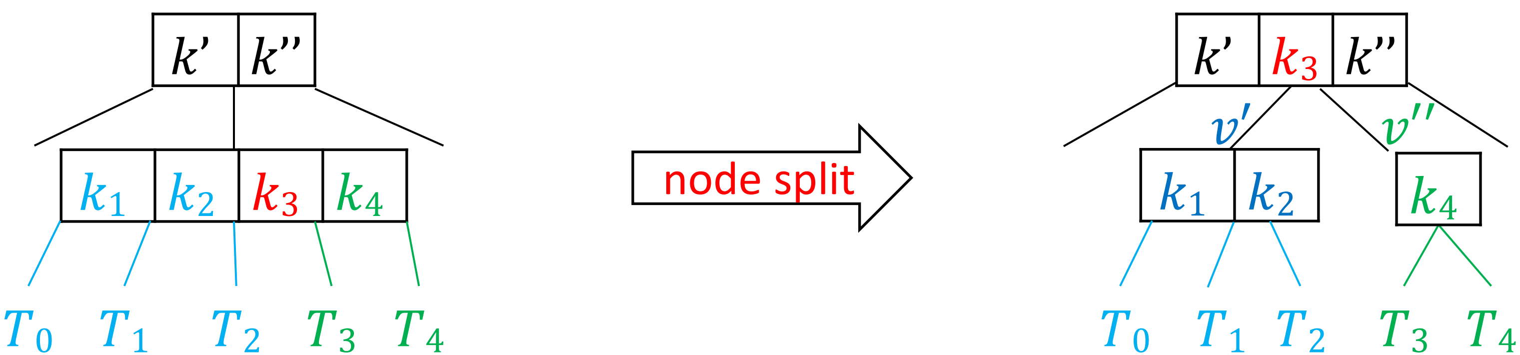
$p \leftarrow$ parent of $v$

$v' \leftarrow$ new node with keys $k_1, k_2$ and subtrees $T_0, T_1, T_2$

$v'' \leftarrow$ new node with key $k_4$ and subtrees $T_3, T_4$

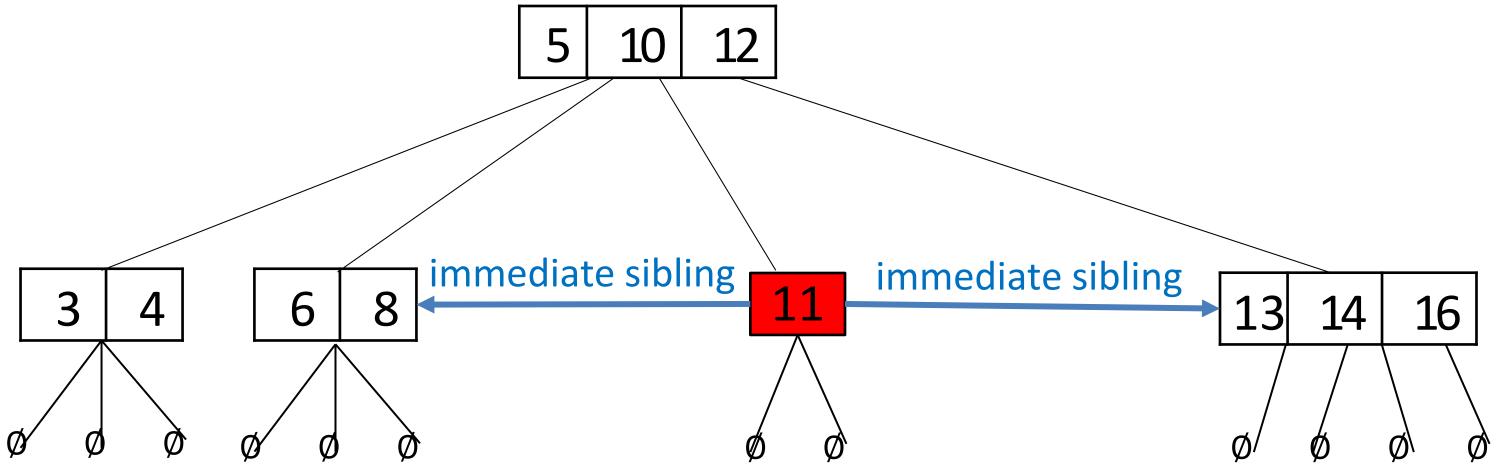
replace $< v >$ by $< v', k_3, v'' >$ in key-subtree-list of $p$

$v \leftarrow p$ //continue checking for overflow upwards

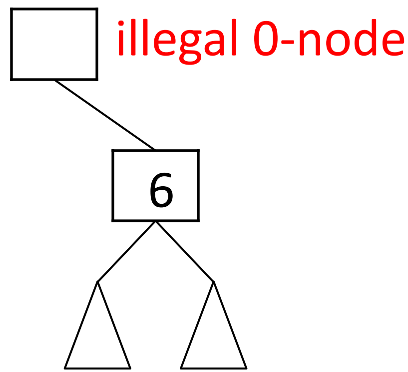


# 2-4 Tree: Immediate Sibling

- A node can have an *immediate* left sibling, immediate right sibling, or both

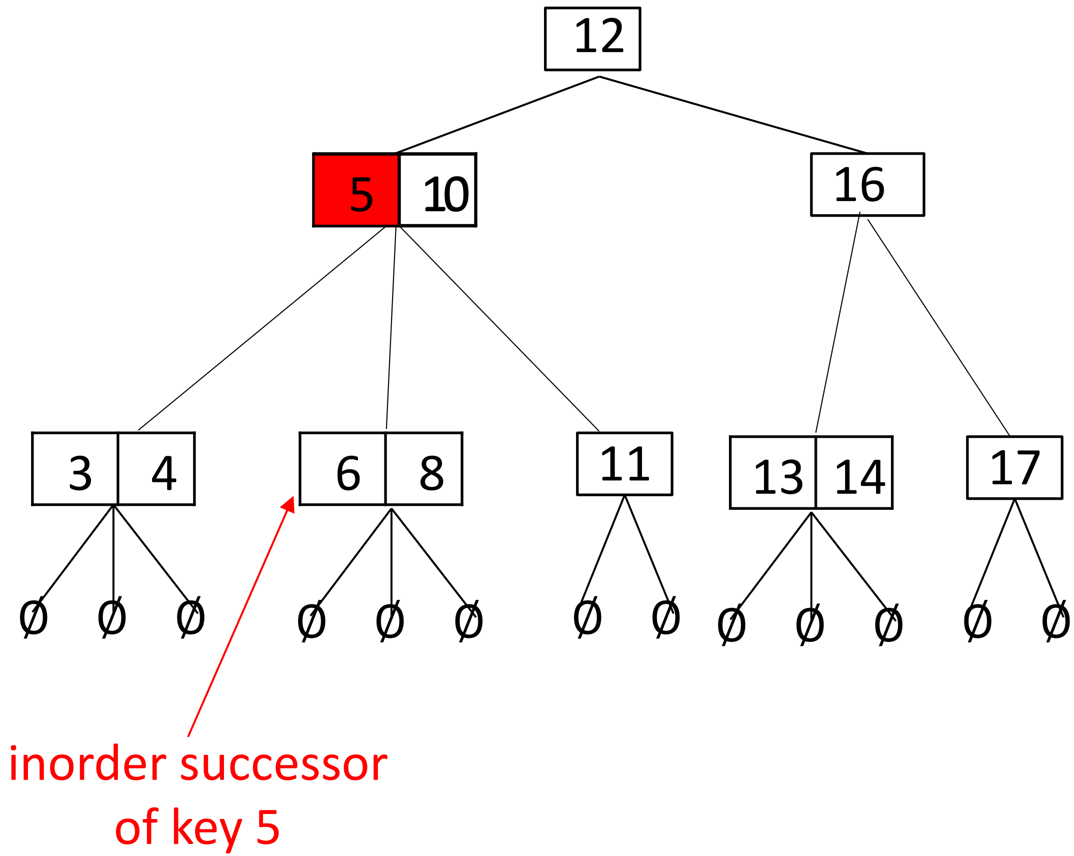



- Any node except the root must have an immediate sibling



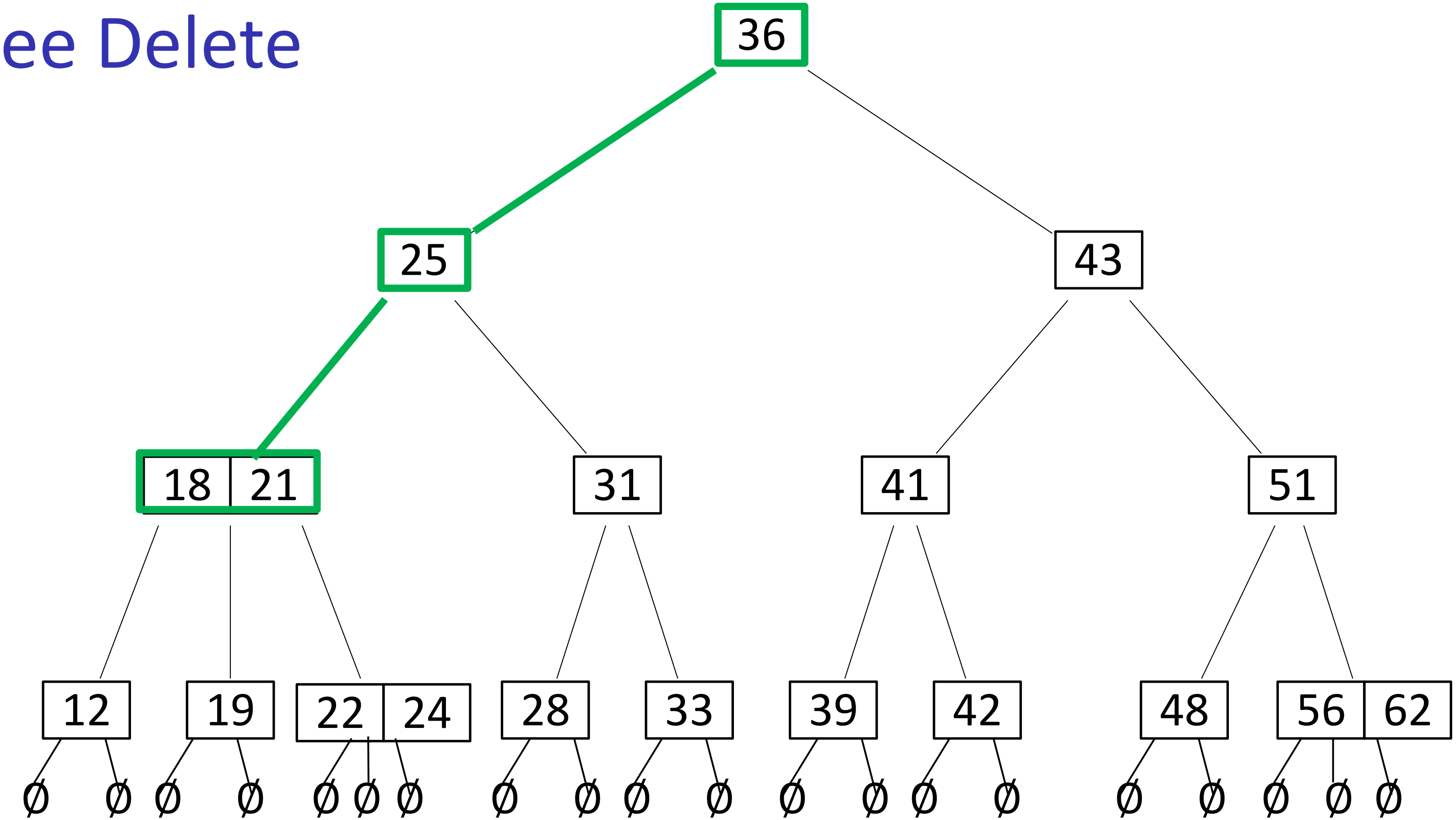
# 2-4 Tree: Inorder Successor

- Inorder successor of key $k$ is the smallest key in the subtree immediately to the right of $k$



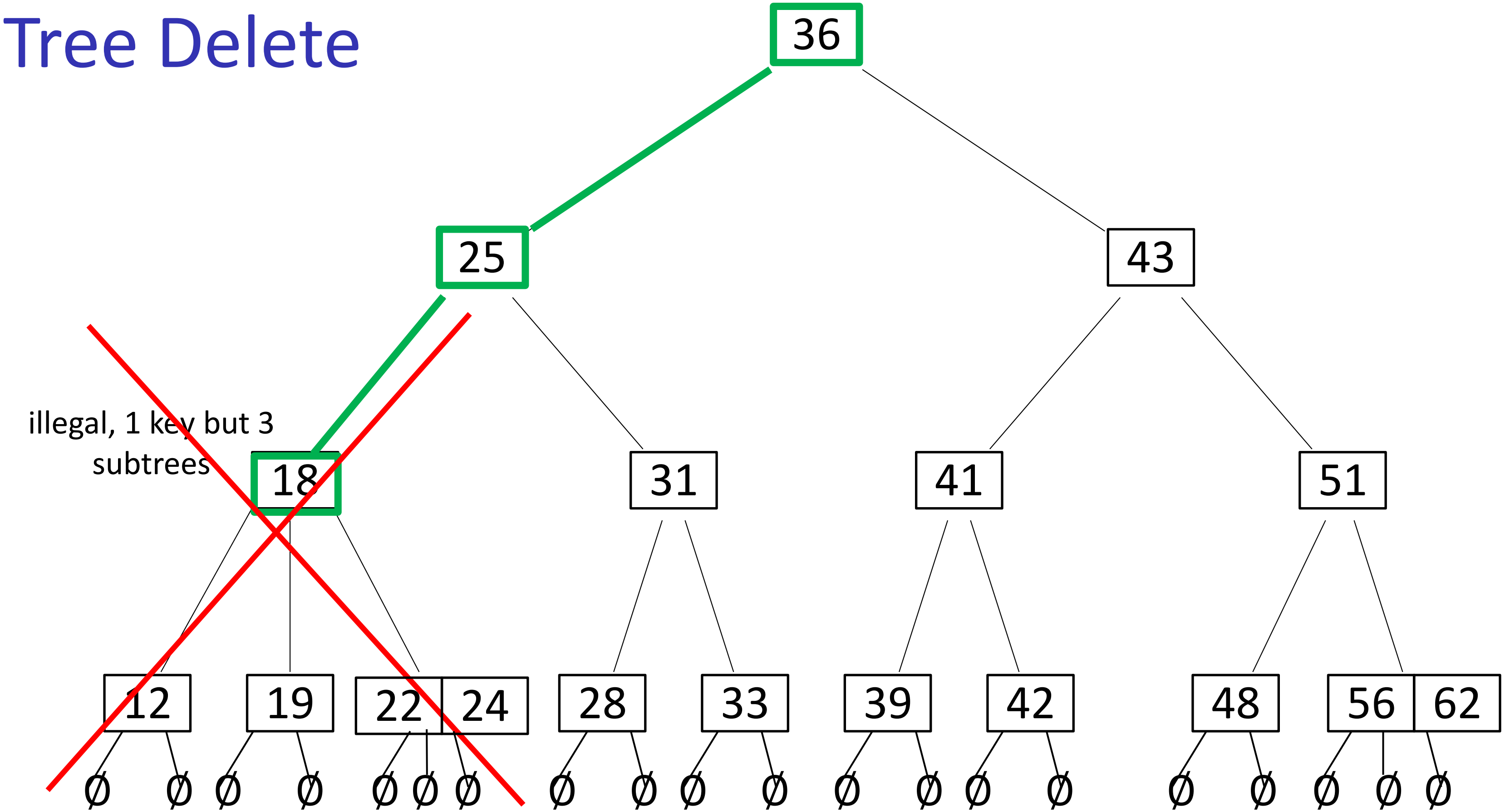

- Inorder successor is guaranteed to be at a leaf node
  - otherwise would have something smaller in the leftmost subtree

# 2-4 Tree Delete



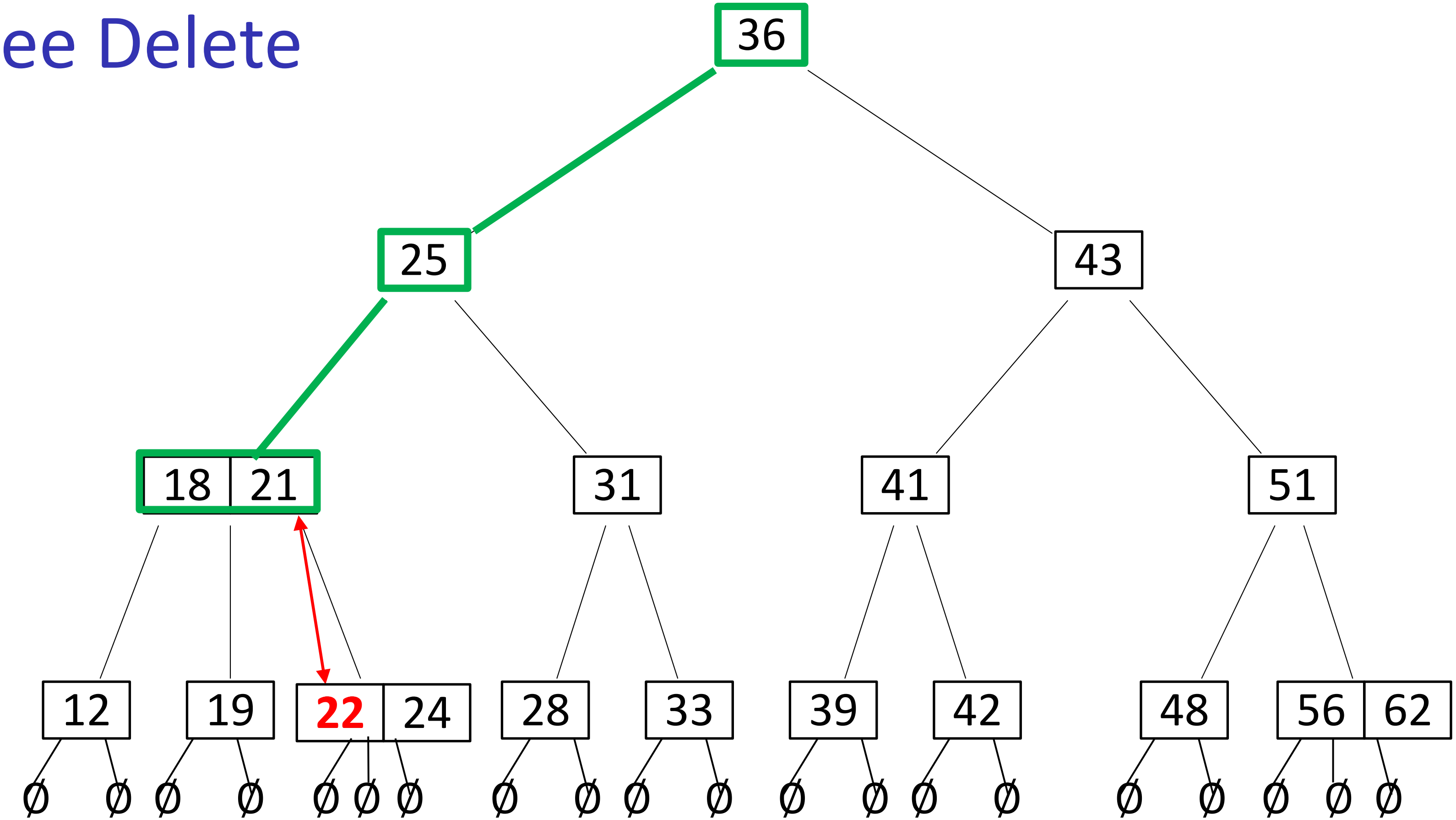

- Example: *delete*(21)
- Search for key to delete
    - if a node found has more than 1 key, it is tempting to delete it directly

# 2-4 Tree Delete



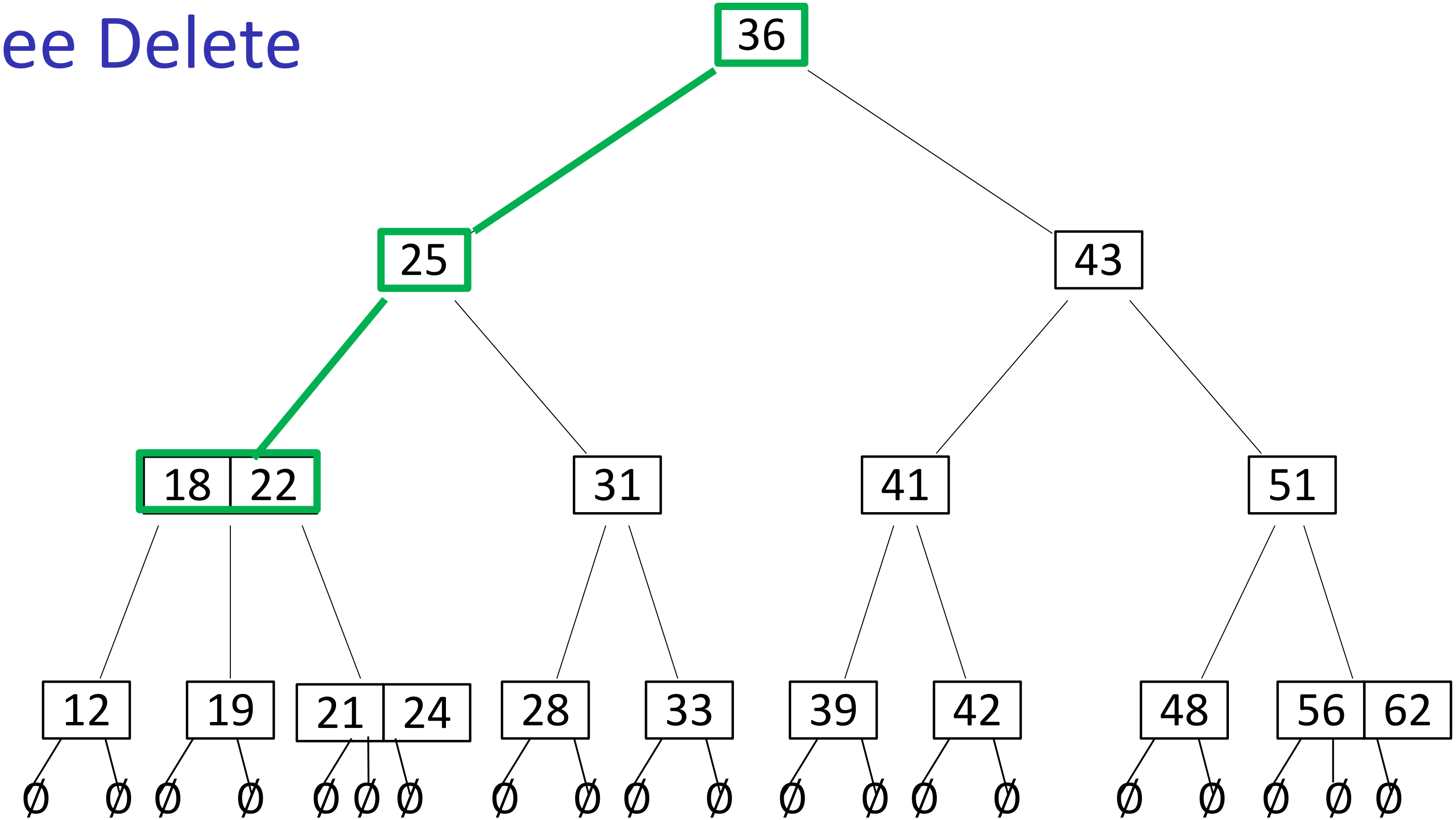

- Example: *delete*(21)
- Search for key to delete
  - if a node found has more than 1 key, it is tempting to delete it directly
  - however, can delete the key directly only if a node is a leaf
    - when we delete a key, we need to delete 1 subtree, easy only at a leaf

# 2-4 Tree Delete



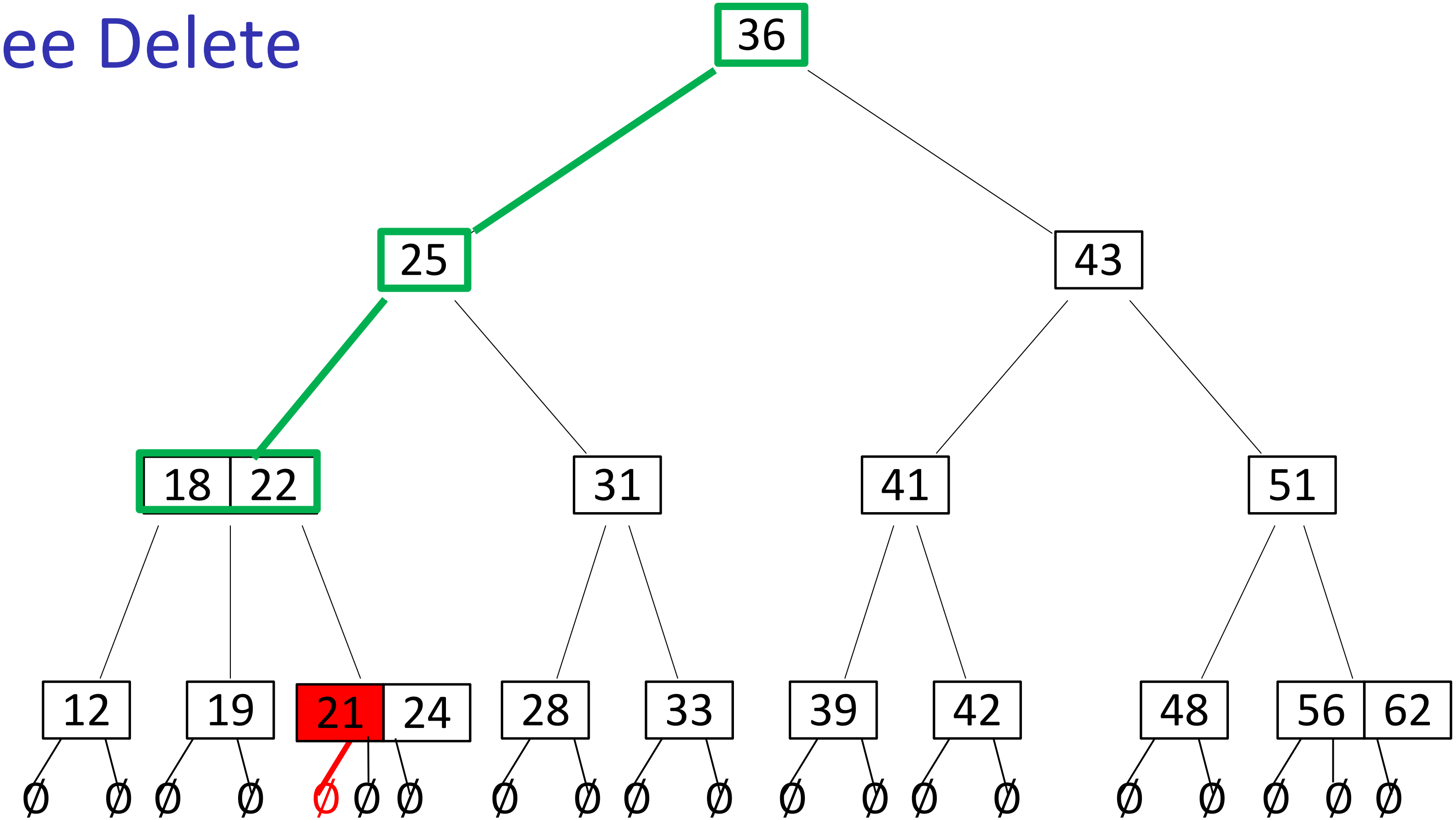

- Example: *delete*(21)
- Search for key to delete
  - can delete keys only from a leaf node, as need to delete a subtree as well
  - if the key is in a node which is not a leaf, replace key with its inorder successor

# 2-4 Tree Delete



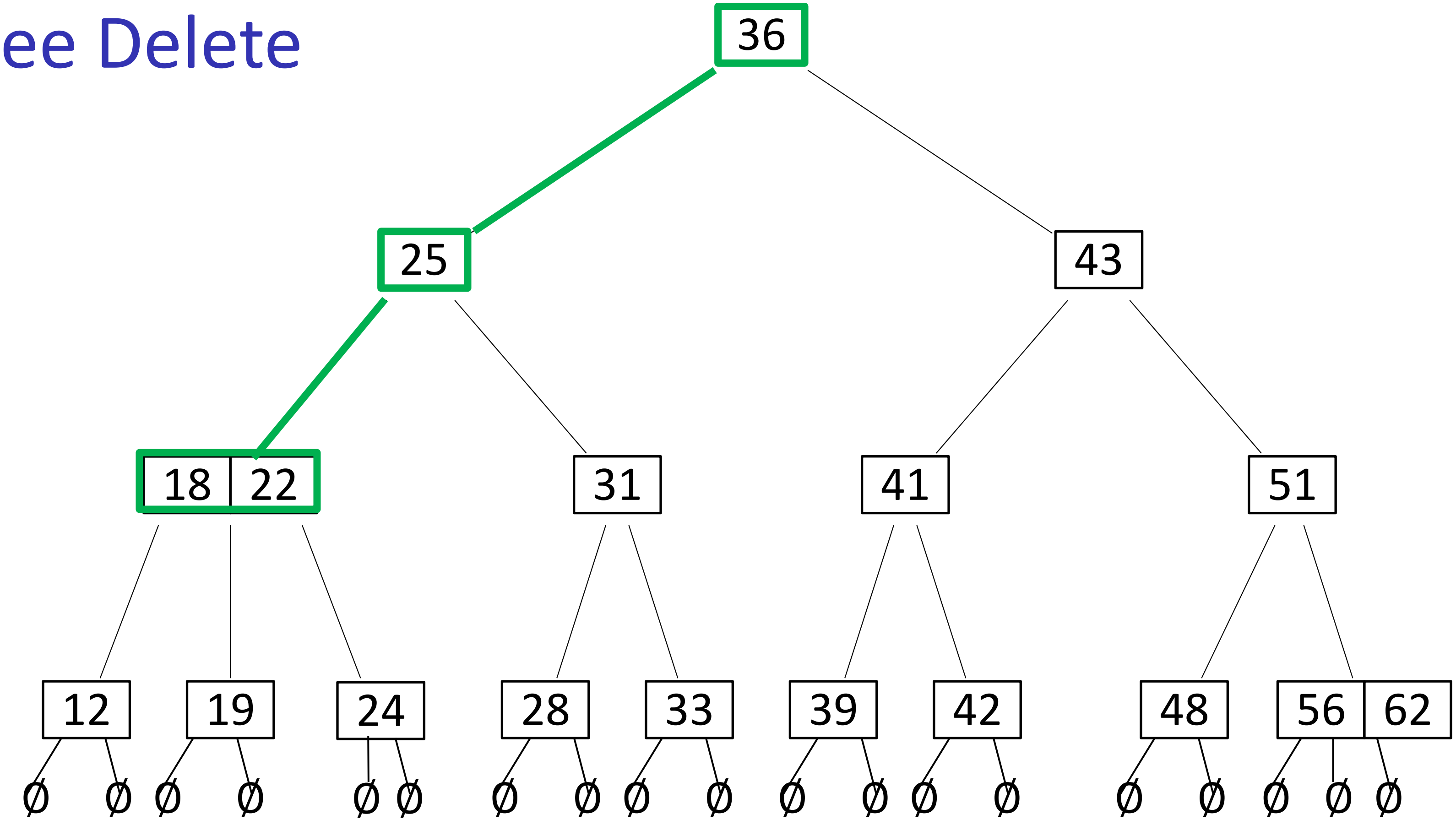

- Example: *delete*(21)
- Search for key to delete
    - can delete keys only from a leaf node, as need to delete a subtree as well
    - if the key is in a node which is not a leaf, replace key with its inorder successor

# 2-4 Tree Delete



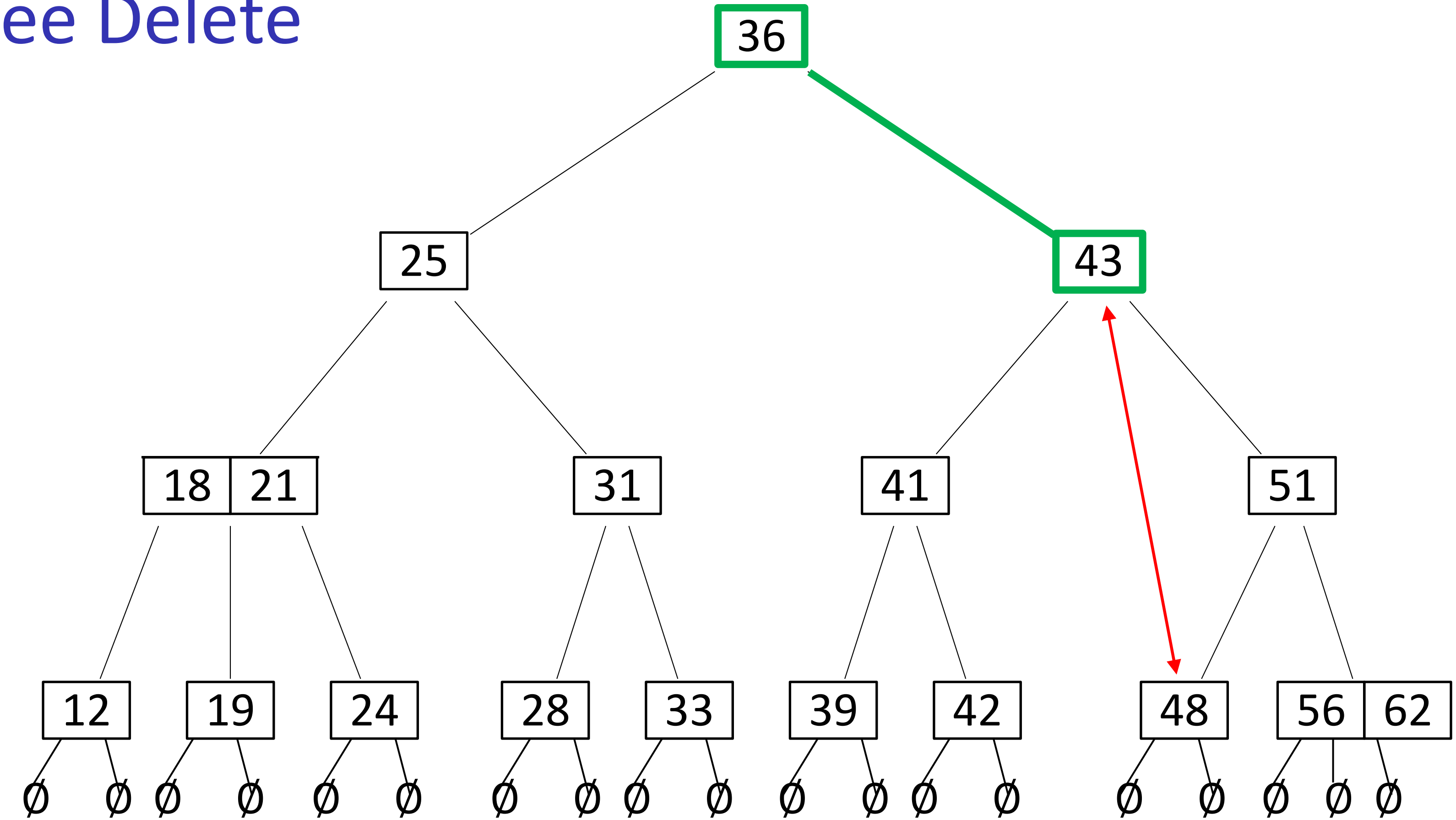

- Example: *delete*(21)
- Search for key to delete
  - can delete keys only from a leaf node, as need to delete a subtree as well
  - if the key is in a node which is not a leaf, replace key with its inorder successor
  - delete key 21 and an empty subtree

# 2-4 Tree Delete



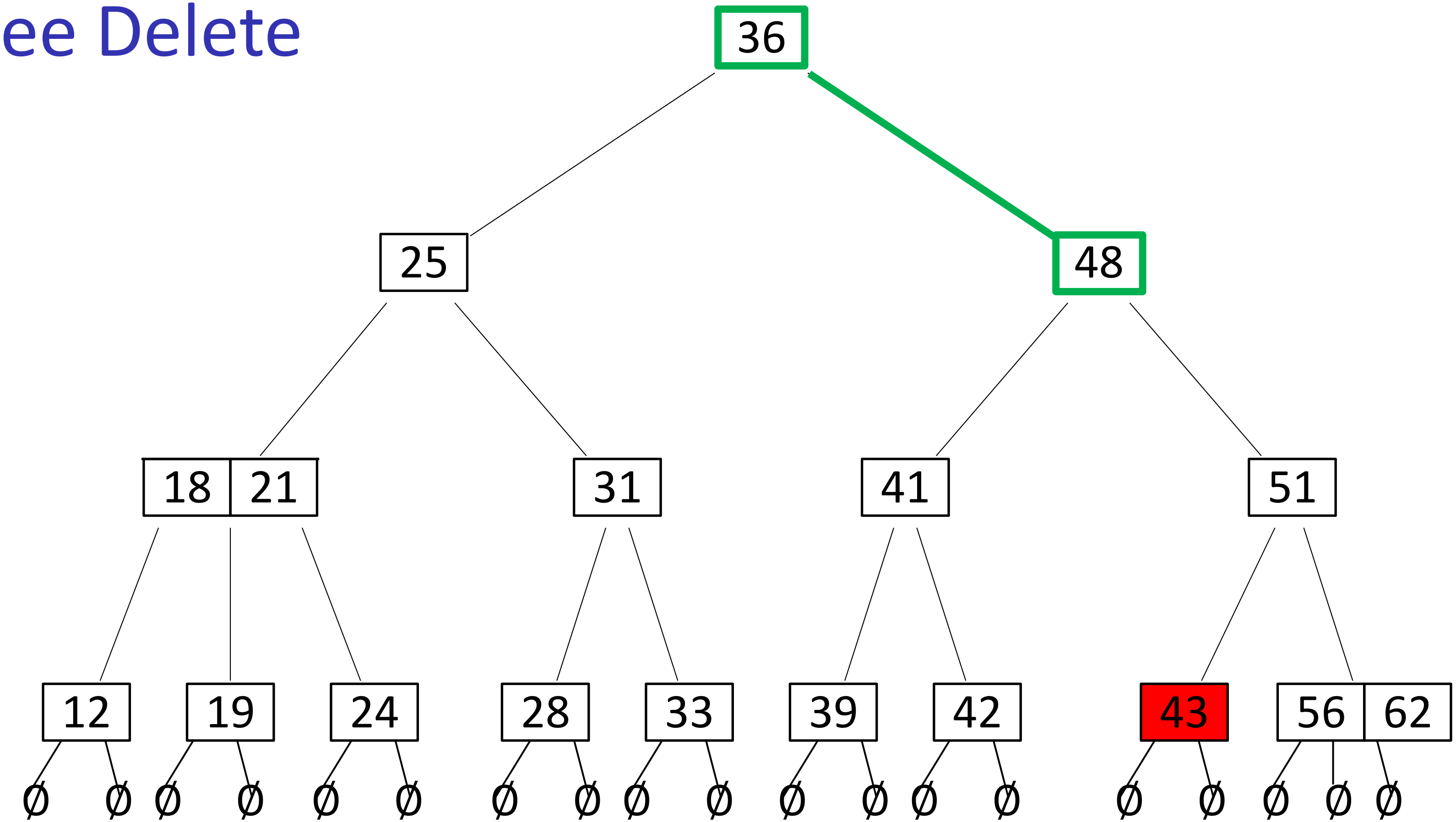

- Example: *delete*(21)
- Search for key to delete
  - can delete keys only from a leaf node, as need to delete a subtree as well
  - if the key is in a node which is not a leaf, replace key with its inorder successor
  - delete key 21 and an empty subtree
  - order property is preserved and we are done

# 2-4 Tree Delete



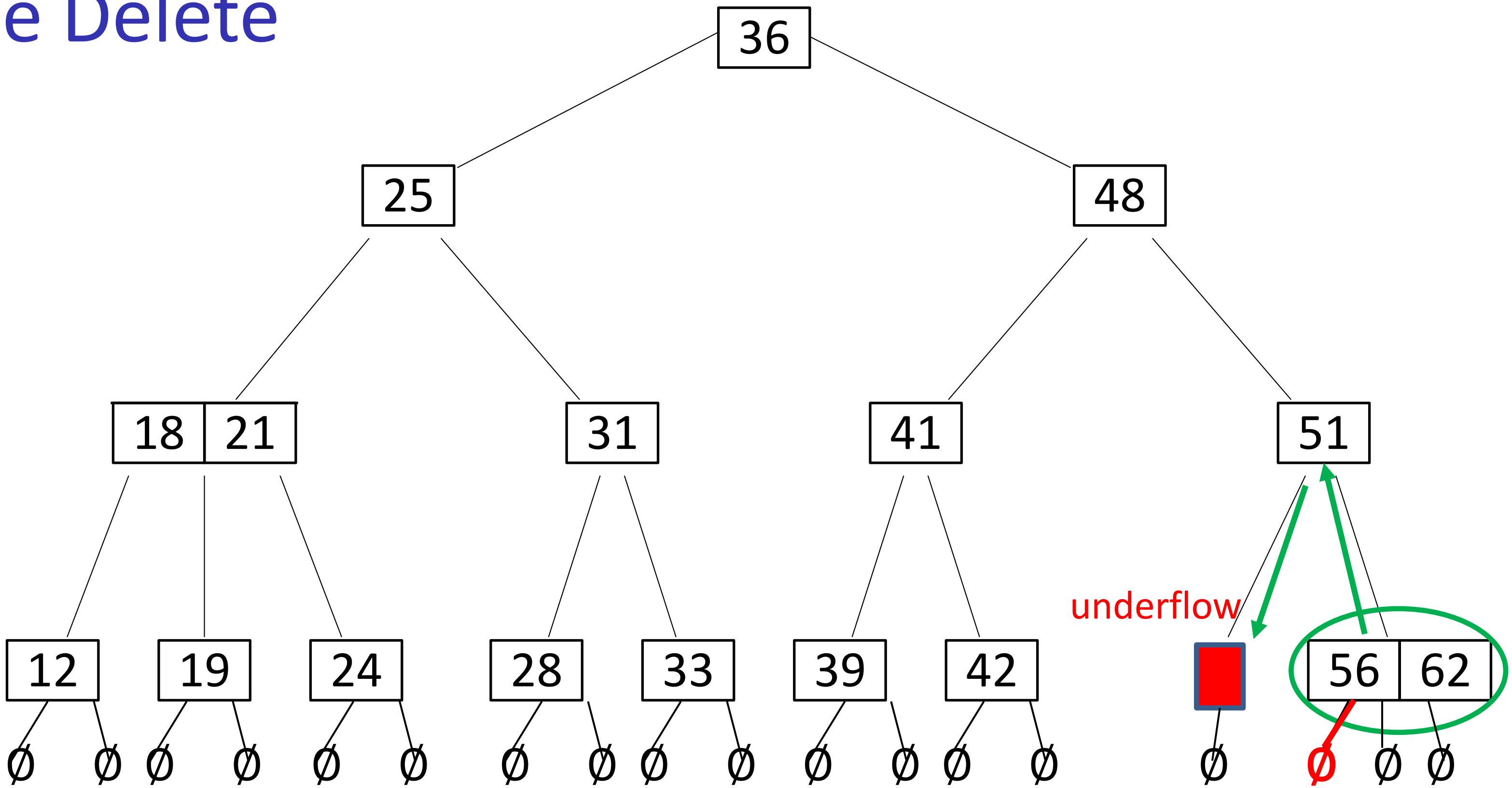

- Example: *delete*(43)
- Search for key to delete
    - can delete keys only from a leaf node
    - replace key with in-order successor

# 2-4 Tree Delete



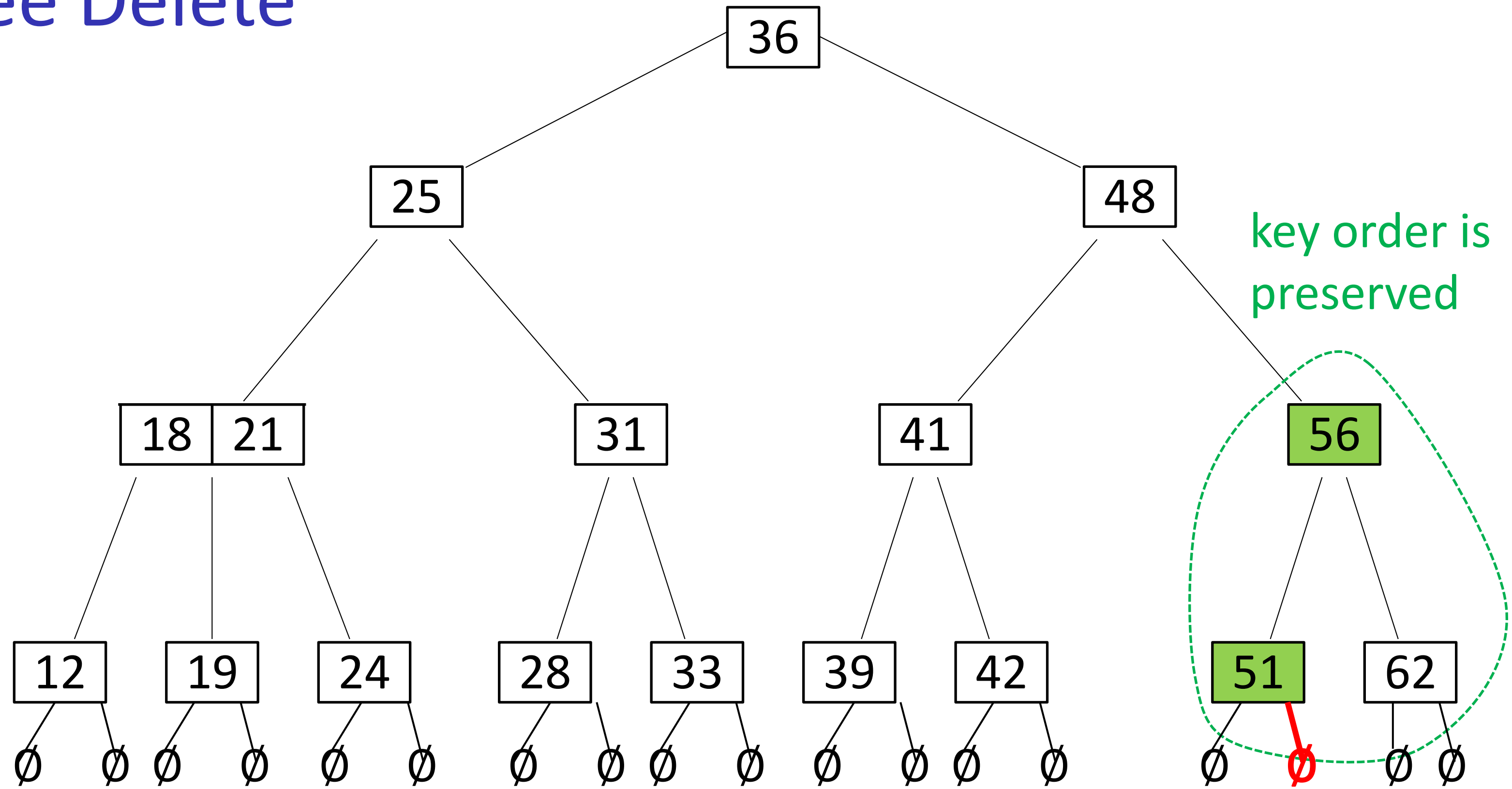

- Example: *delete*(43)
- Search for key to delete
  - can delete keys only from a leaf node
  - replace key with in-order successor
  - delete key 43 and a subtree

# 2-4 Tree Delete



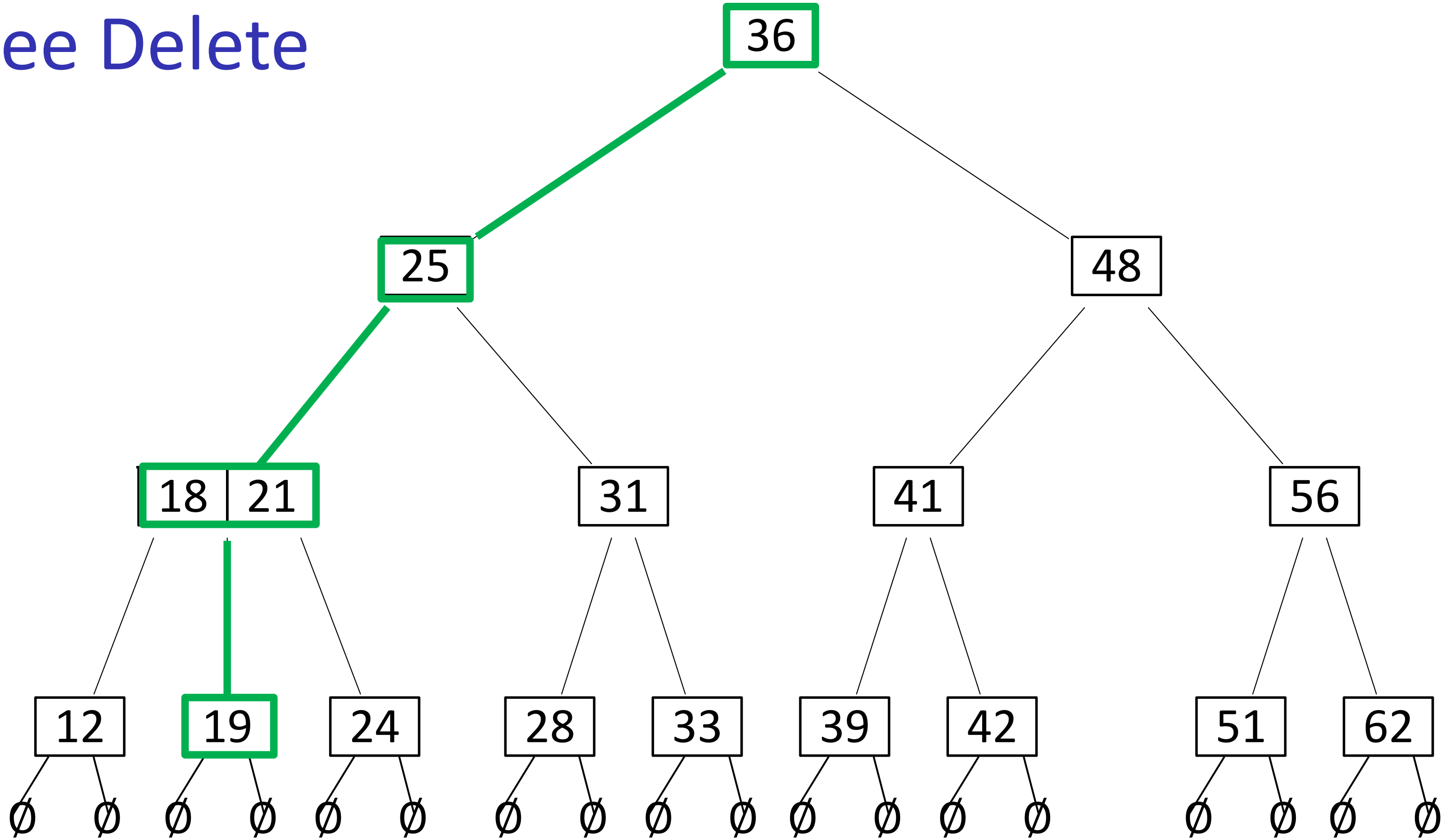

- Example: *delete*(43)
  - *rich* immediate sibling, transfer key from sibling, with help from the parent
    - sibling is *rich* if it is a 2-node or 3-node
    - adjacent subtree from sibling is also transferred

# 2-4 Tree Delete



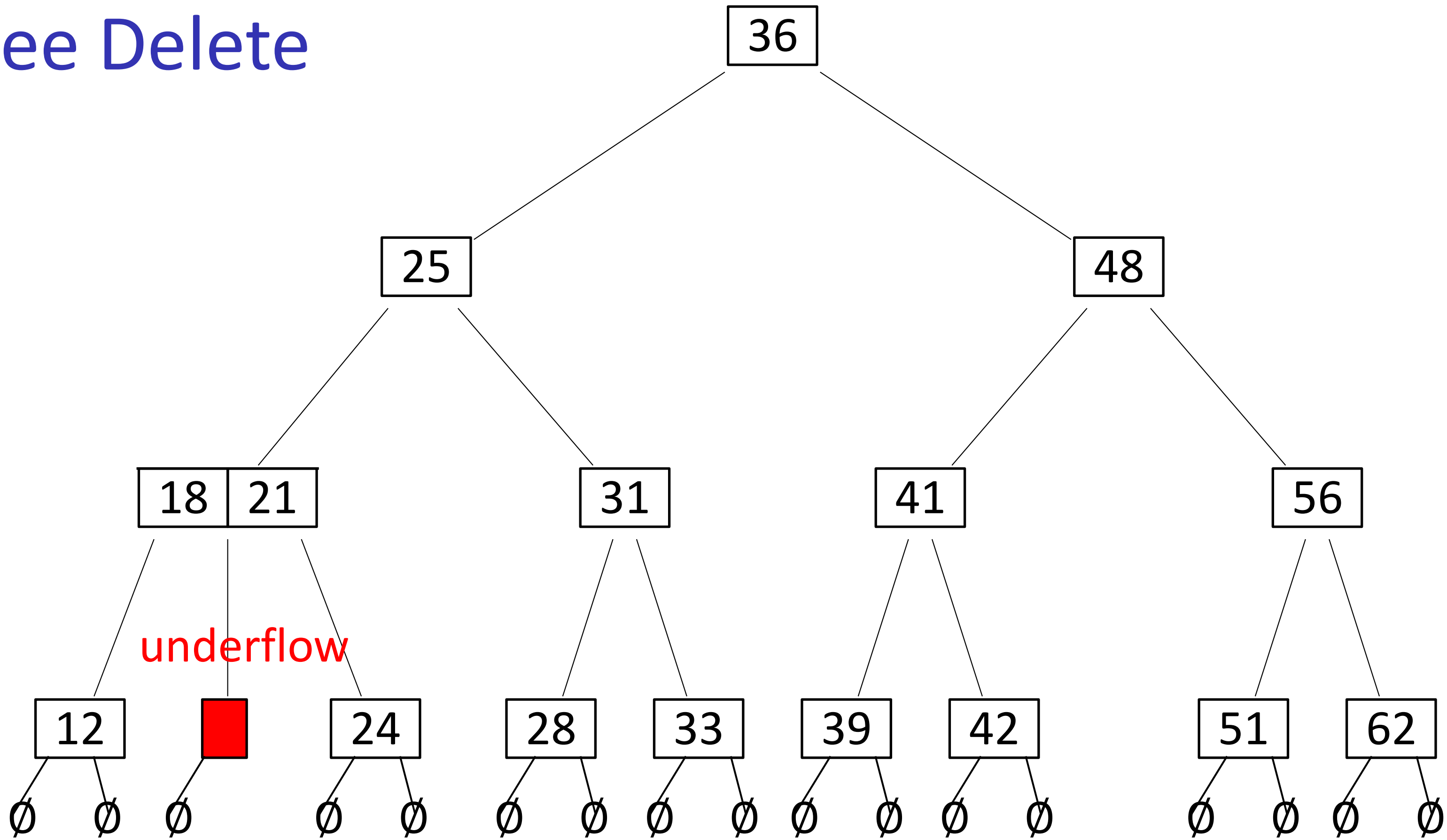

- Example: *delete*(43)
  - *rich* immediate sibling, transfer key from sibling, with help from the parent
    - sibling is *rich* if it is a 2-node or 3-node
    - adjacent subtree from sibling is also transferred
    - order property is preserved

# 2-4 Tree Delete



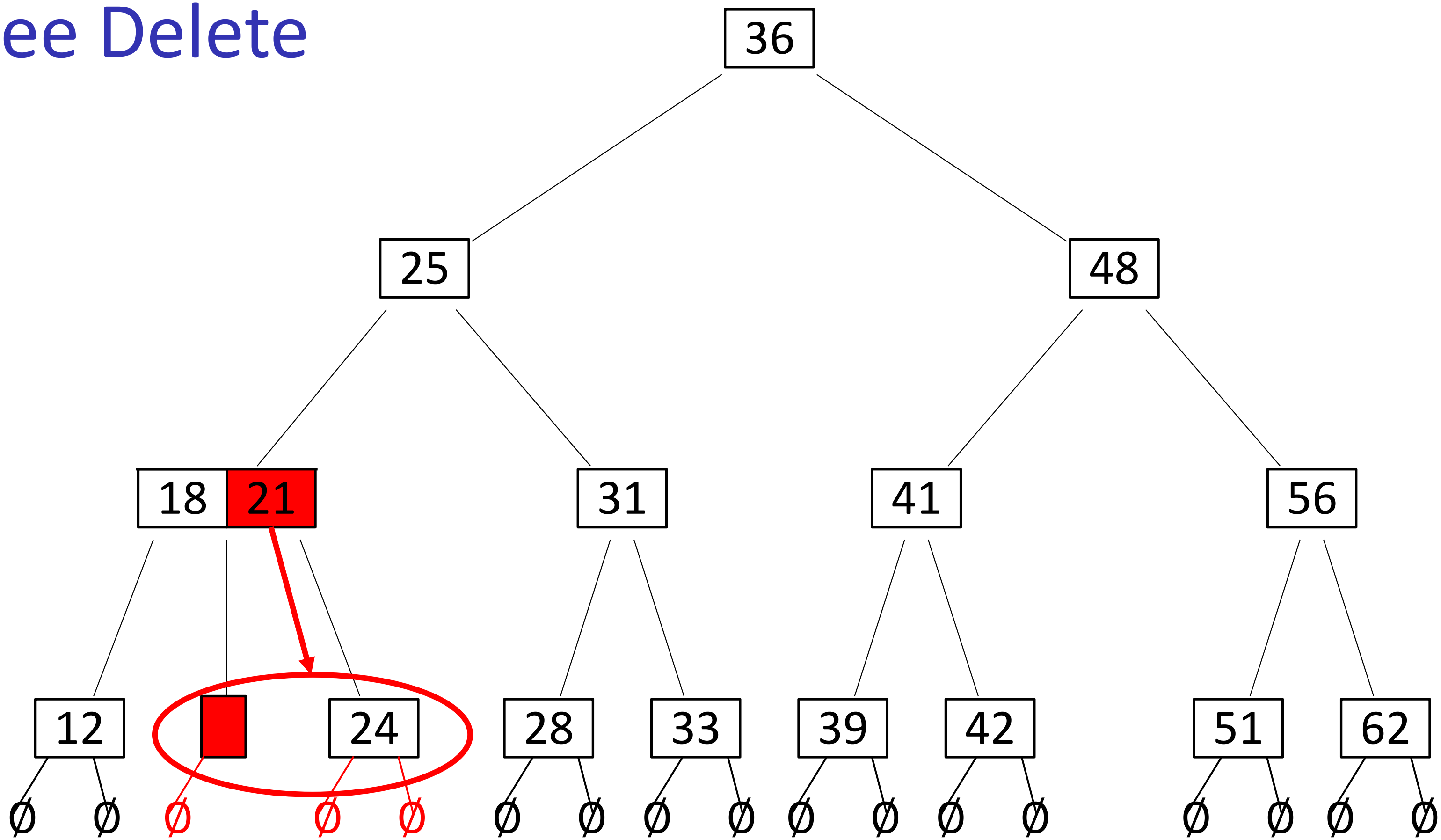

- Example: *delete*(19)
  - first search(19)

# 2-4 Tree Delete



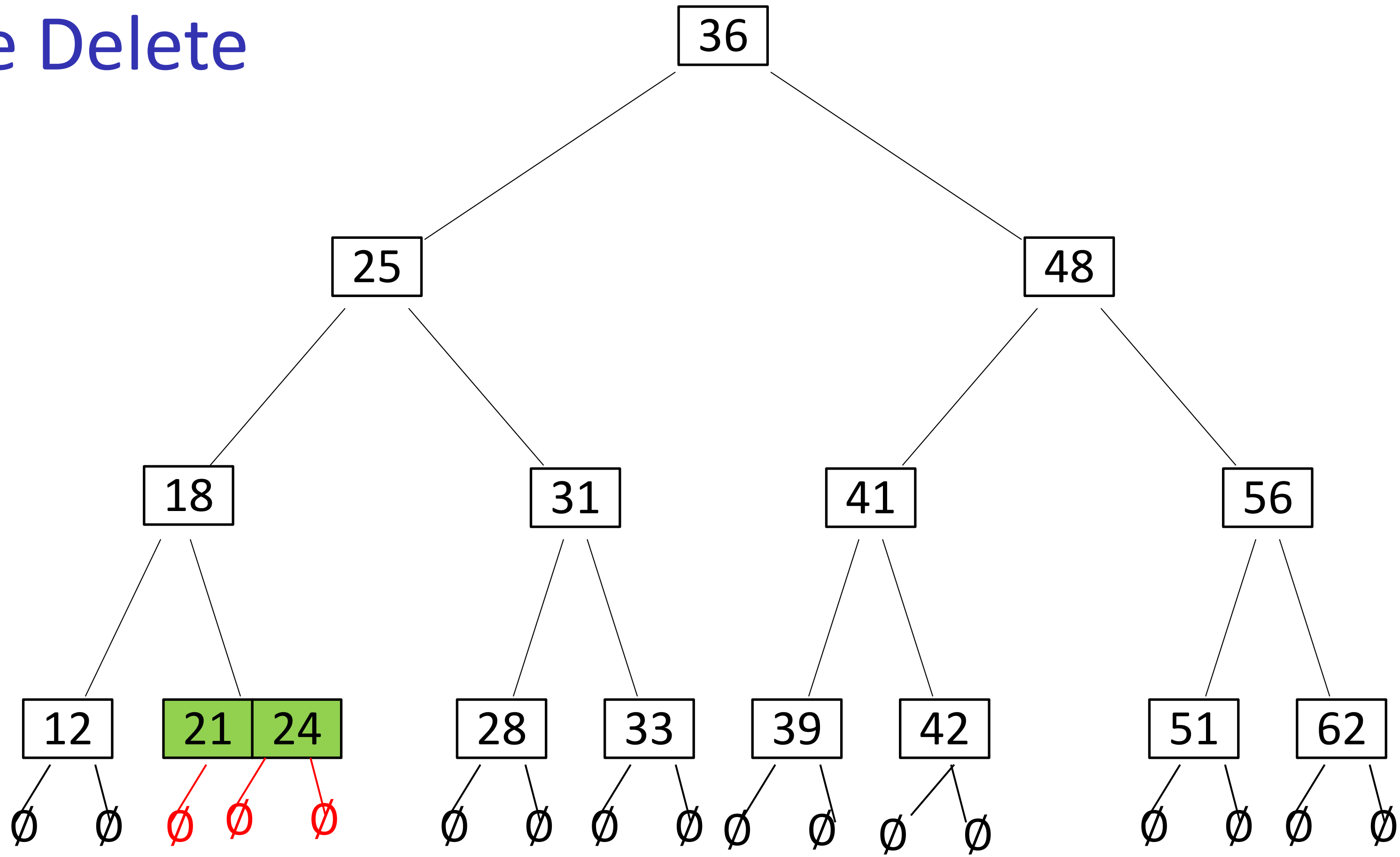

- Example: *delete*(19)
  - first search(19)
  - then delete key 19 (and an empty subtree) from the node
  - immediate siblings exist, but not rich, cannot transfer

# 2-4 Tree Delete



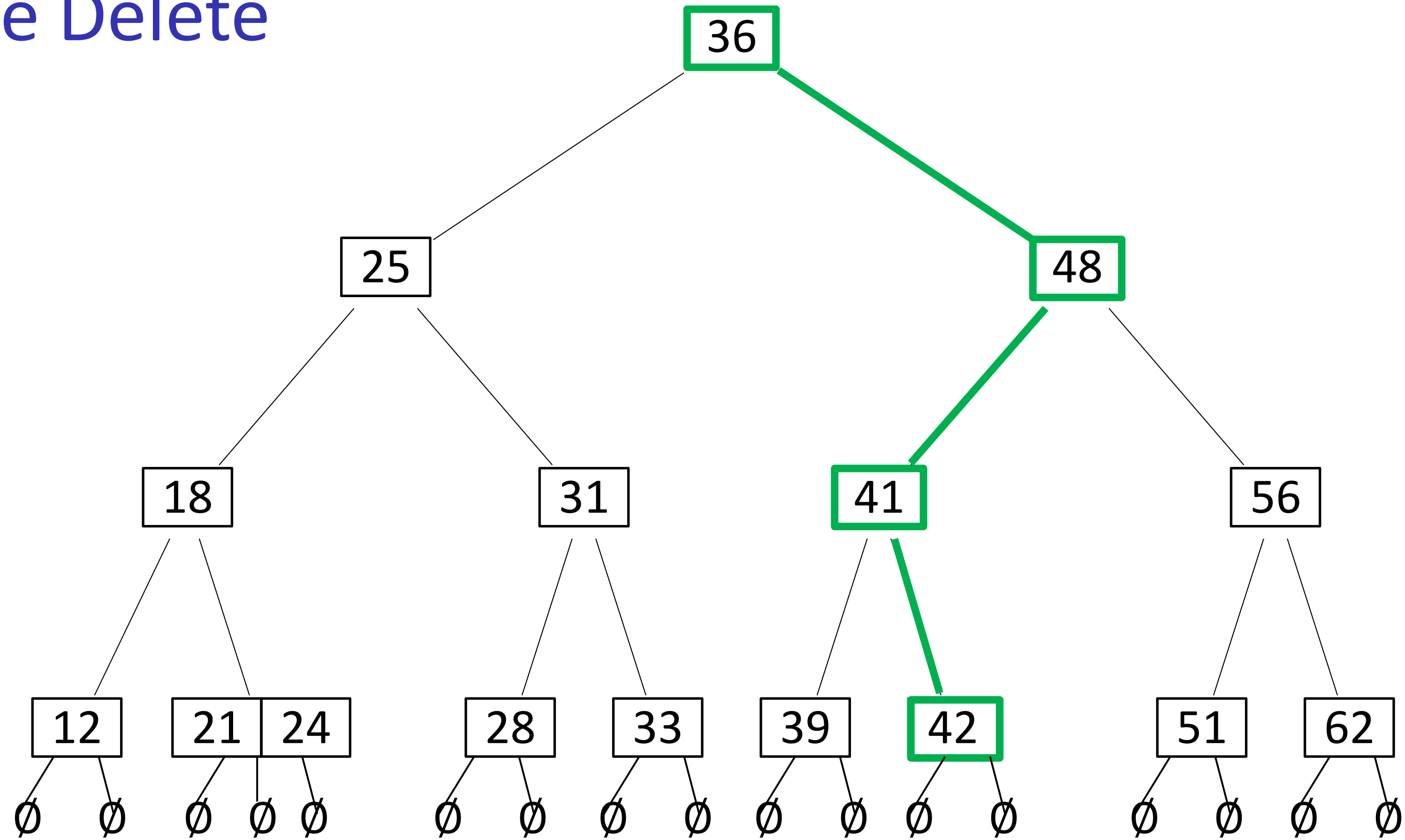

- Example: *delete*(19)
  - immediate siblings exist, but not rich, cannot transfer
    - merge with right immediate sibling with help from parent

# 2-4 Tree Delete



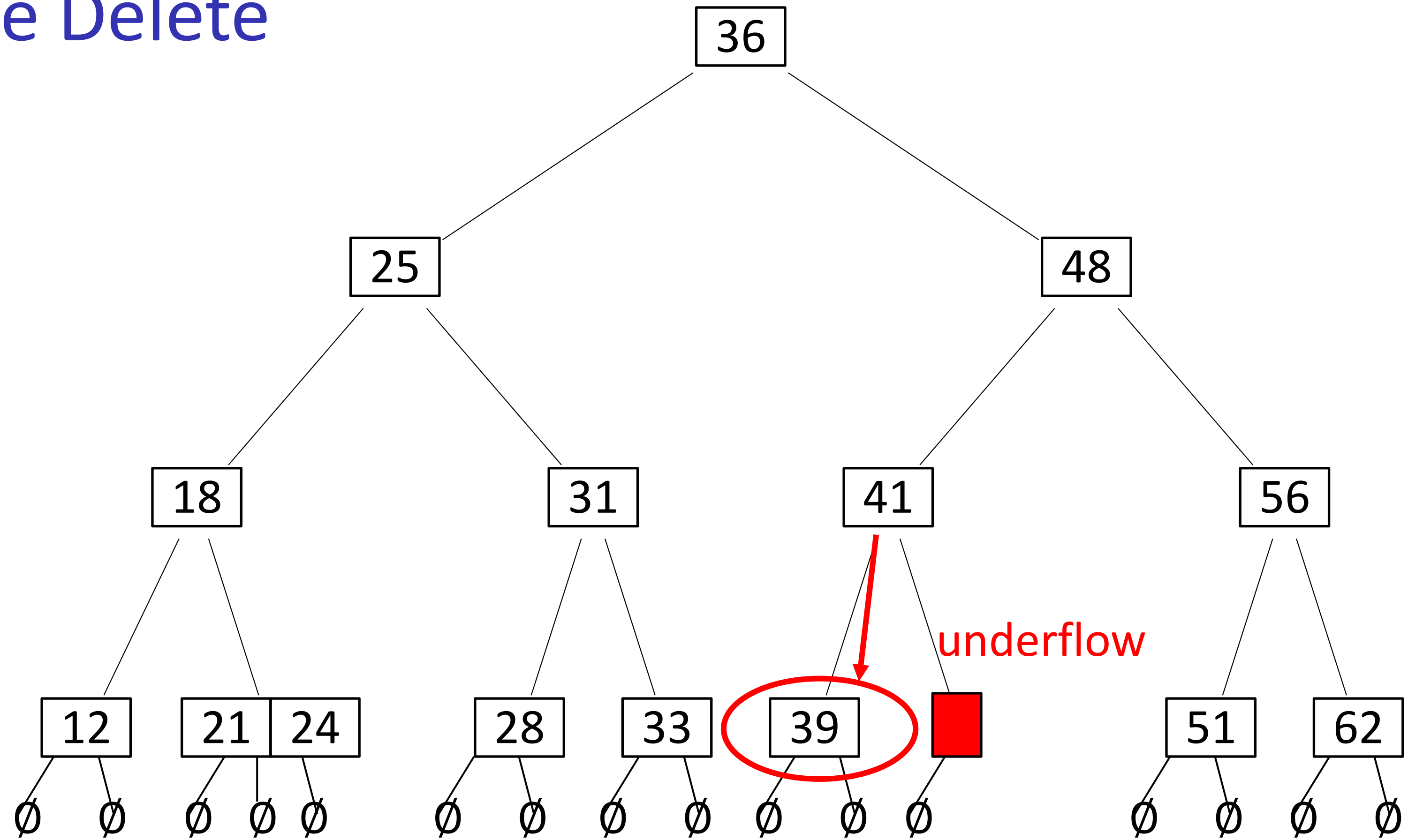

- Example: *delete*(19)
  - immediate siblings exist, but not rich, cannot transfer
    - merge with right immediate sibling with help from parent
      - all subtrees merged together as well
  - structural and order properties are preserved

# 2-4 Tree Delete



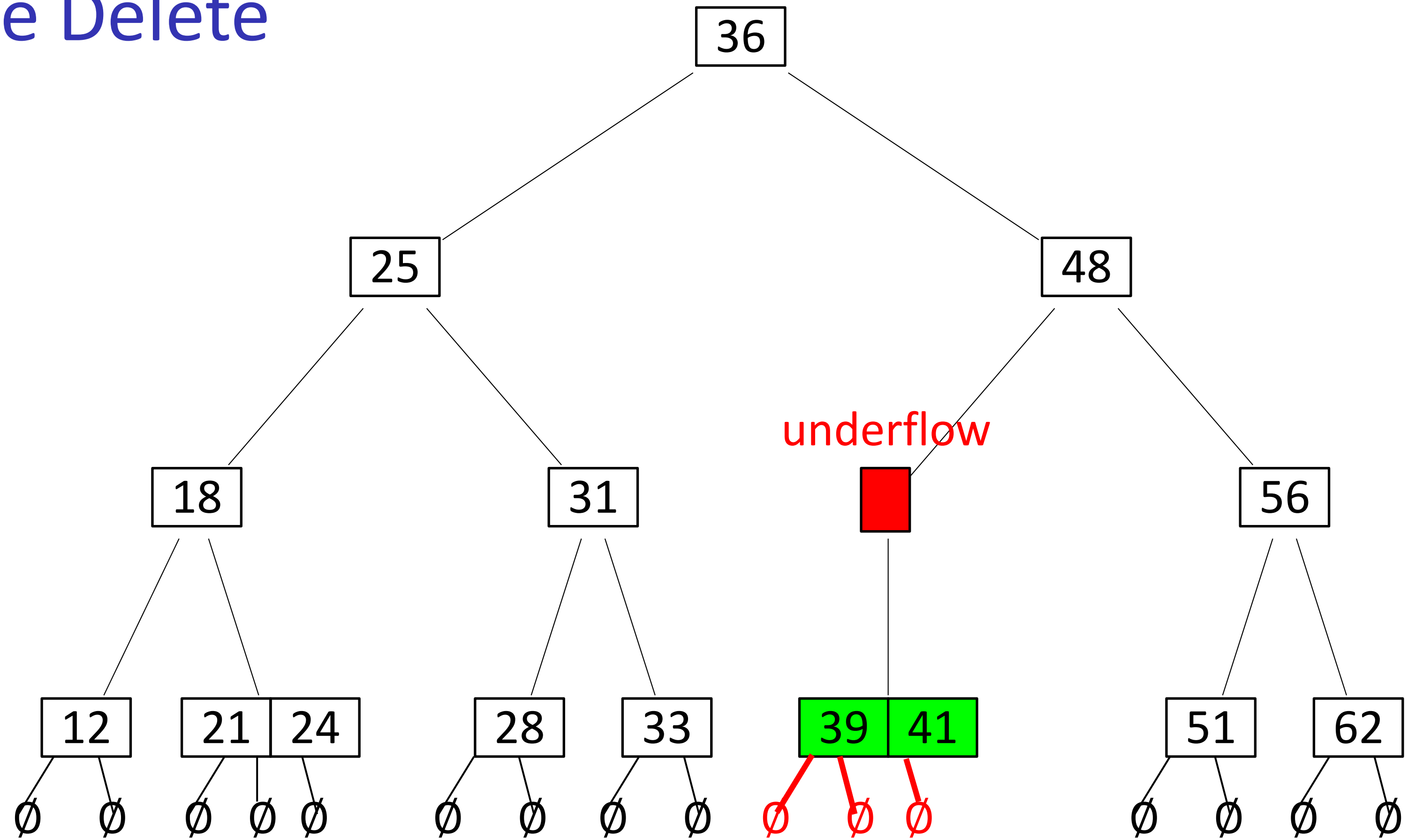

- Example: *delete*(42)
  - first search(42)
  - delete key 42 with one empty subtree

# 2-4 Tree Delete



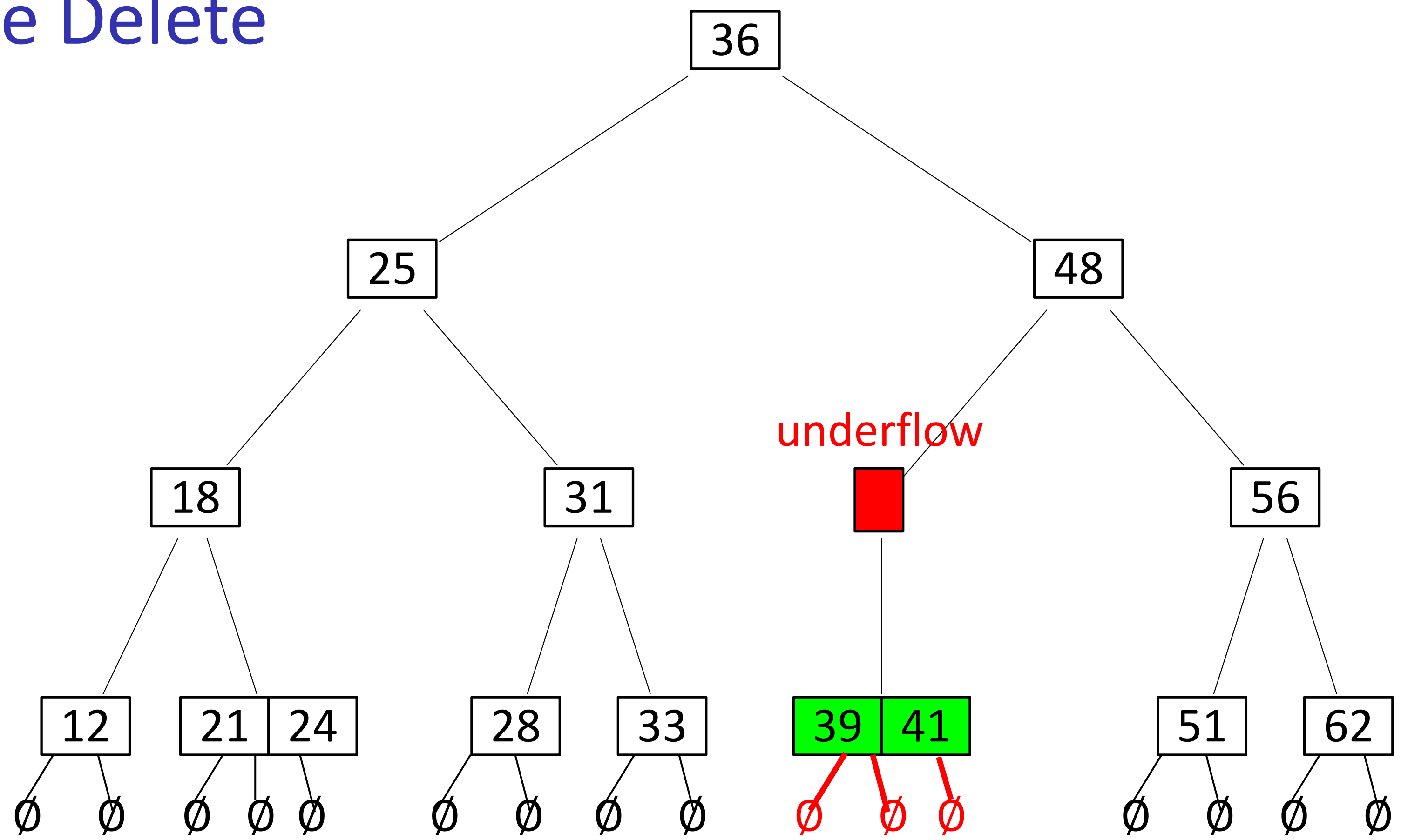

- Example: *delete*(42)
  - first search(42)
  - the only immediate sibling is not rich, perform merge

# 2-4 Tree Delete



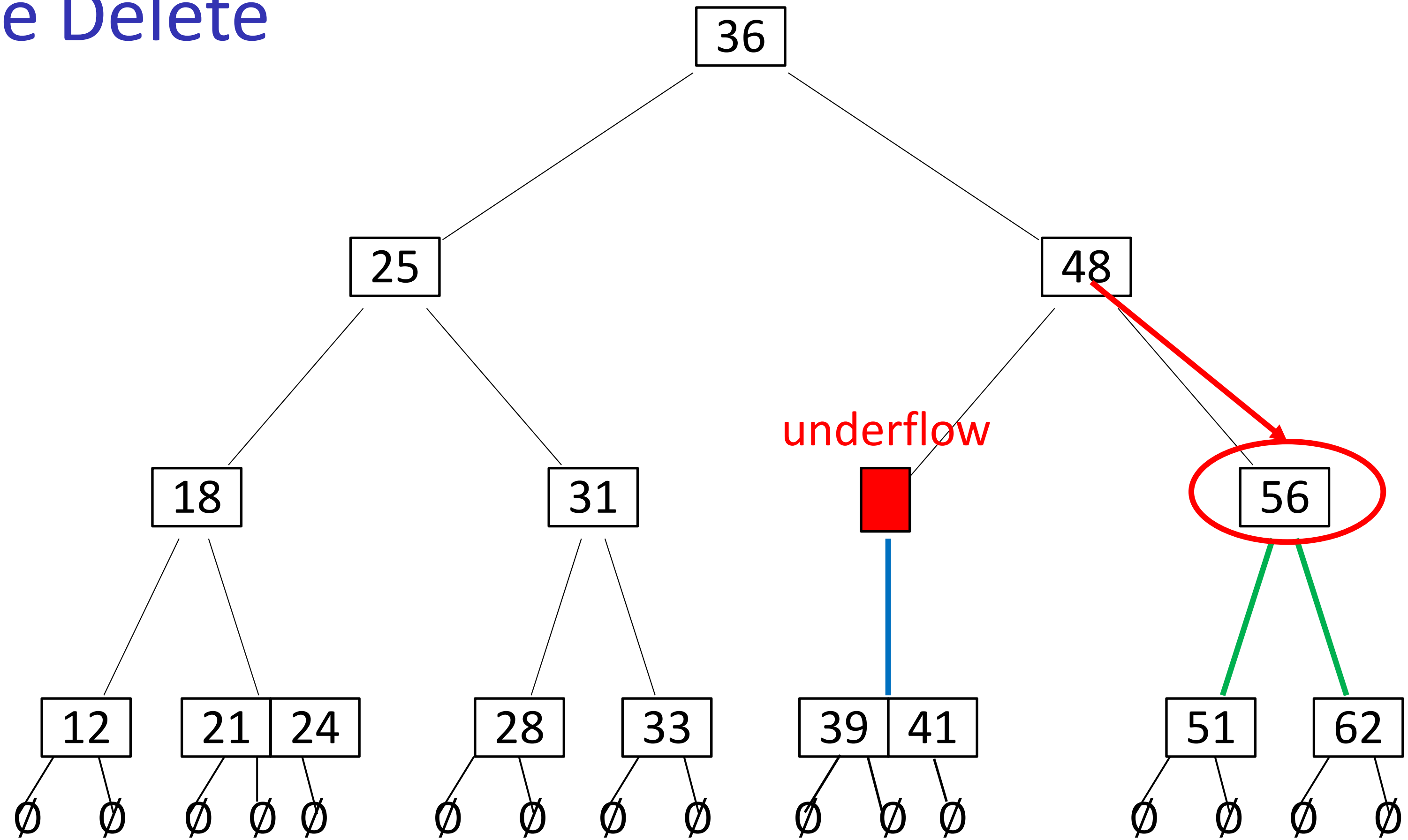

- Example: *delete*(42)
    - first search(42)
    - the only immediate sibling is not rich, perform merge
        - all subtrees merged together as well

# 2-4 Tree Delete



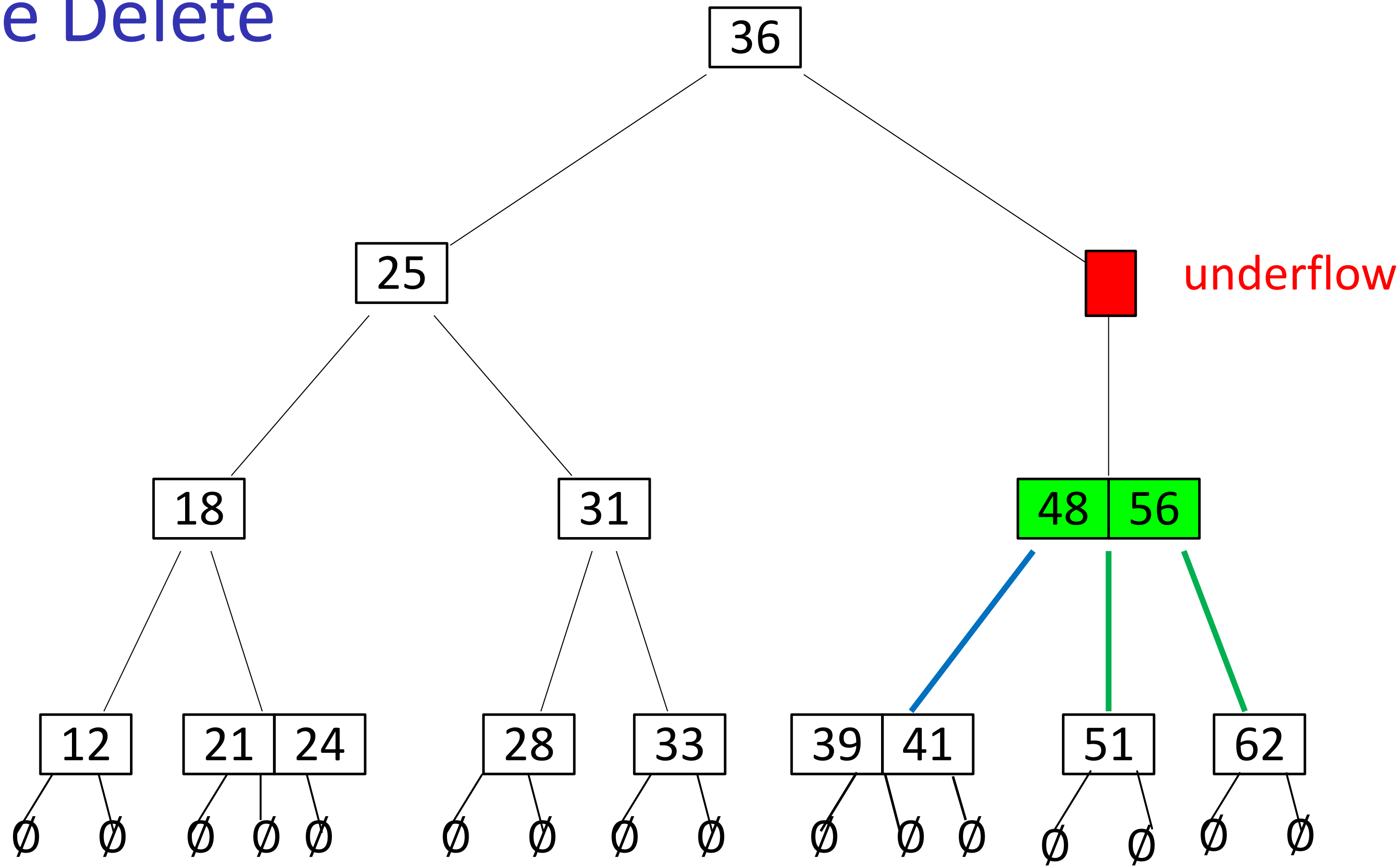

- Example: *delete*(42)
  - merge operation can cause underflow at the parent node
  - while needed, continue fixing the tree upwards
    - possibly all the way to the root

# 2-4 Tree Delete



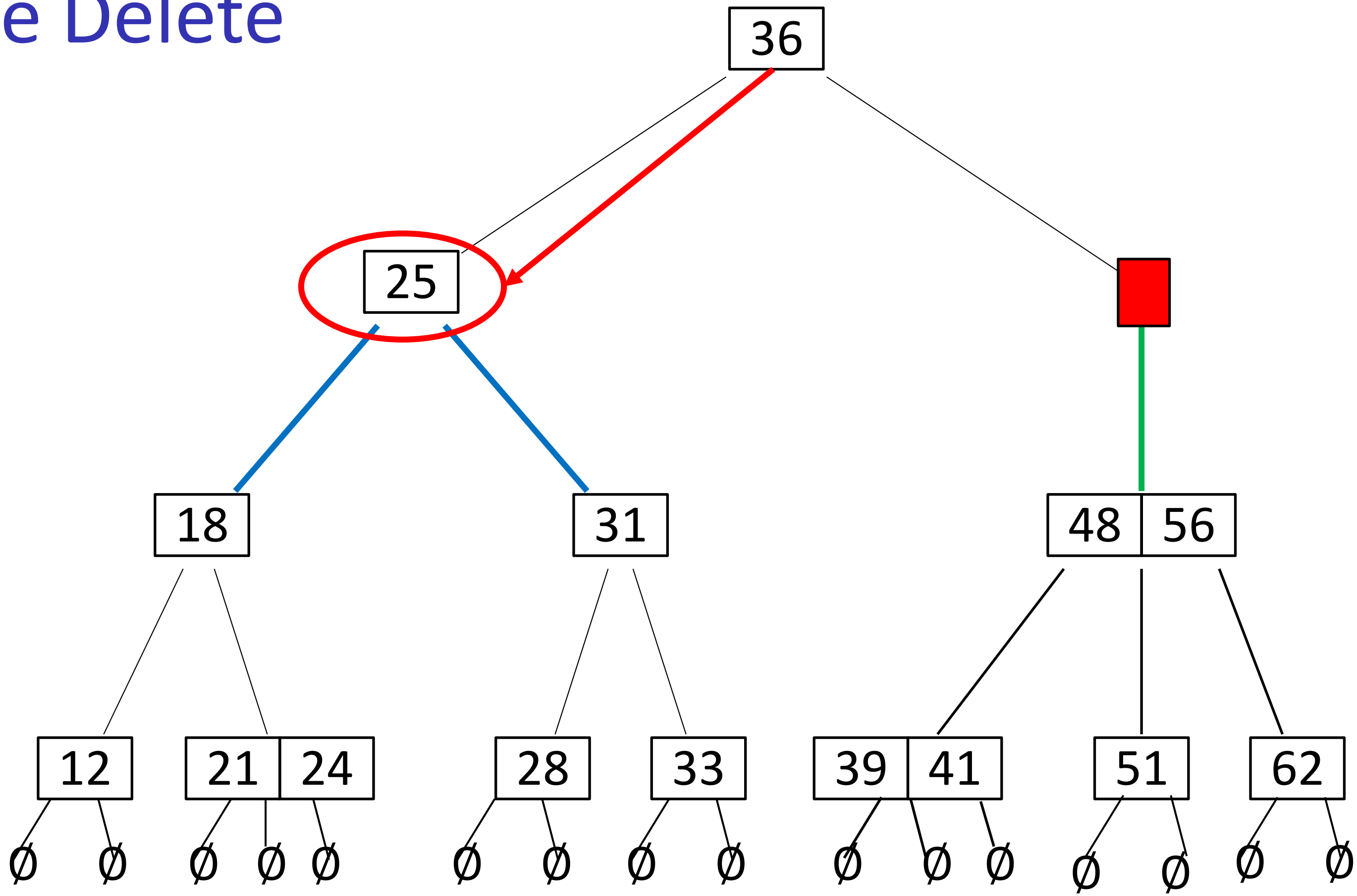

- Example: *delete*(42)
  - the only sibling is not rich, perform a merge

# 2-4 Tree Delete



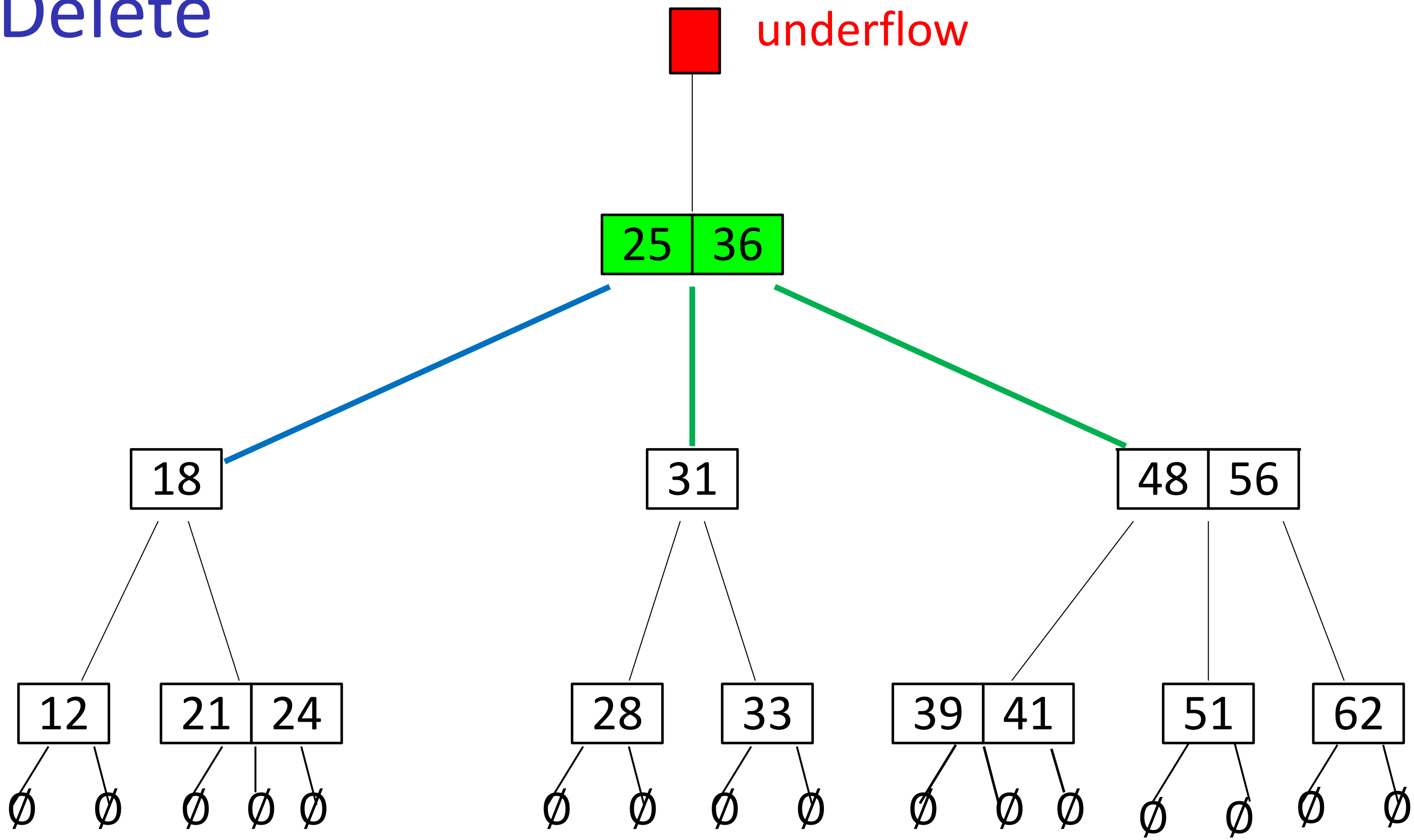

- Example: *delete*(42)
  - the only sibling is not rich, perform a merge
  - subtrees are merged as well
  - continue fixing the tree upwards

# 2-4 Tree Delete



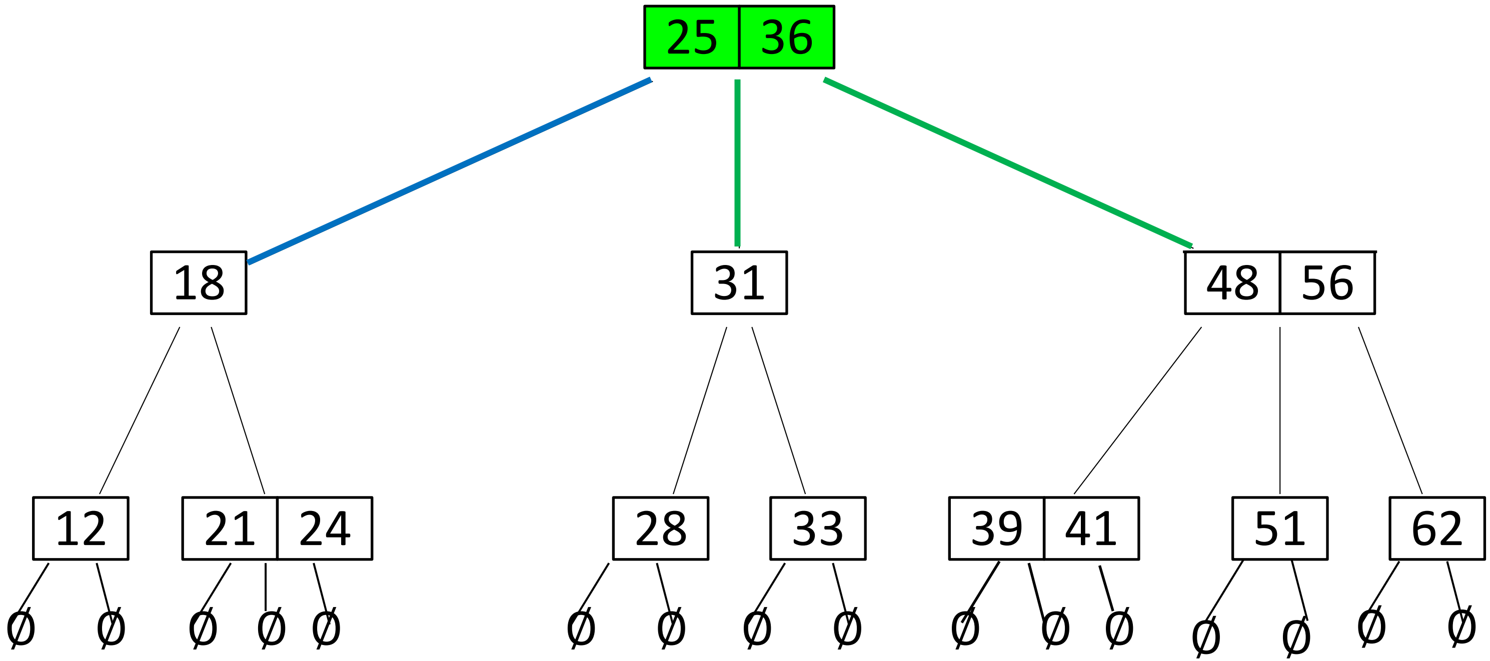

- Example: *delete*(42)
  - the only sibling is not rich, perform a merge

# 2-4 Tree Delete



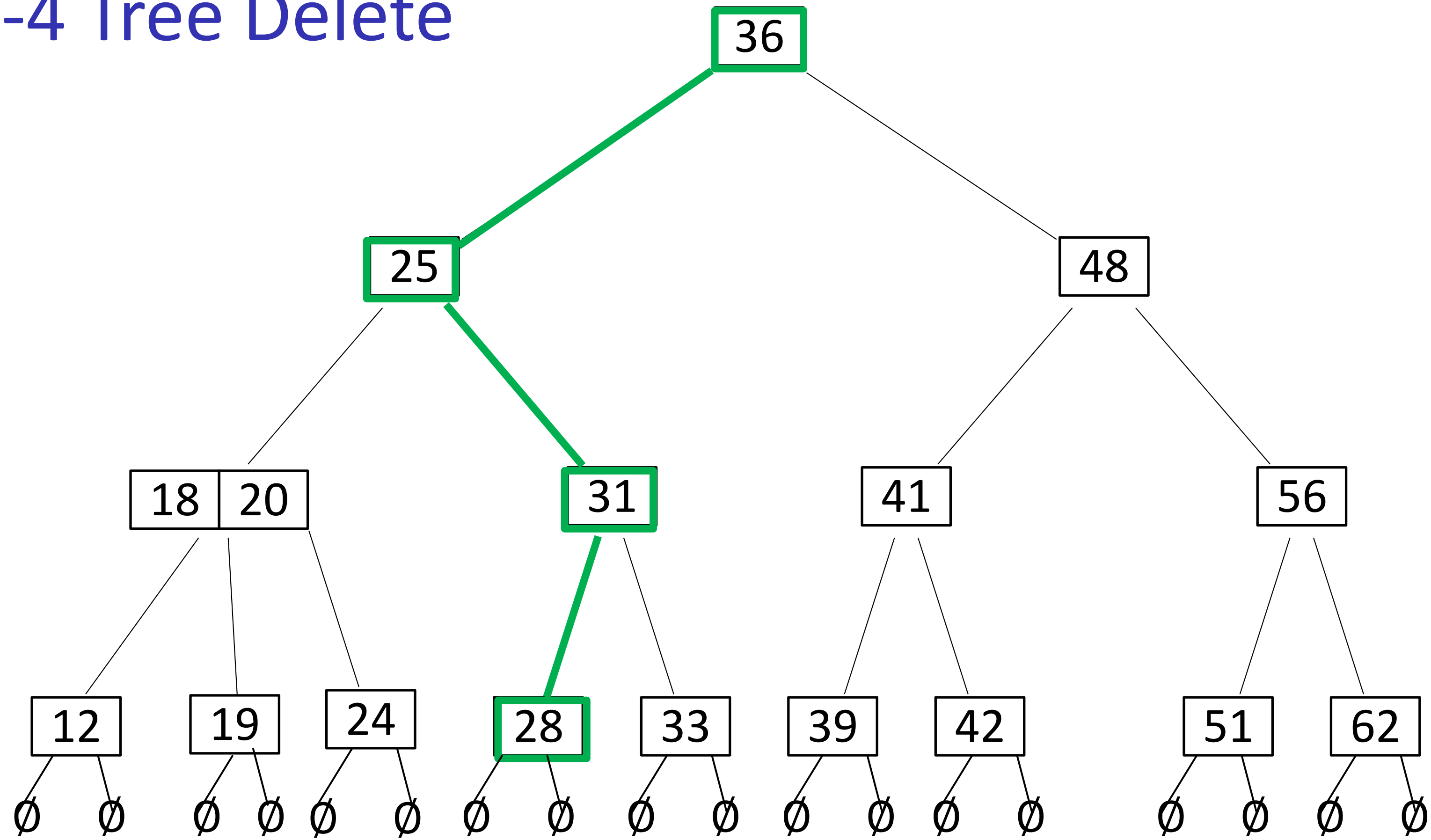

- Example: *delete*(42)
  - the only sibling is not rich, perform merge
  - underflow at parent node
    - it is the root, delete root

# 2-4 Tree Delete



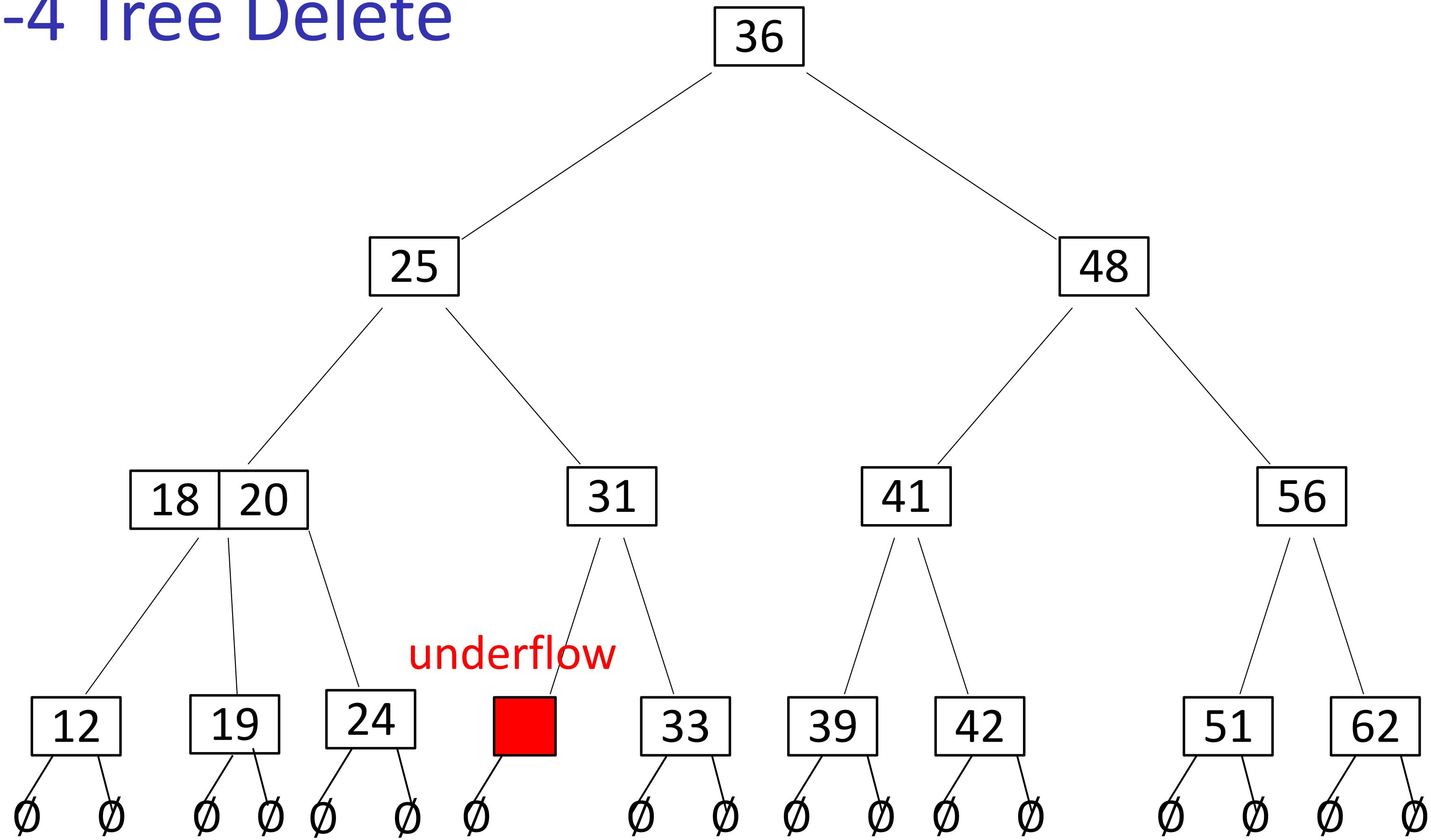

- Example: *delete*(42)
  - the only sibling is not rich, perform merge
  - underflow at parent node
    - it is the root, delete root

# 2-4 Tree Delete



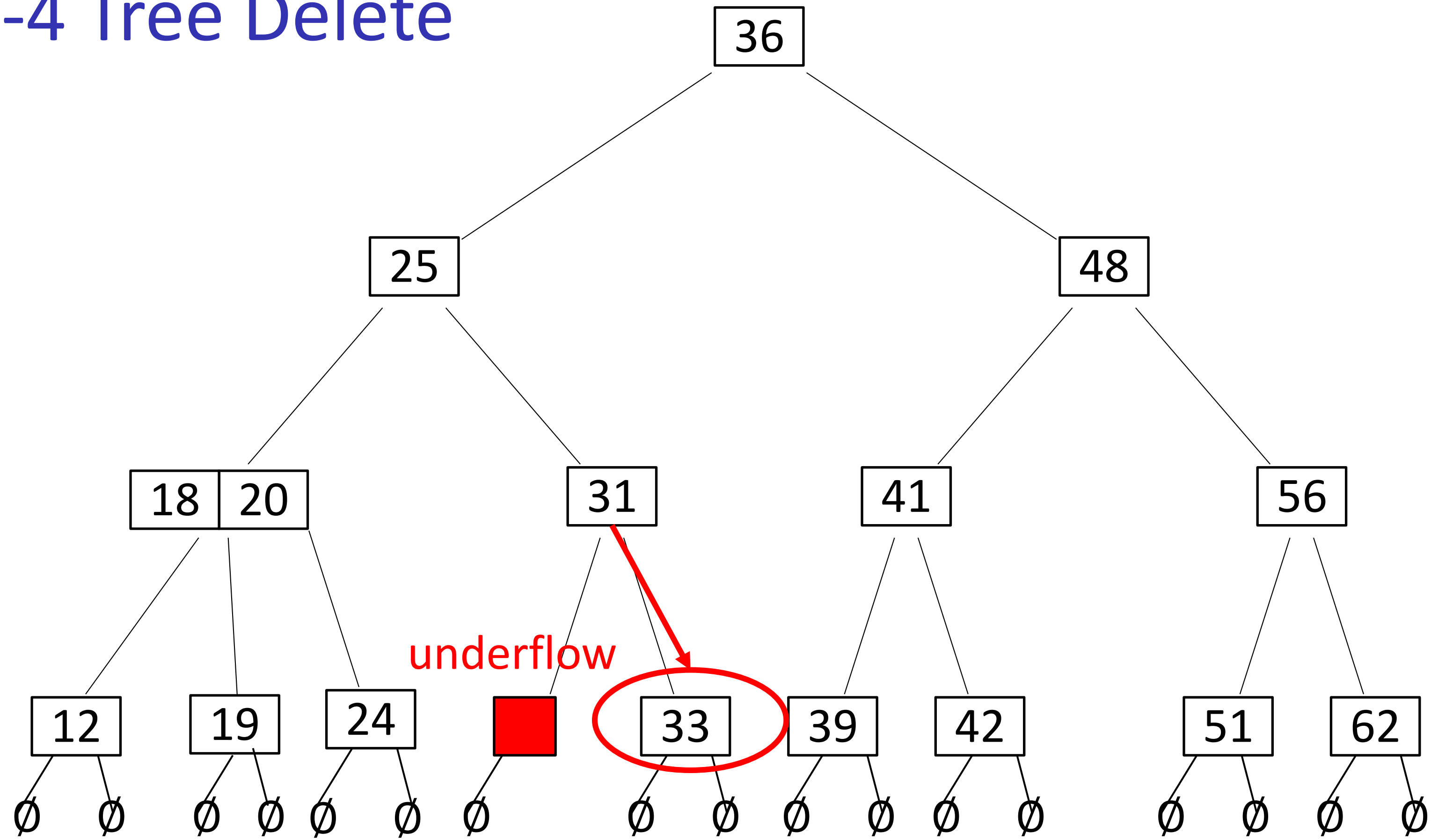

- Example: *delete*(28)
    - first search(28)
    - delete key 28 with one empty subtree

# 2-4 Tree Delete



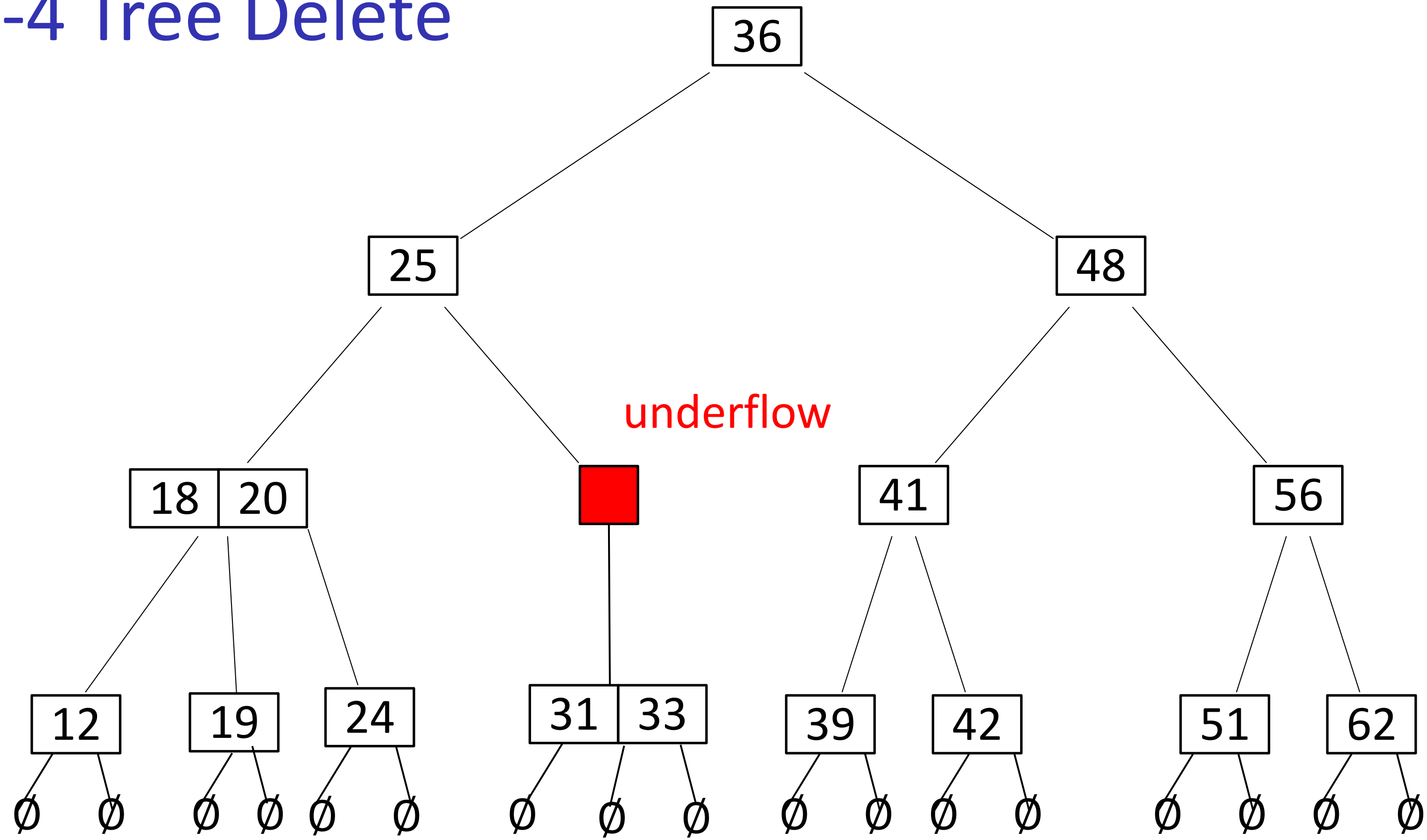

- Example: *delete*(28)
  - first search(28)
  - delete key 28 with one empty subtree

# 2-4 Tree Delete



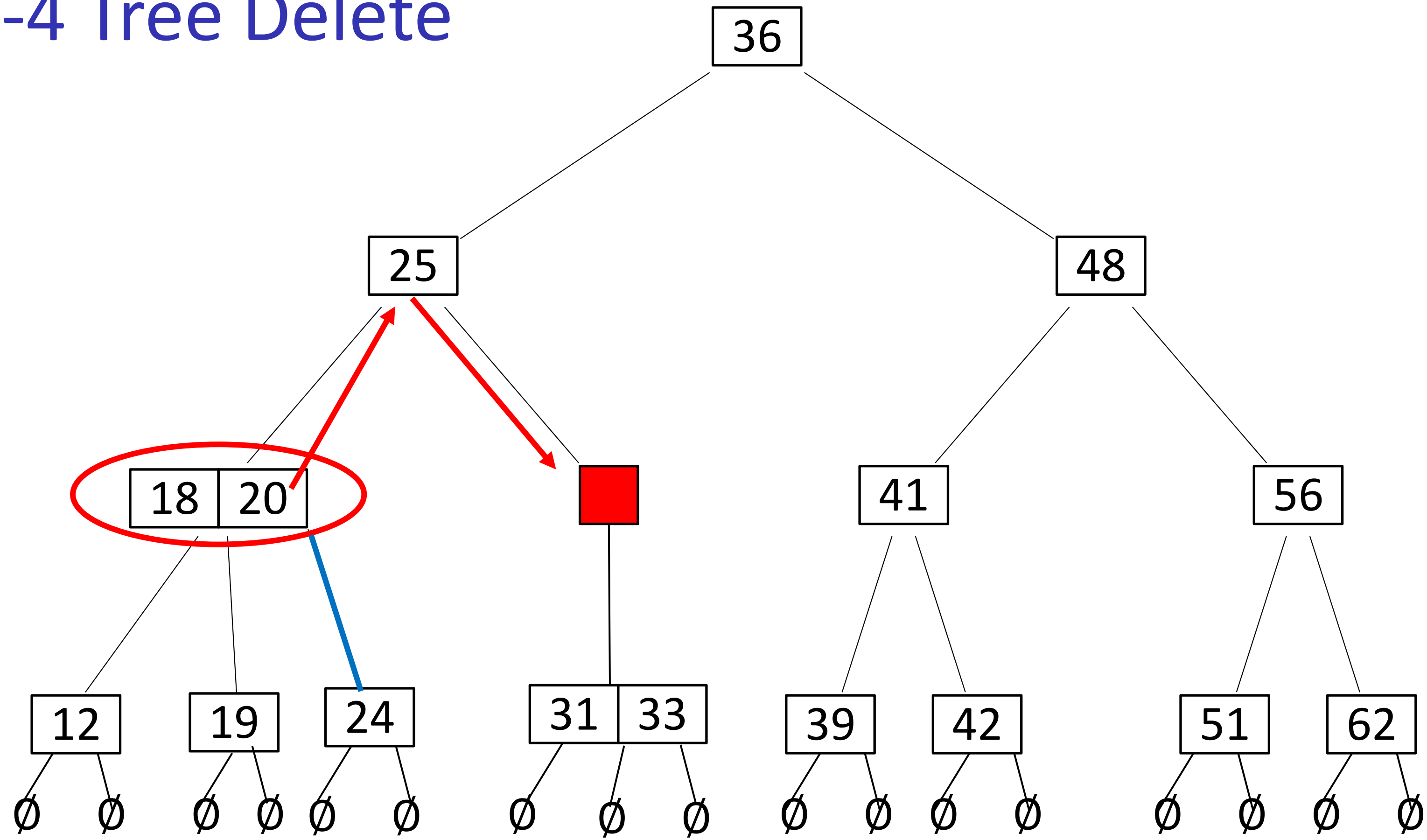

- Example: *delete*(28)
    - first search(28)
    - delete key 28 with one empty subtree
    - merge with the only immediate sibling, who is not rich

# 2-4 Tree Delete



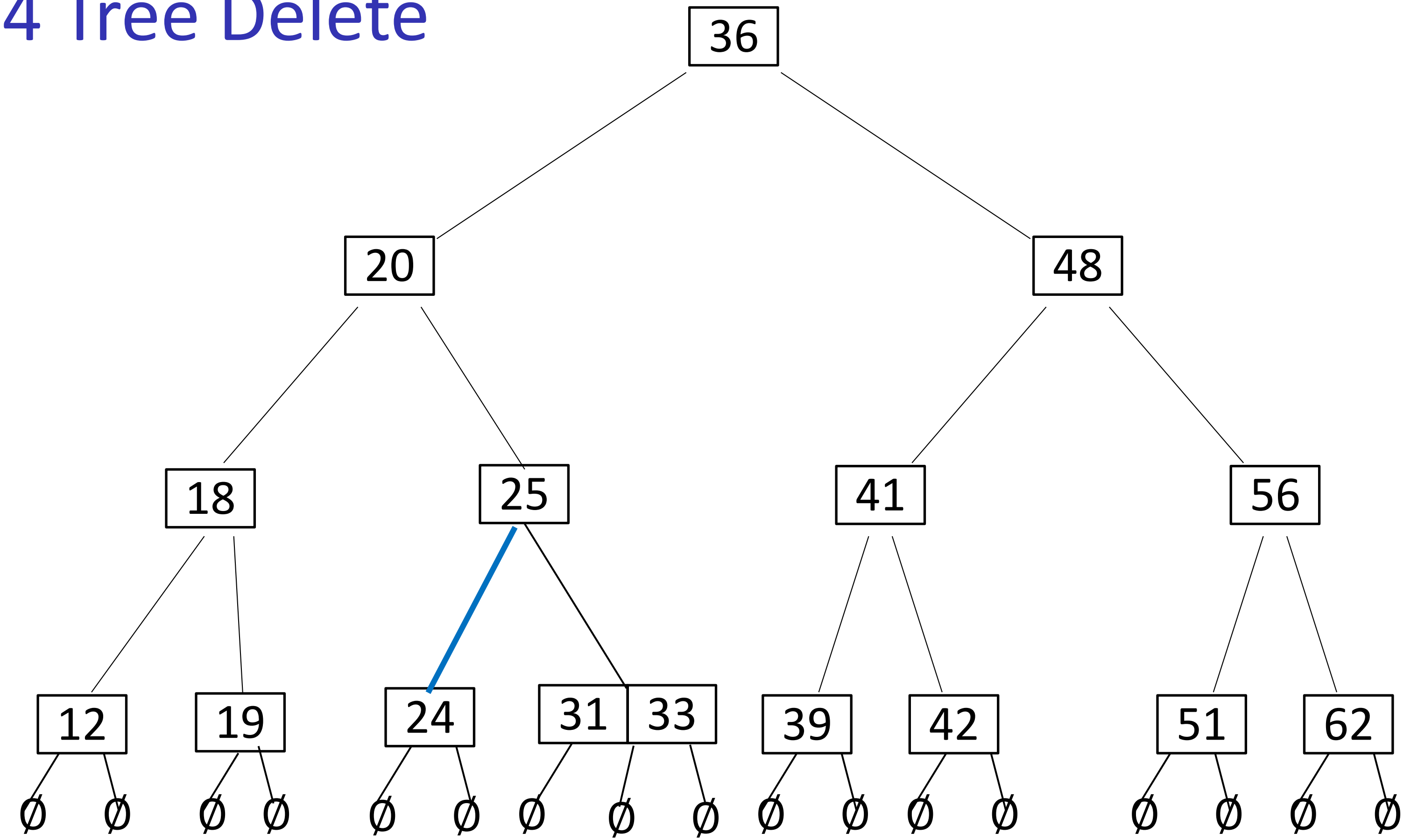

- Example: *delete*(28)
  - first search(28)
  - delete key 28 with one empty subtree
  - merge with the only immediate sibling, who is not rich

# 2-4 Tree Delete




- Example: *delete*(28)
  - transfer from a rich immediate sibling

# 2-4 Tree Delete




- Example: *delete*(28)
  - transfer from a rich immediate sibling
    - together with a subtree

# 2-4 Tree Delete Summary

- If key not at a leaf node, swap with inorder successor (guaranteed at leaf node)
- Delete key and one empty subtree from the leaf node involved in swap
- If underflow
    - If there is an immediate sibling with more than one key, transfer
        - no further underflows caused
            - do not forget to transfer a subtree as well
        - convention: if two siblings have more than one key, transfer with the right sibling
    - If all immediate siblings have only one key, merge
        - there must be at least one sibling, unless root
            - if root, delete
        - convention: if two immediate siblings with one key, merge with the right one
        - merge may cause underflow at the parent node, continue to the parent and fix it, if necessary

# Deletion from a 2-4 Tree

```
24Tree::delete(k)
    v ← 24Tree::search(k)  //node containing k
    if v is not a leaf
            swap k with its inorder successor k′
            swap v with leaf that contained k′
    delete k  and one empty subtree in key-subtree-list of v
    while v has 0 keys // underflow
        if v is the root, delete v and break
        if v has immediate sibling u with 2 or more KVPs  // transfer, then done!
            transfer the key of u that is nearest to v to p
            transfer the key of p between u and v to v
            transfer the subtree of u that is nearest to  v to v
            break
        else // merge and repeat
            u ← immediate sibling of v
            transfer the key of p between u and v to u
            transfer the subtree of v to u
            delete node v
            v ← p
```

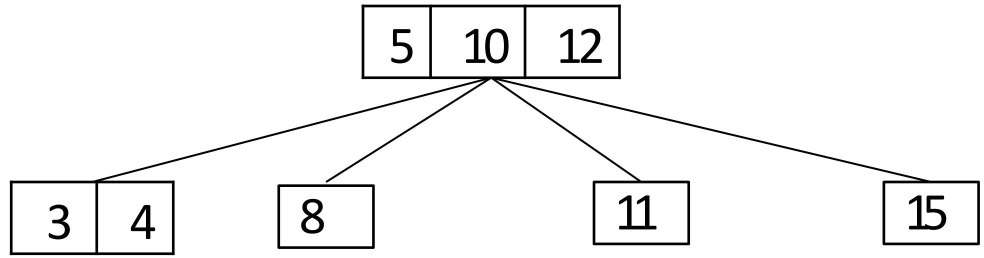
# 2-4 Tree Summary

- 2-4 tree has height $O(\log n)$
  - in internal memory, all operations have run-time $O(\log n)$
  - this is no better than AVL-trees in theory
  - but 2-4 trees are faster than AVL-trees in practice, especially when converted to binary search trees called red-black trees
- 2-4 tree has height $\Omega(\log n)$
  - $n$ is the number of KVPs
  - for a tree of height $h$
    - $n \leq 3(4^0 + 4^1 \ldots + 4^h)$
    - $n \leq 4^{h+1} - 1$
    - $\log_4(n+1) - 1 \leq h$
  - thus $h$ is $\Omega(\log n)$
- So 2-4 tree is not significantly better than AVL-tree wrt block transfers
- But can generalize the concept to decrease the height

# Outline

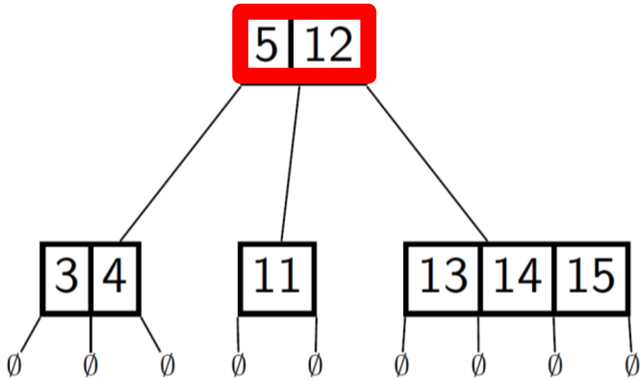
- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

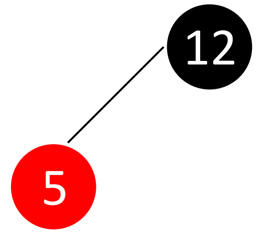
# Problem with 2-4 trees



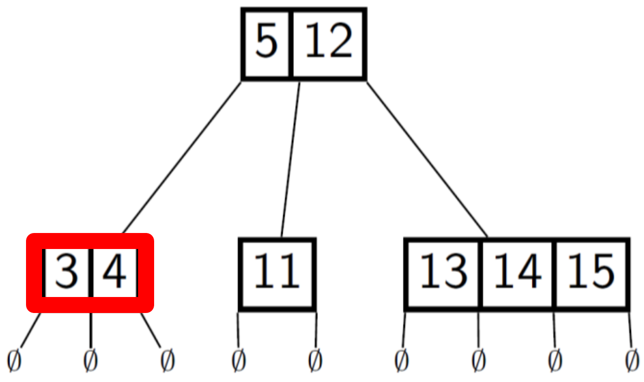

- Have 3 kinds of nodes
  - 1-node, 2-node, 3-node
  - need to store up to 7 items at each node
    - 3 keys and 4 subtree references
- How should we store keys and subtrees?
  - array of length 7
    - wastes space
  - linked list
    - overhead for list-nodes, also wastes space
  - theoretical bound not affected, but matters in practice
- Better idea
  - design a class of binary search trees that mirrors 2-4 tree

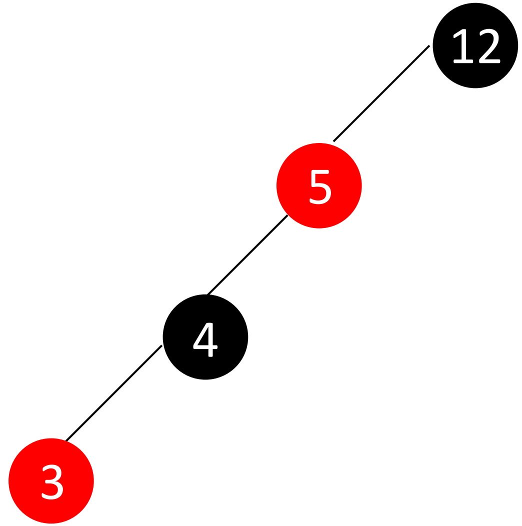
# 2-4 tree to red-black tree



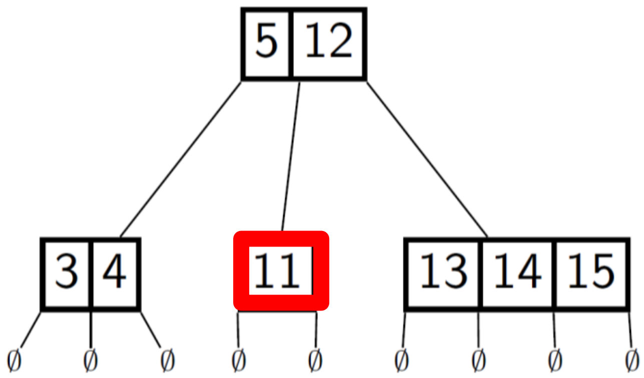



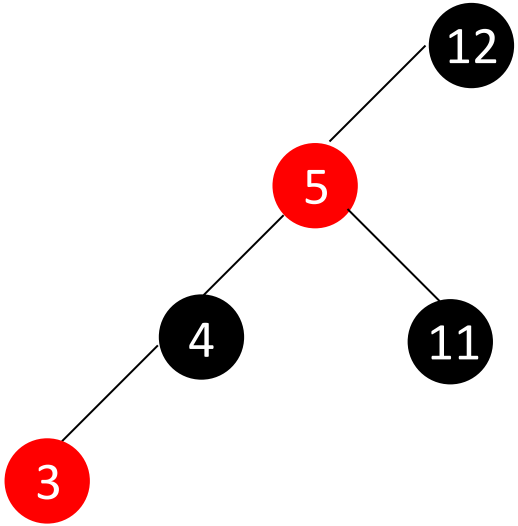
# 2-4 tree to red-black tree



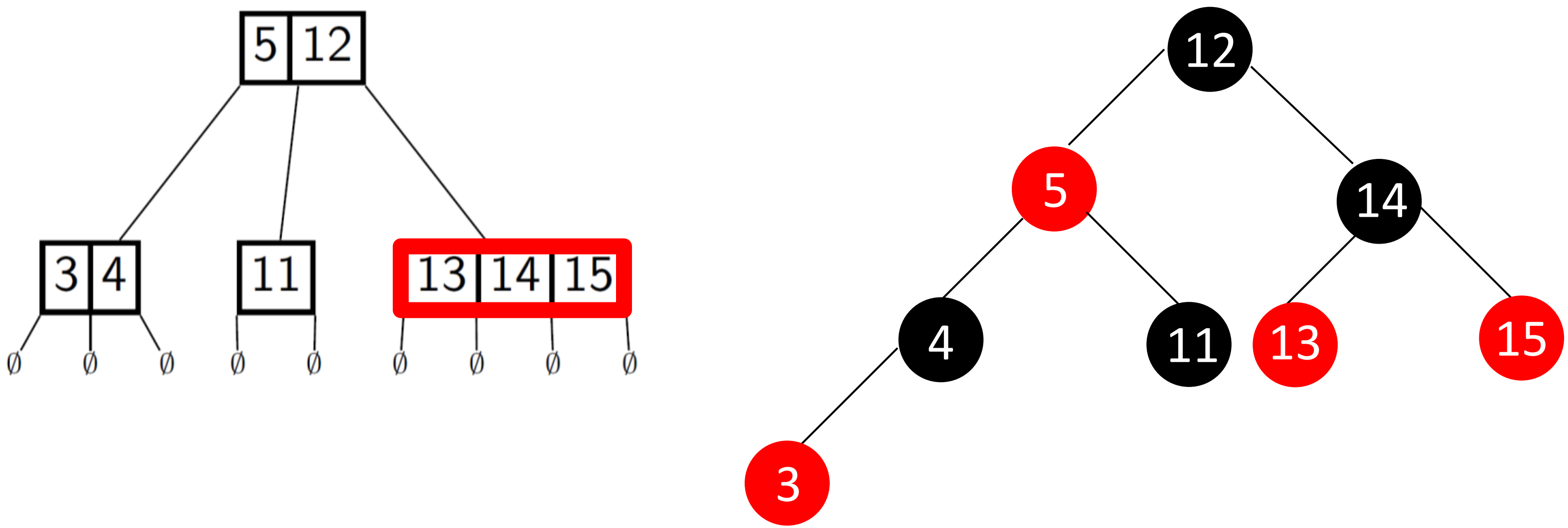



# 2-4 tree to red-black tree

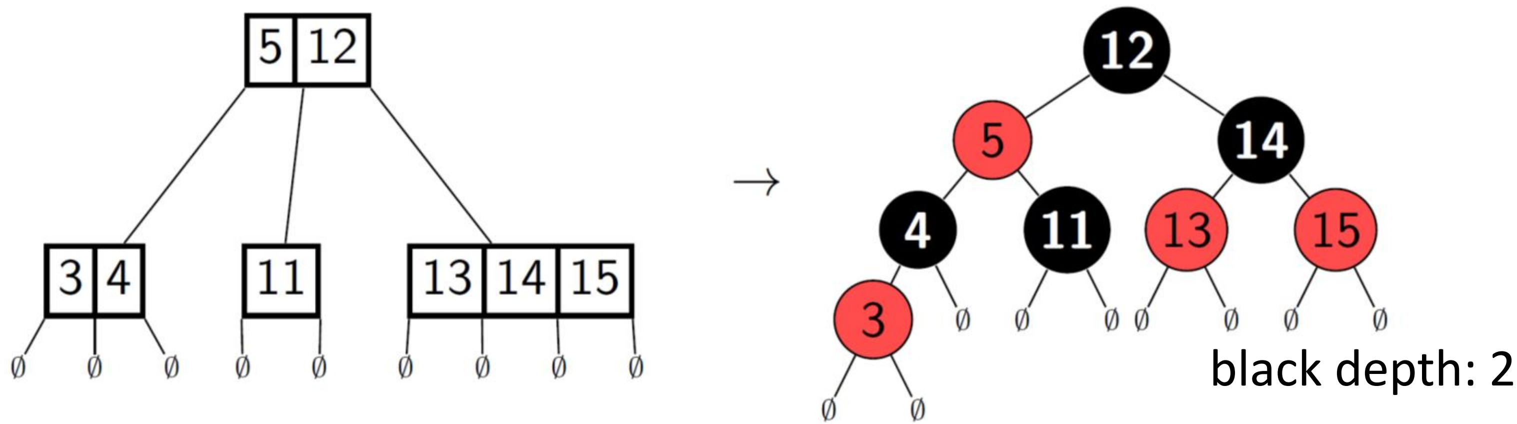


# 2-4 tree to red-black tree

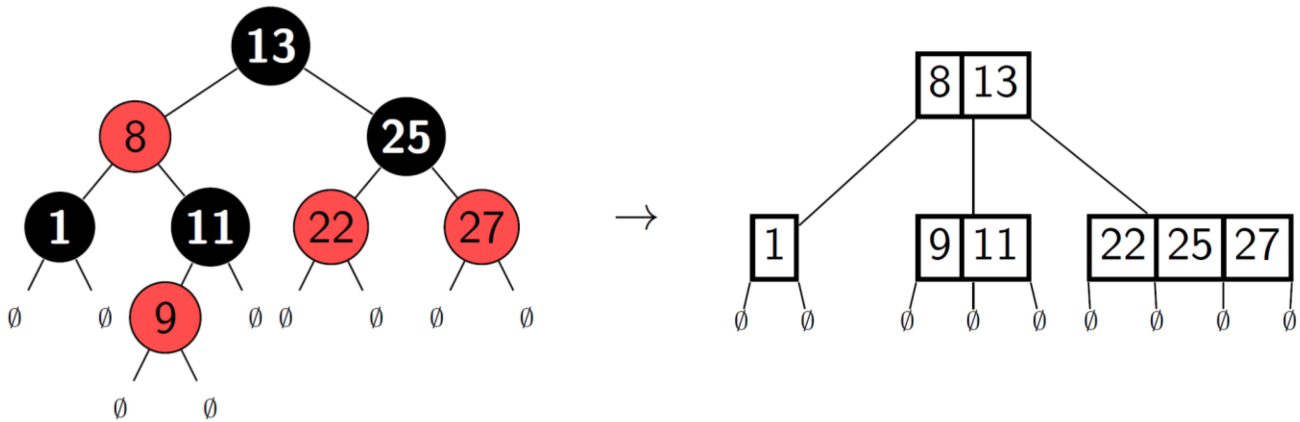


# 2-4 tree to red-black tree



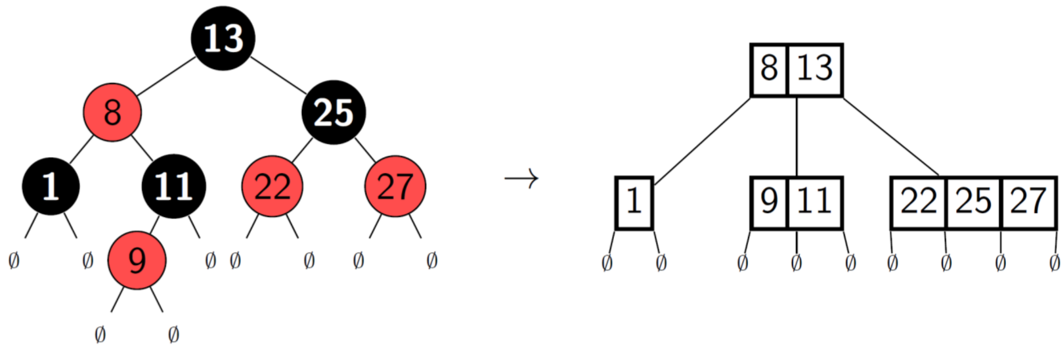

- Binary search tree that mirrors 2-4 tree
- $d$-node becomes a black node with $d-1$ red children
  - assembled so that they form a BST of height at most 1
- Overhead: red/black 'color' is stored with just 1 extra bit per node
- Resulting properties
  - any red node has a black parent
  - any empty subtree of $T$ has the same black-depth
    - number of black nodes on path form root to $T$

# Red-Black tree to 2-4 tree




- Lemma: Any red-black tree can be converted to a 2-4 tree
- Proof:
  - black node with $0 \leq d \leq 2$ red children becomes a $(d+1)$ node
  - this covers all nodes
    - no red node has a red child
  - empty subtrees on the same level due to the same blackdepth

# Red-Black tree to 2-4 tree

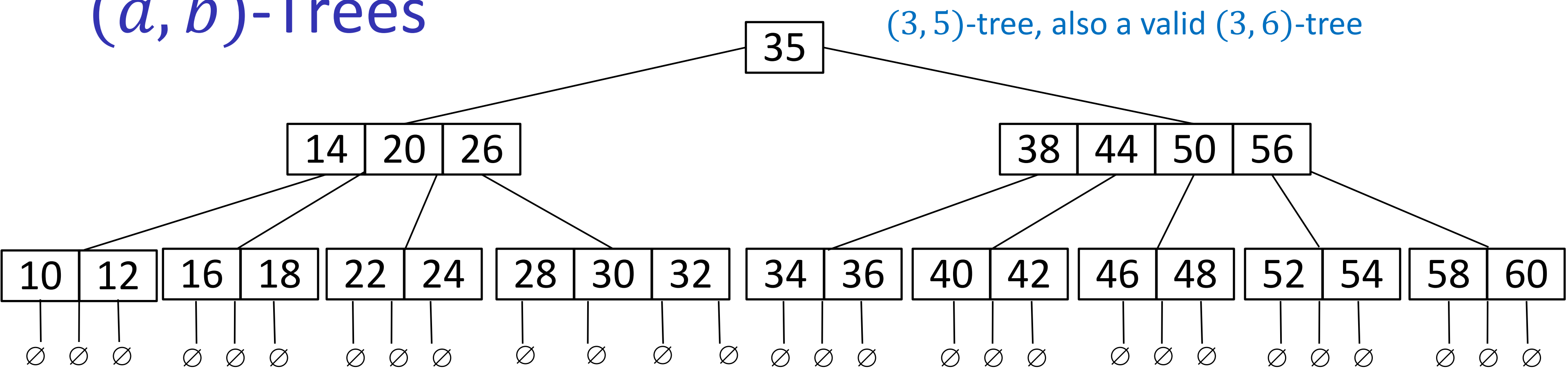



- Red-black trees have height $O(\log n)$
  - each level of 2-4 tree creates at most 2 levels in red-black tree
- Insert/delete can be done in $O(\log n)$ time
  - convert relevant part to 2-4 tree
  - do insert/delete as in 2-4 tree
  - convert relevant parts back to red-black tree
- Insert/delete can be done in $O(\log n)$ without conversion
  - no details
- Red/black trees are very popular balanced search trees (std::map)

# Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

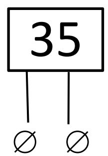
# $(a, b)$-Trees




- 2-4 Tree is a specific type of $(a, b)$-tree
- $(a, b)$-tree satisfies
  - each node has at least $a$ subtrees, unless it is the root
    - root must have at least 2 subtrees
  - each node has at most $b$ subtrees
  - if node has $d$ subtrees, then it stores $d - 1$ key-value pairs (KVPs)
  - all empty subtrees are at the same level
  - keys in the node are between keys in the corresponding subtrees
  - requirement: $2 \leq a \leq \left\lceil \frac{b}{2} \right\rceil$
    - lower bound on $a$ is needed to bound height
    - upper bound on $a$ is needed during operations
    - $b \geq 3$ follows from $2 \leq a \leq \left\lceil \frac{b}{2} \right\rceil$

# $(a, b)$-Trees: Root

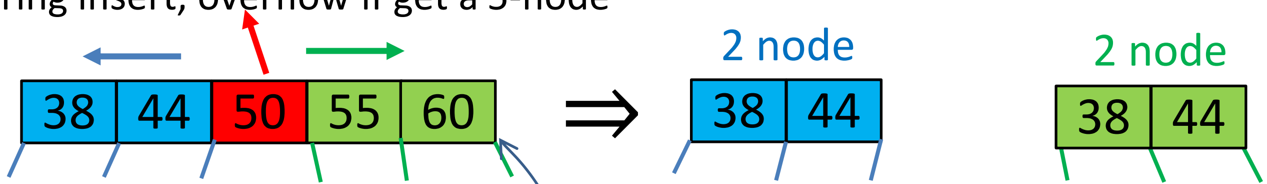
- Why special condition for the root?
- Needed for (a,b)-tree storing very few KVP
- (3,5) tree storing only 1 KVP




- Could not build it if forced the root to have at least 3 children
    - number of keys at any node is one less than number of subtrees

# $(a, b)$-Trees: Condition on $a$ Explained

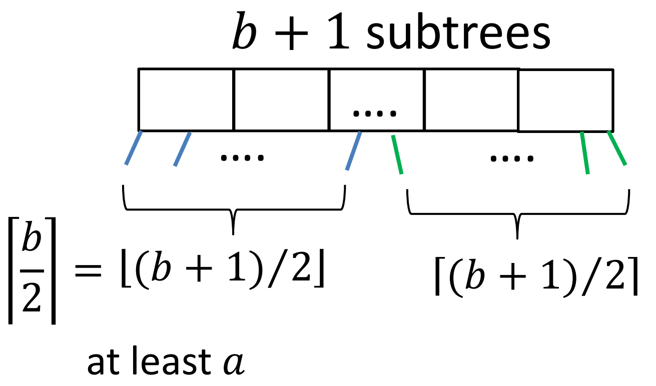
- Because $a \leq \left\lceil \frac{b}{2} \right\rceil$ *search*, *insert*, *delete* work just like for 2-4 trees
  - straightforward redefinition of underflow and overflow
- For example, for $(3,5)$-tree
  - at least 3 children, at most 5
    - allowed:  2-node, 3-node, 4-node
  - during insert, overflow if get a 5-node




  - 2-node is smallest allowed node
- If $a > \left\lceil \frac{b}{2} \right\rceil$, no valid split exists for overflowed node
  - like requiring to split a pie in 2 parts, and each part is bigger than half!
  - for example if allow $(4,5)$-tree
    - allowed: 3-node, 4-node
    - overflow when get 5-node
    - equal (best possible) split of 5-node results in two 2-node
      - 2-node is not allowed for (4,5)-tree

# $(a, b)$-Trees: Condition on $a$ Explained

- Require $a \leq \left\lceil \frac{b}{2} \right\rceil$
- Overflow means node has $b + 1$ subtrees

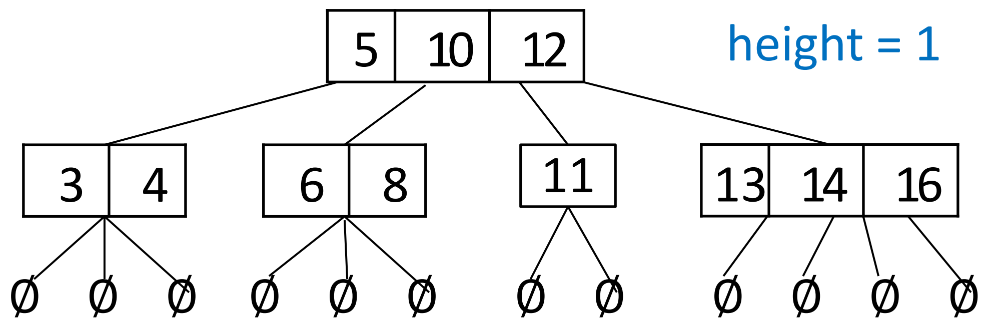


# $(a, b)$-Trees Delete

- For example, for $(3,5)$-tree
  - at least 3 children, at most 5
    - each node is at least a 2-node, at most a 4-node
  - during delete, underflow if get a 1-node
  - if we have an immediate sibling which is rich (3 or 4-node), do transfer
  - otherwise, do merge
    - guaranteed to have at least one sibling which is a 2-node

# Height of $(a, b)$-tree

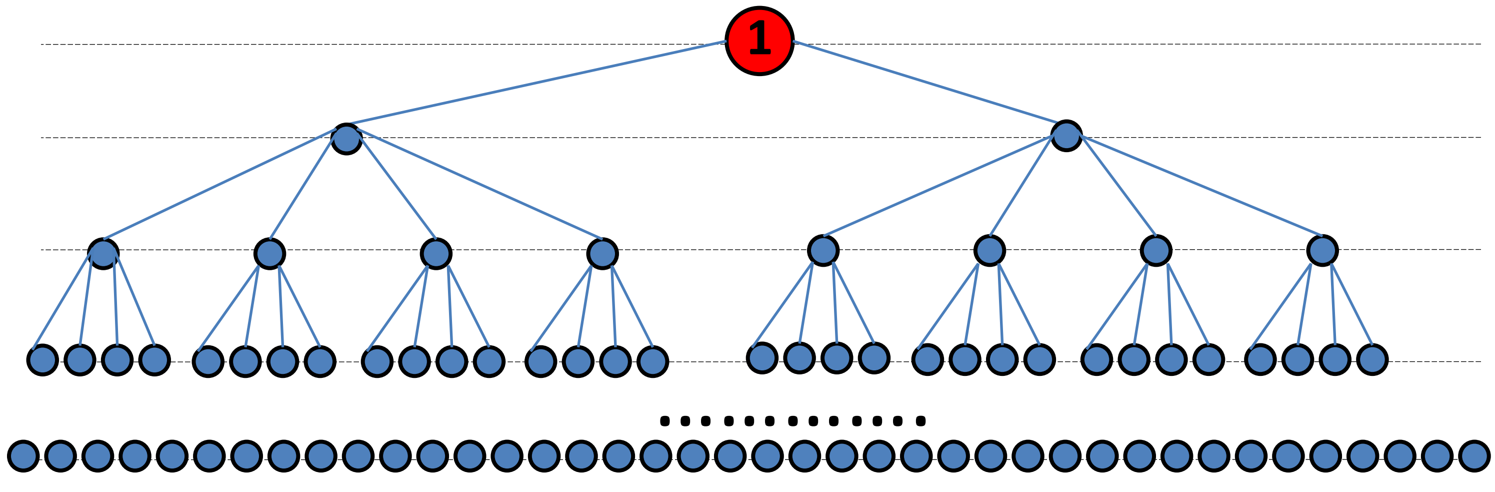
- Height = number of levels **not** counting empty subtrees



# Height of $(a, b)$-tree

- Consider (a,b)-tree with the *smallest number* of KVP and of height $h$
  - red node (the root) has 1 KVP, blue nodes have $(a-1)$ KVP

| level | # of nodes |
|---|---|
| 0 | 1 |
| 1 | $2a^0$ |
| 2 | $2a^1$ |
| 3 | $2a^2$ |
| ***h*** | $2a^{h-1}$ |




$$\text{\# of KVPs} = \mathbf{1} + \sum_{i=0}^{h-1} 2a^i(a-1) = \mathbf{1} + 2(a-1)\sum_{i=0}^{h-1} a^i = 2a^h - 1$$

$$\sum_{i=0}^{h-1} a^i = \frac{a^h - 1}{a - 1}$$

- Let $n$ the number of KVP in *any* $(a, b)$-tree of height $h$

$$n \geq 2a^h - 1 \text{, therefore, } \log_a \frac{n+1}{2} \geq h$$

- Height of tree with $n$ KVPs is $O(\log_a n) = O(\log n / \log a)$

# $(a, b)$-Tree Analysis in Internal/External Memory

- Internal memory
  - search, insert, delete each require visiting $\Theta(height)$ nodes
  - height is $O(\log n / \log a)$
  - recall that $a \leq \left\lceil \frac{b}{2} \right\rceil$ is required for insert and delete to work correctly
  - therefore, chose $a = \left\lceil \frac{b}{2} \right\rceil$ to minimize the height
  - store from $a$ to $b$ items at a node: work at a node can be done in $O(\log b)$ time
  - total cost

$$O\left(\frac{\log n}{\log a} \cdot \log b\right) = O\left(\frac{\log n}{\log \left\lceil \frac{b}{2} \right\rceil} \cdot \log b\right) = O\left(\frac{\log b}{\log b - 1} \cdot \log n\right) = O(\log n)$$

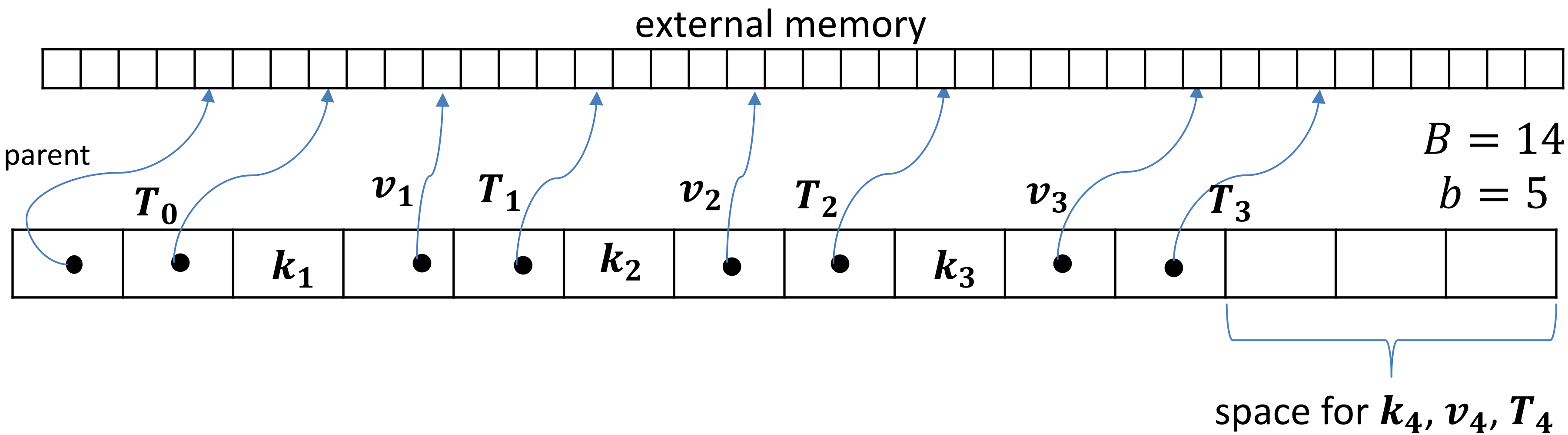
  - this is not better than AVL-trees in internal memory
- External memory
  - we count just block transfers
  - running time is $O(\log n / \log a)$, assuming each node fits into one block
  - makes sense to make $a$ as large as possible so that a node still fits into one block

# Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - red-black trees
    - (*a*, *b*)-Trees
    - B-Trees

# B-trees: Motivation

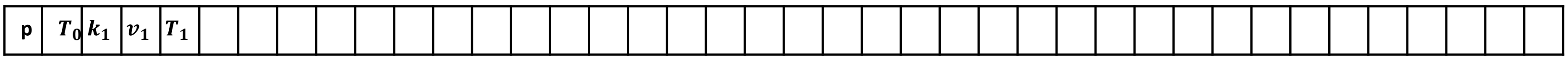
- B-tree is a type of $(a, b)$-tree tailored to the external memory model
- Each block in external memory stores one tree node




- Choose $b$ so that the largest node ($b$ subtrees) fits into one block
  - store $b - 1$ keys directly (not through reference)
  - $b - 1$ value references, $b$ subtree references, reference to parent
  - values can be stored in the block directly if they do not take much space
- Typically, $b \in \Theta(B)$
  - $B = b \cdot const$

# B-trees: Motivation

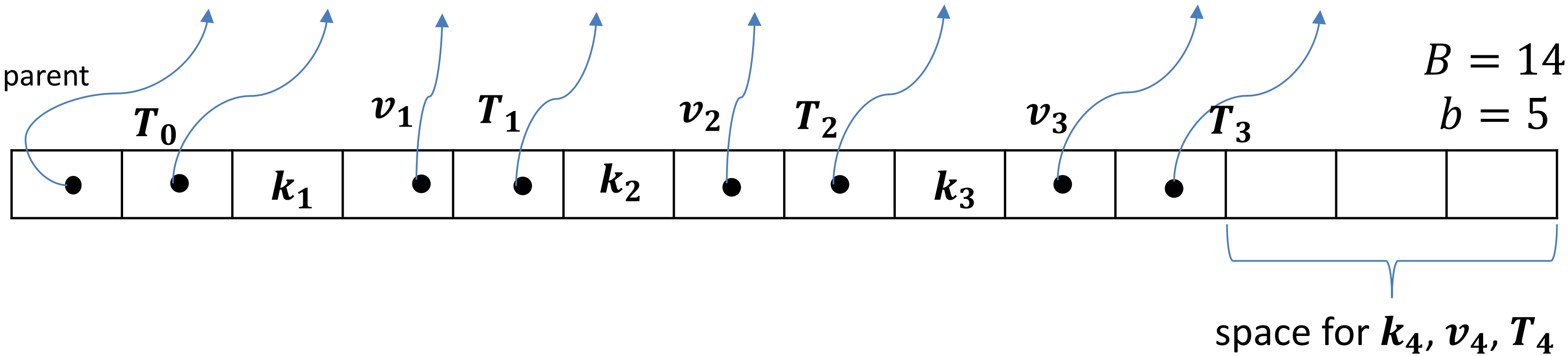
- How to chose $a$?
- Height is $O(\log n / \log a)$, so small $a$ leads to large height
  - therefore, more block transfers
- In addition, allowing small $a$ wastes block space
  - example: $a = 1$ and $B = 40$




- Therefore, make $a$ as large as possible
- Largest allowed $a = \lceil b/2 \rceil$

# B-trees: Definition

- $B$-tree is $(a, b)$-tree s.t.
  - $a = \lceil b/2 \rceil$
- Usually specify $B$-tree by just giving $b$
  - $b$ is called the order of $B$-tree
  - $B$-tree or order $b$ is a $(\lceil b/2 \rceil, b)$-tree
- For external memory
  - chose $b$ s.t. the largest possible node (i.e. $b$ subtrees) fits into a block
  - each block will be at least half full
- Example: node for B-tree of order 5



# B-tree Analysis in External Memory

- Search, insert, and delete each requires visiting $\Theta(height)$ nodes
  - $\Theta(height)$ block transfers
- Work within a node is done in internal memory, no block transfers
- The height is $\Theta(\log_b n)$ which is $\Theta(\log_B n)$
  - since $b \in \Theta(B)$
  - Proof (assuming $b \geq B/3$ and $B \geq 9$):

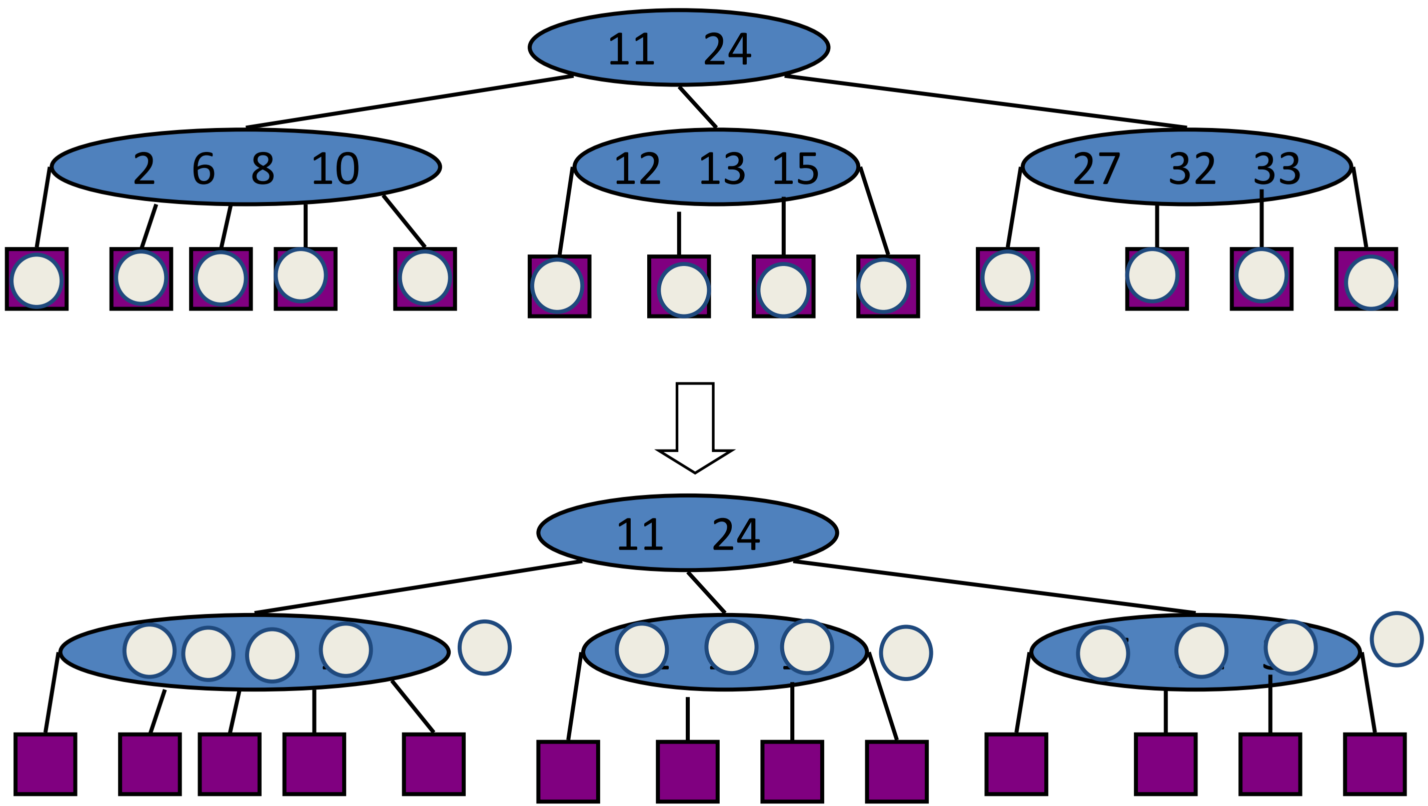
$$\log_b n = \frac{\log n}{\log b} \leq \frac{\log n}{\log B/3} \leq \frac{\log n}{\log \sqrt{B}} = 2 \log_B n$$

- So all operations require $\Theta(\log_B n)$ block transfers
  - can show that this is asymptotically optimal
- There are variants that are even better in practice
- B-trees are hugely important for storing databases (cs448)

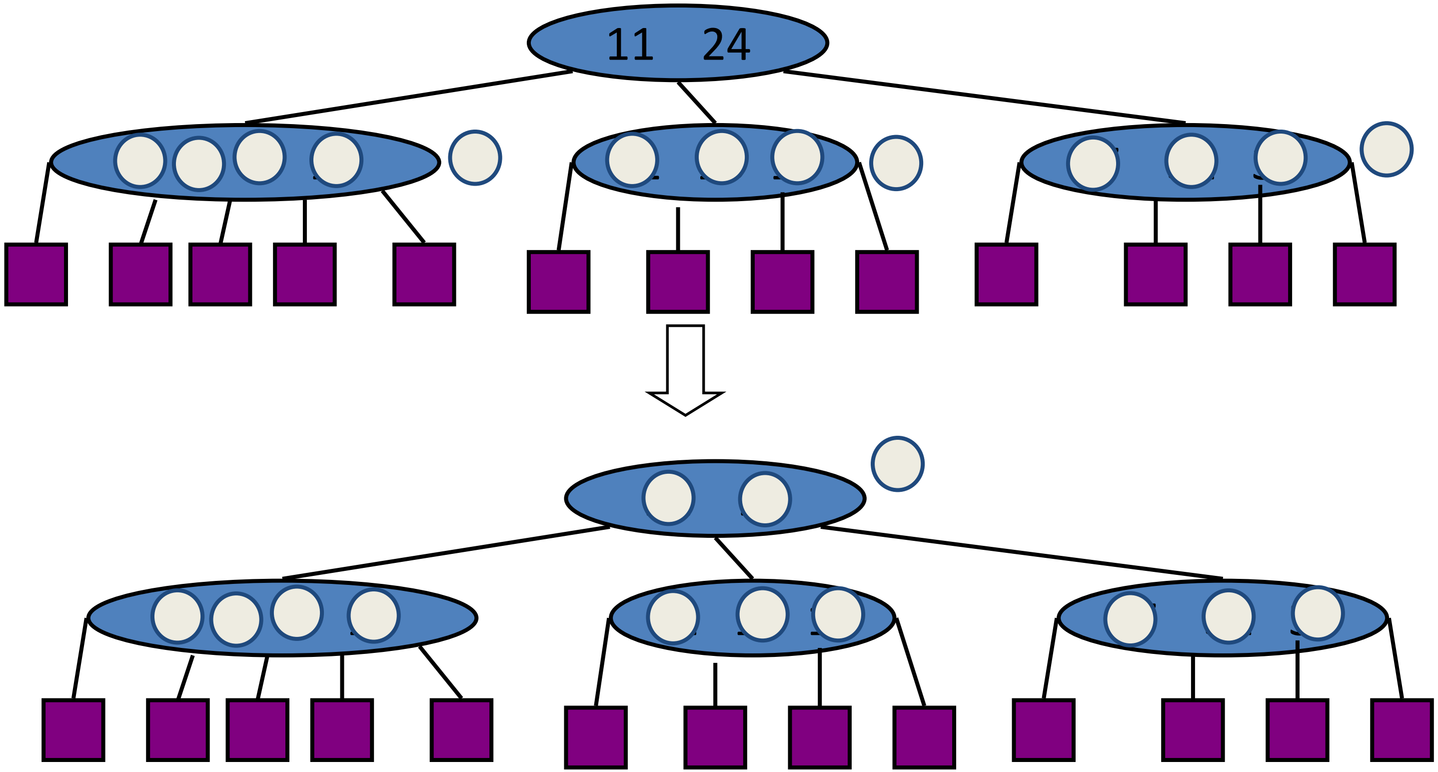
# Useful Fact about $(a, b)$-trees

- number of of KVP = number of empty subtrees – 1 in any $(a, b)$-tree

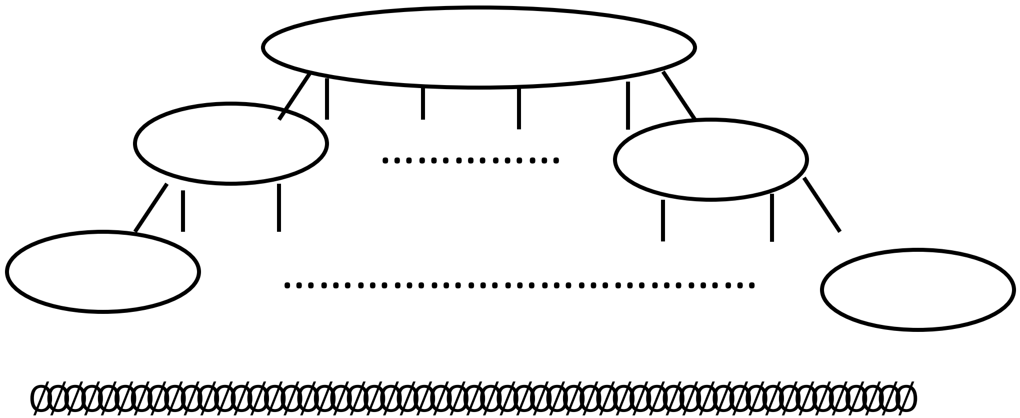
**Proof:** Put one stone on each empty subtree and pass the stones up the tree. Each node keeps 1 stone per KVP, and passes the rest to its parent. Since for each node, #KVP = # children – 1, each node will pass only 1 stone to its parent. This process stops at the root, and the root will pass 1 stone outside the tree. At the end, each KVP has 1 stone, and 1 stone is outside the tree.



# Useful Fact about $(a, b)$-trees



# Example of B-tree usage



1 node (root)

200 nodes

$200^2$ nodes

$200^3$ empty subtrees

- $B$-tree of order 200
  - $B$-tree of order 200 and height 2 can store up to $200^3 - 1$ KVPs
  - if we store root in internal memory, then only 2 block reads are needed to retrieve any item
  - compare: AVL tree of height at least 23 to store as many KVPs