CS 240 – Data Structures and Data Management

Module 11: External Memory

A. Hunt O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2023

Outline

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

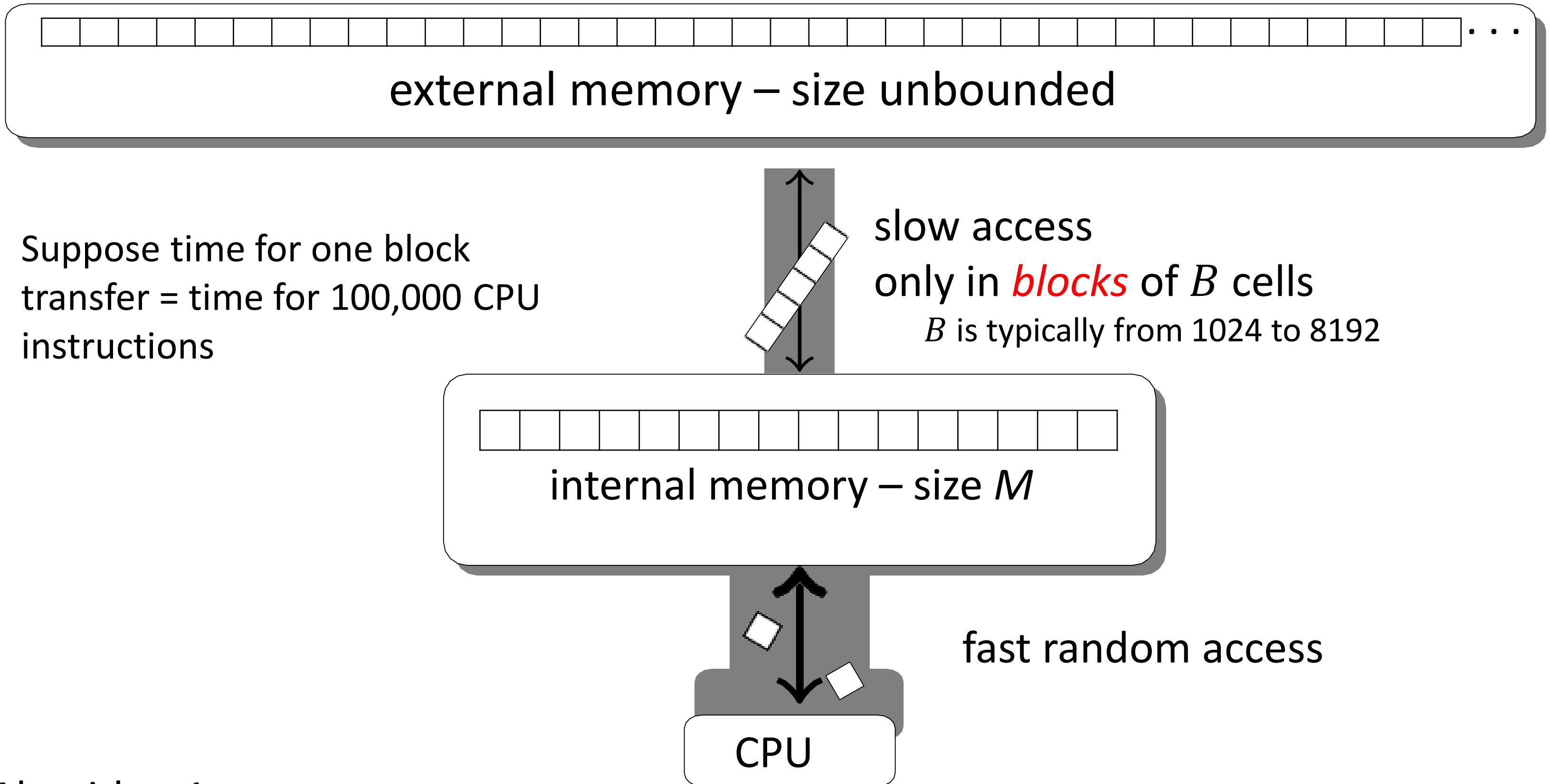
Outline

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

Different levels of memory

- 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
- To simplify, we focus on two 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

Adding 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 $= \cancel{1{,}000} + 1{,}000 \cdot 100{,}000 = \cancel{10^3} + 10^8$
- Algorithm 2
  ~~10,000 CPU instructions~~ + 10 block transfers $= \cancel{10{,}000} + 10 \cdot 100{,}000 = \cancel{10^4} + 10^6$

dominating factors

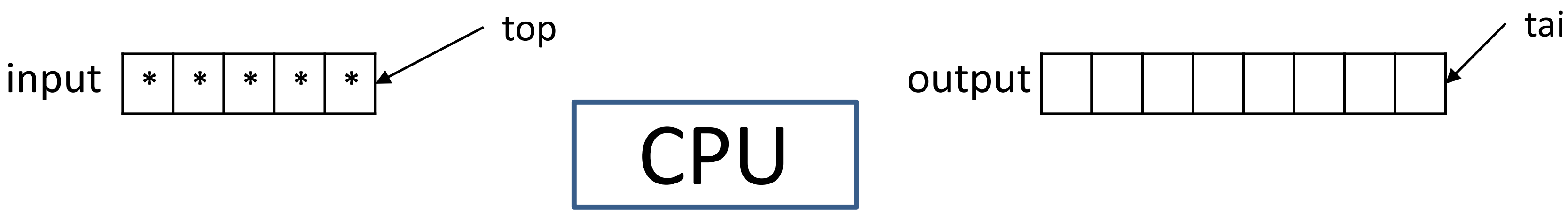
- **New cost of computation:** number of blocks transferred (or 'probes', 'disk transfers', 'page loads') between internal and external memory
- We will revisit ADTs/problems with the objective of minimizing block transfers

Outline

- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - (*a*, *b*)-Trees
    - B-Trees
  - Extendible Hashing

Stream Based Algorithms in Internal Memory

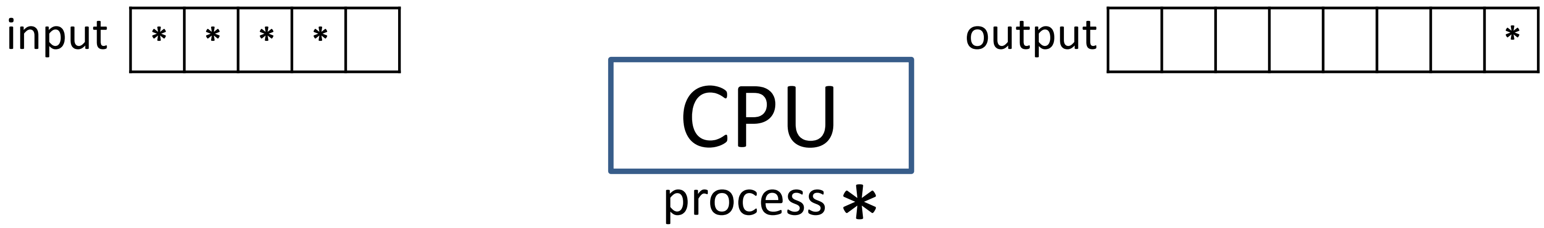
- We studied some algorithms that handle input/output with streams
  - can access only the top item in input stream, can append only to tail of the output stream



- 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

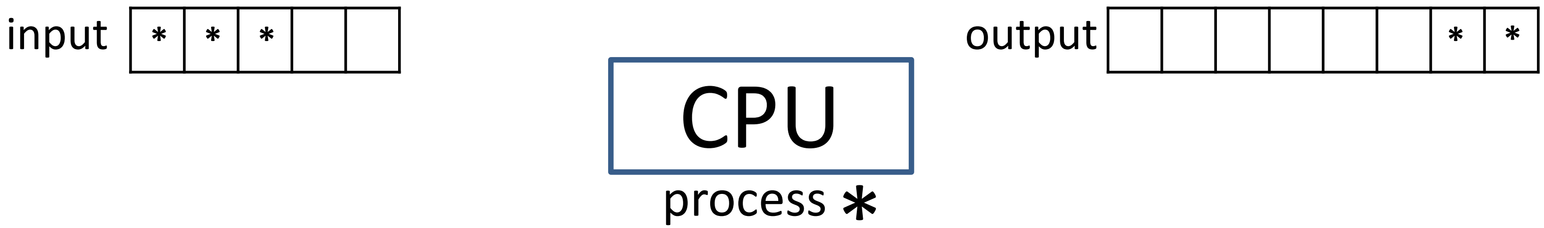
- We studied some algorithms that handle input/output with streams
  - can access only the top item in input stream, can append only to tail of the output stream



- 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

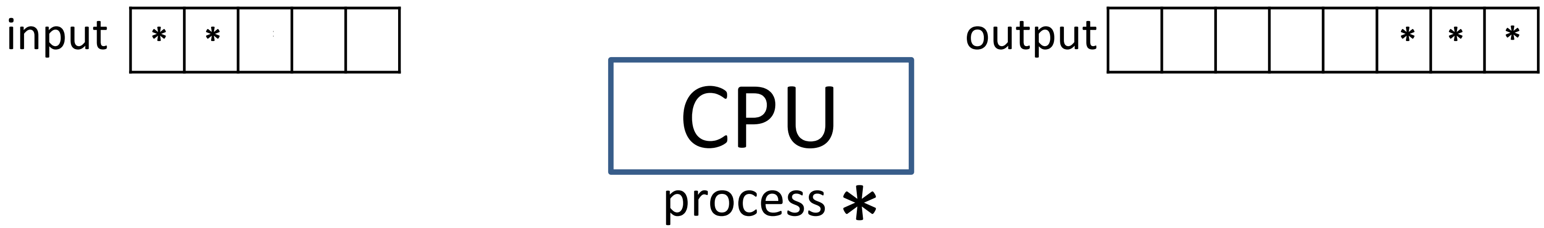
- We studied some algorithms that handle input/output with streams
  - can access only the top item in input stream, can append only to tail of the output stream



- 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

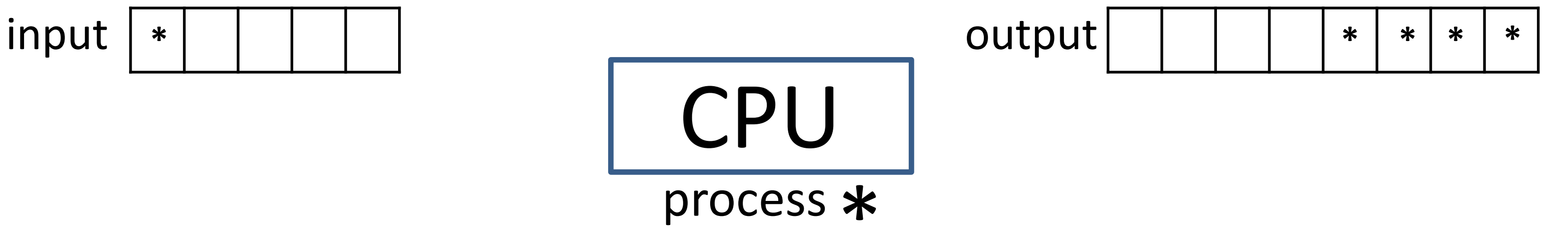
- We studied some algorithms that handle input/output with streams
  - can access only the top item in input stream, can append only to tail of the output stream



- 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

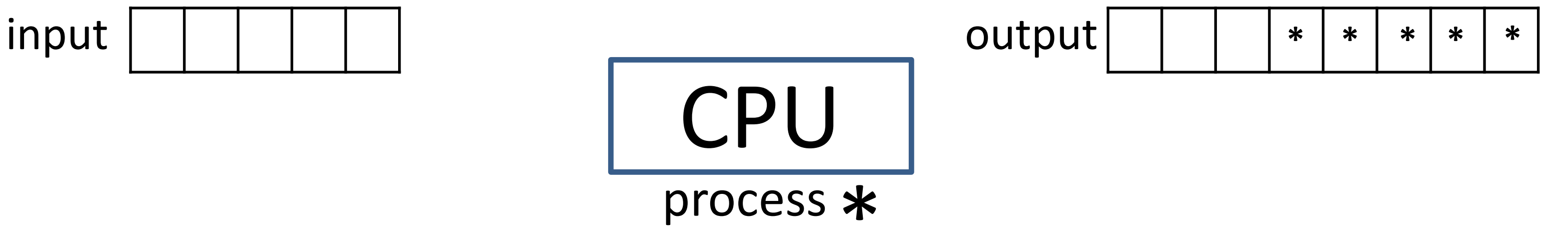
- We studied some algorithms that handle input/output with streams
  - can access only the top item in input stream, can append only to tail of the output stream



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

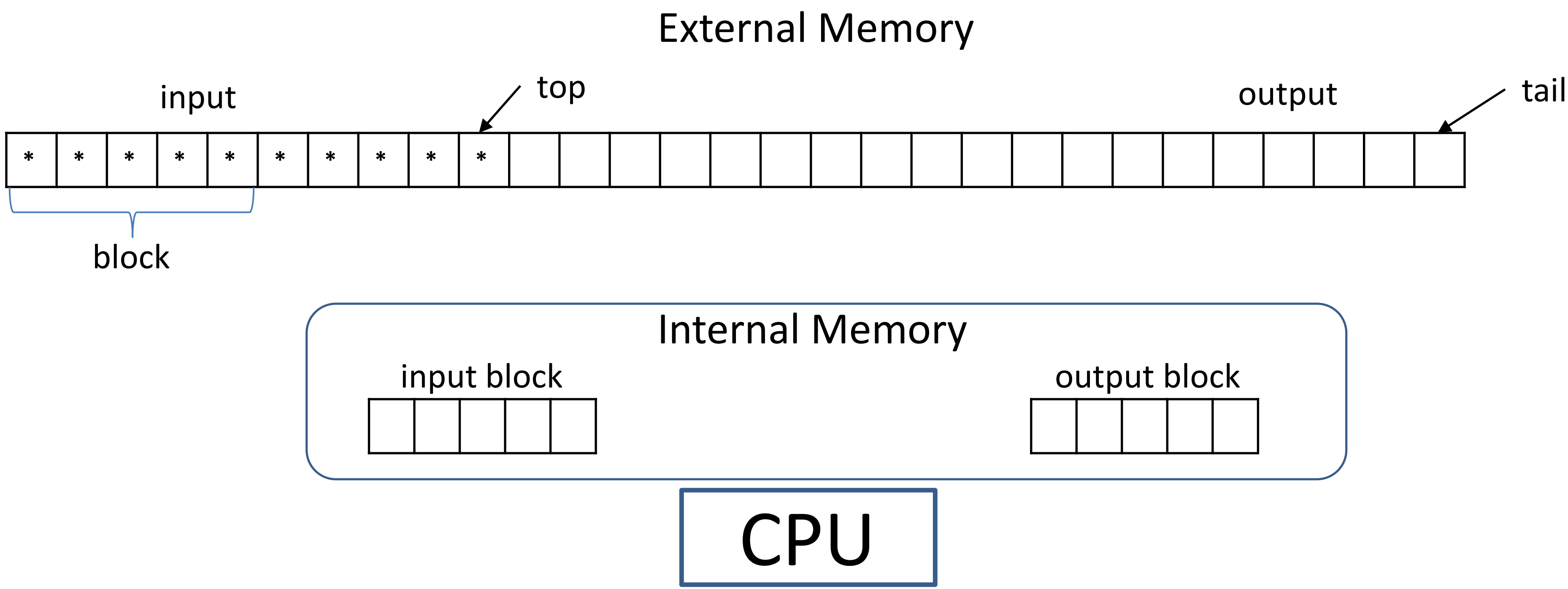
Stream Based Algorithms in Internal Memory

- We studied some algorithms that handle input/output with streams
    - can access only the top item in input stream, can append only to tail of the output stream



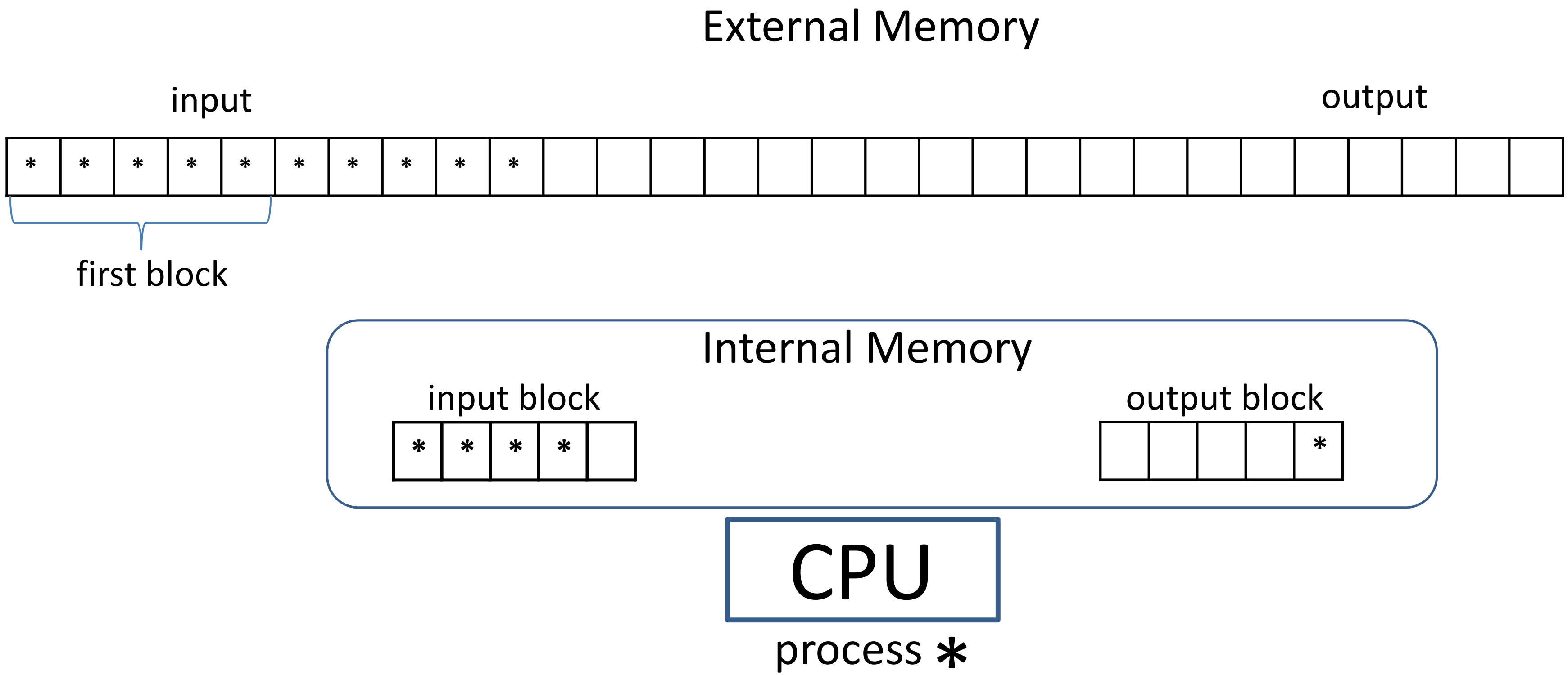
- Repeat
    1. take item off top of the input
    2. process item
    3. put the result of processing at the tail of output

Stream Based Algorithms in External Memory

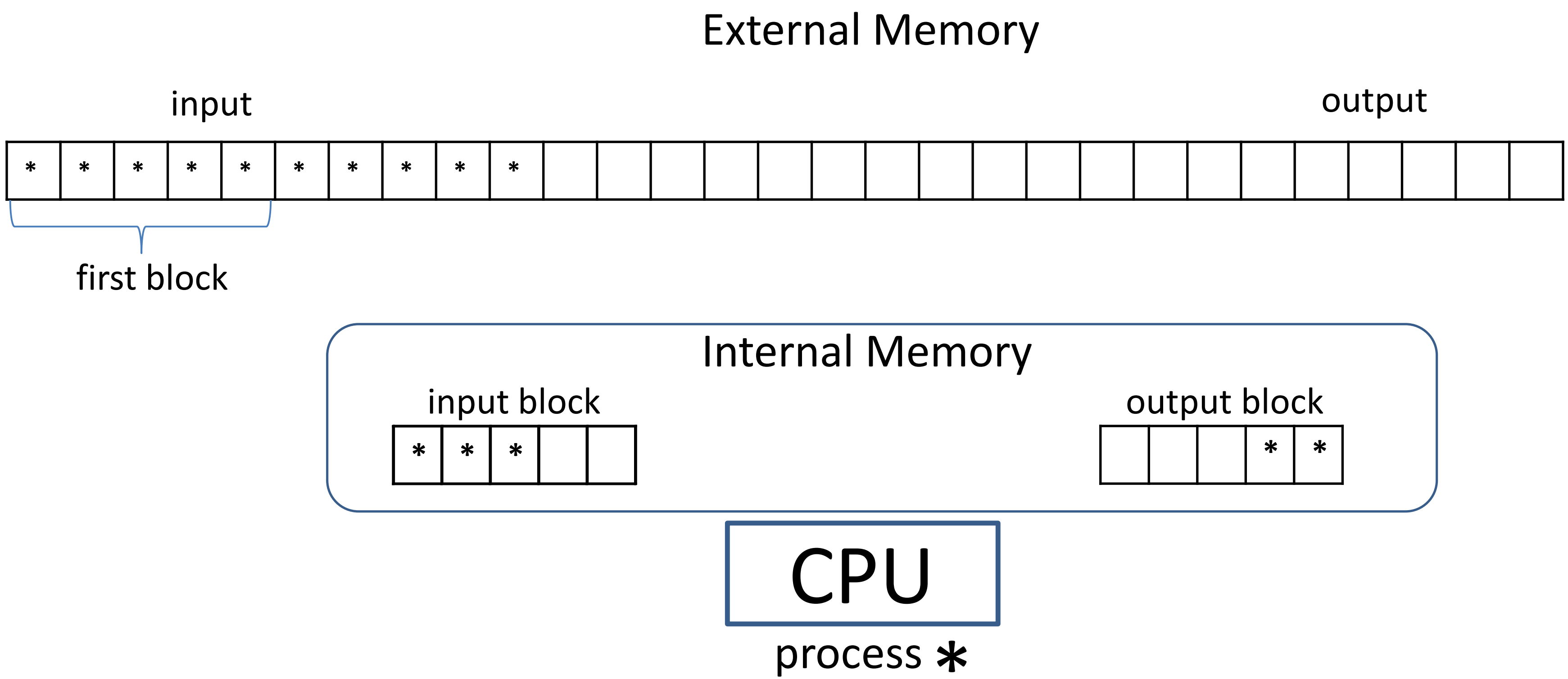


- Data in external memory has to be placed in internal memory before it can be processed
- Idea: perform the same algorithm as before, but in "block-wise" manner
  - have one block for input, one block for output in internal memory
  - transfer a block (size B) to internal memory, process it as before, 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

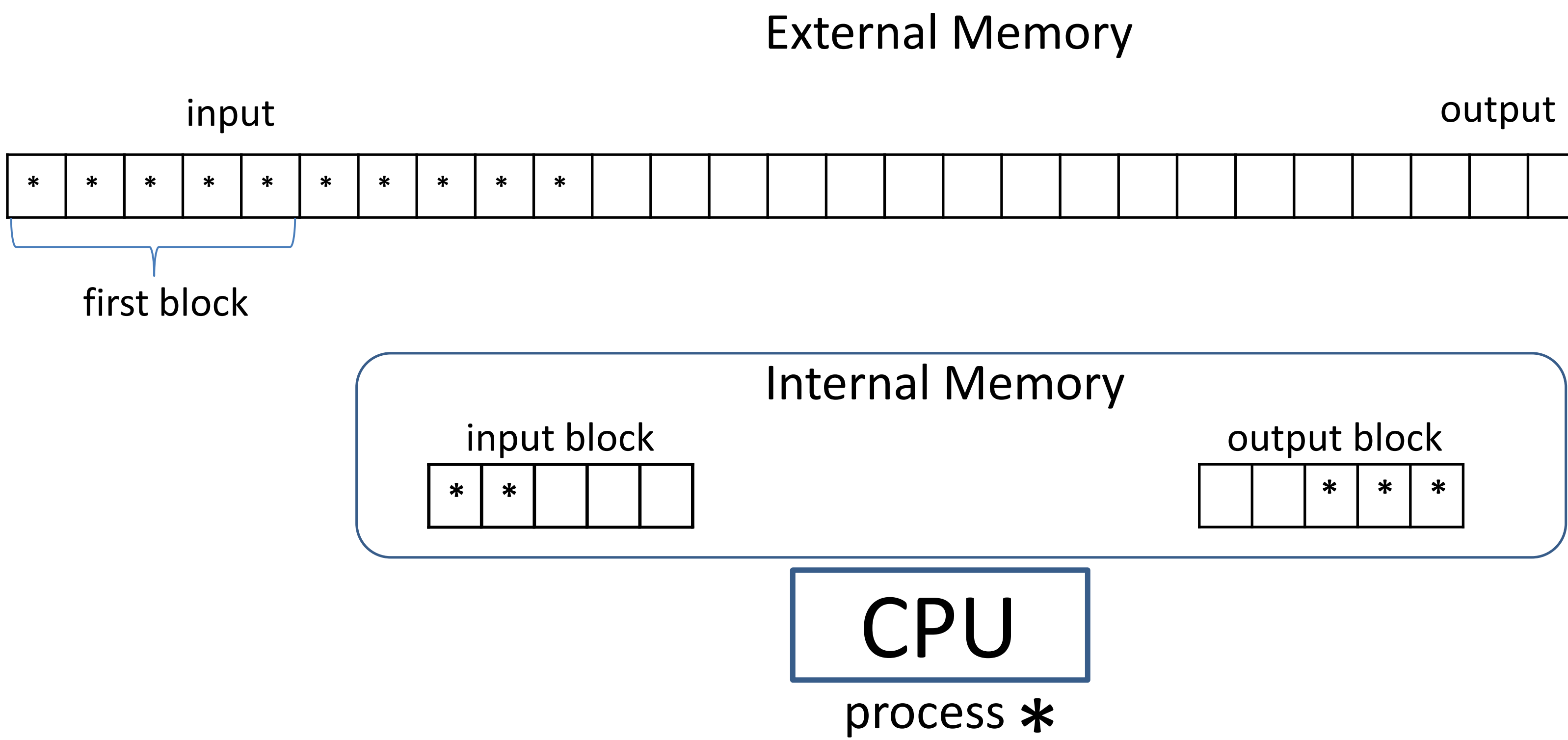
Stream Based Algorithms in External Memory



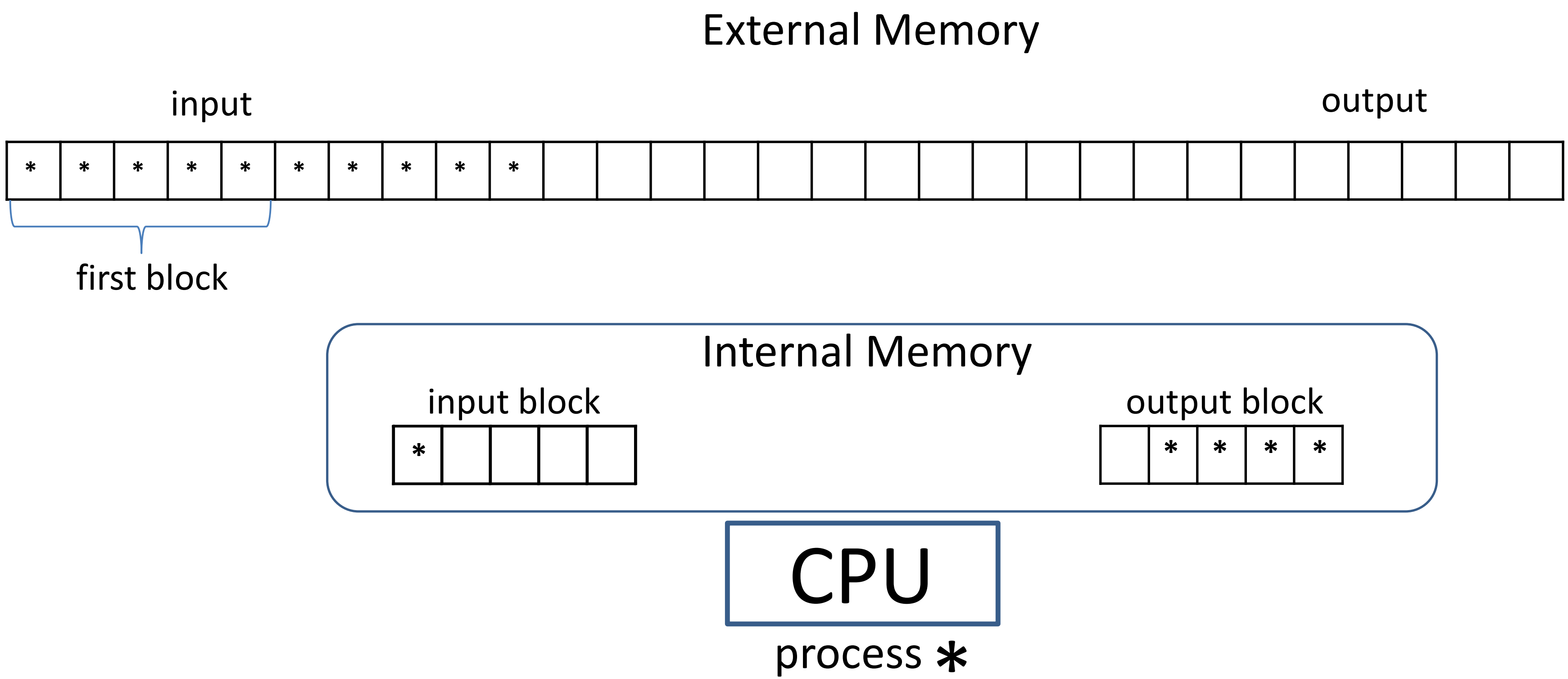
Stream Based Algorithms in External Memory



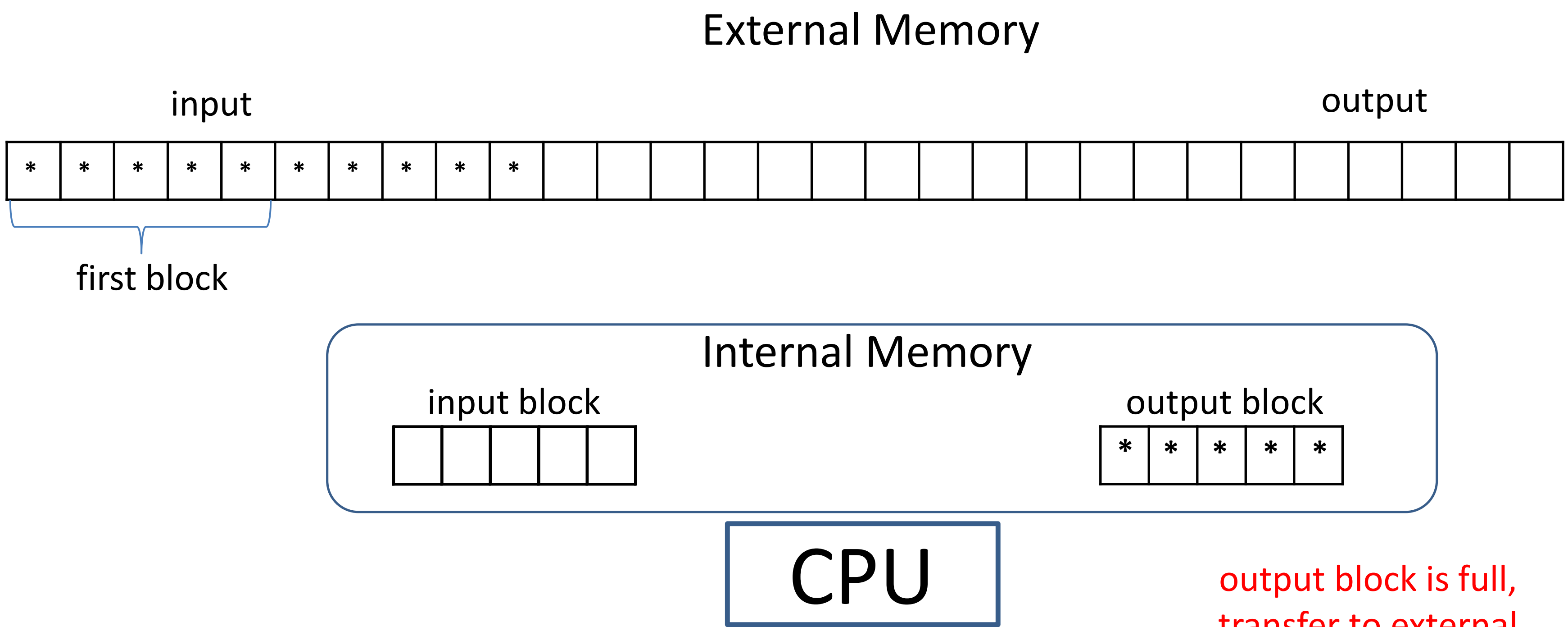
Stream Based Algorithms in External Memory



Stream Based Algorithms in External Memory

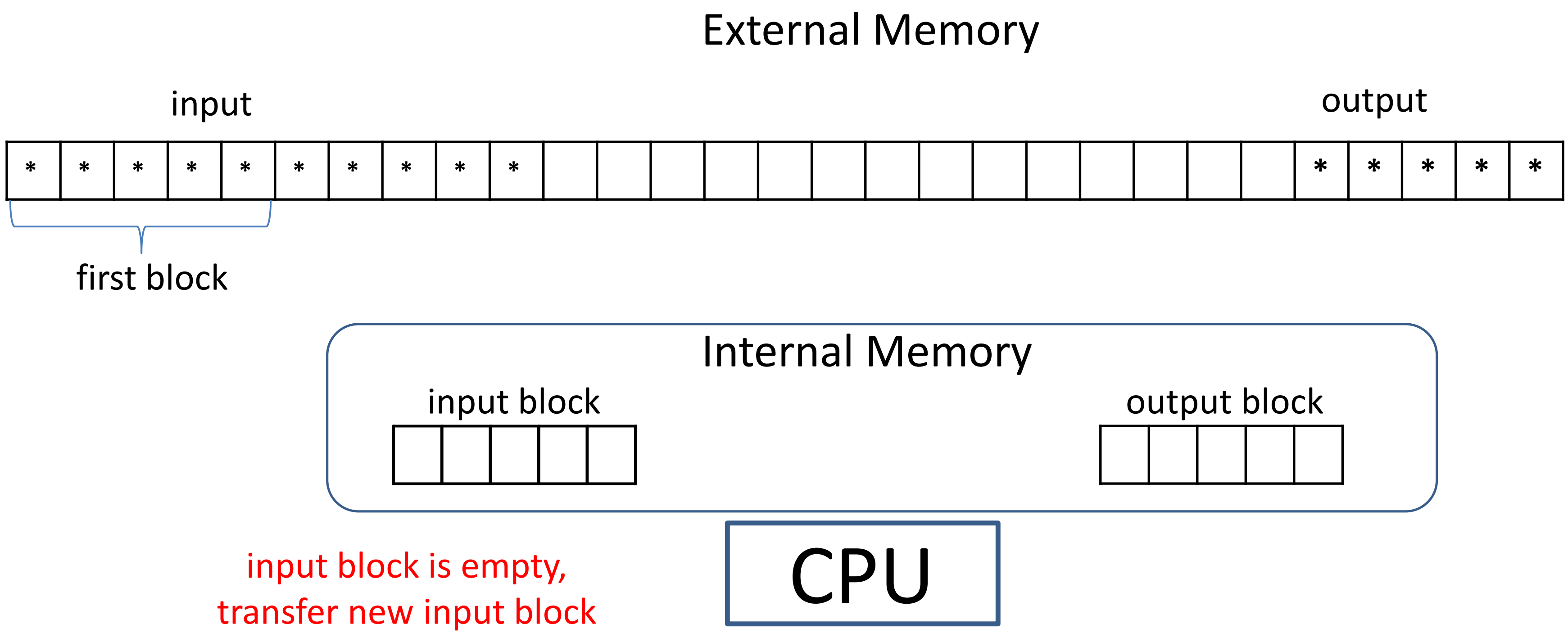


Stream Based Algorithms in External Memory



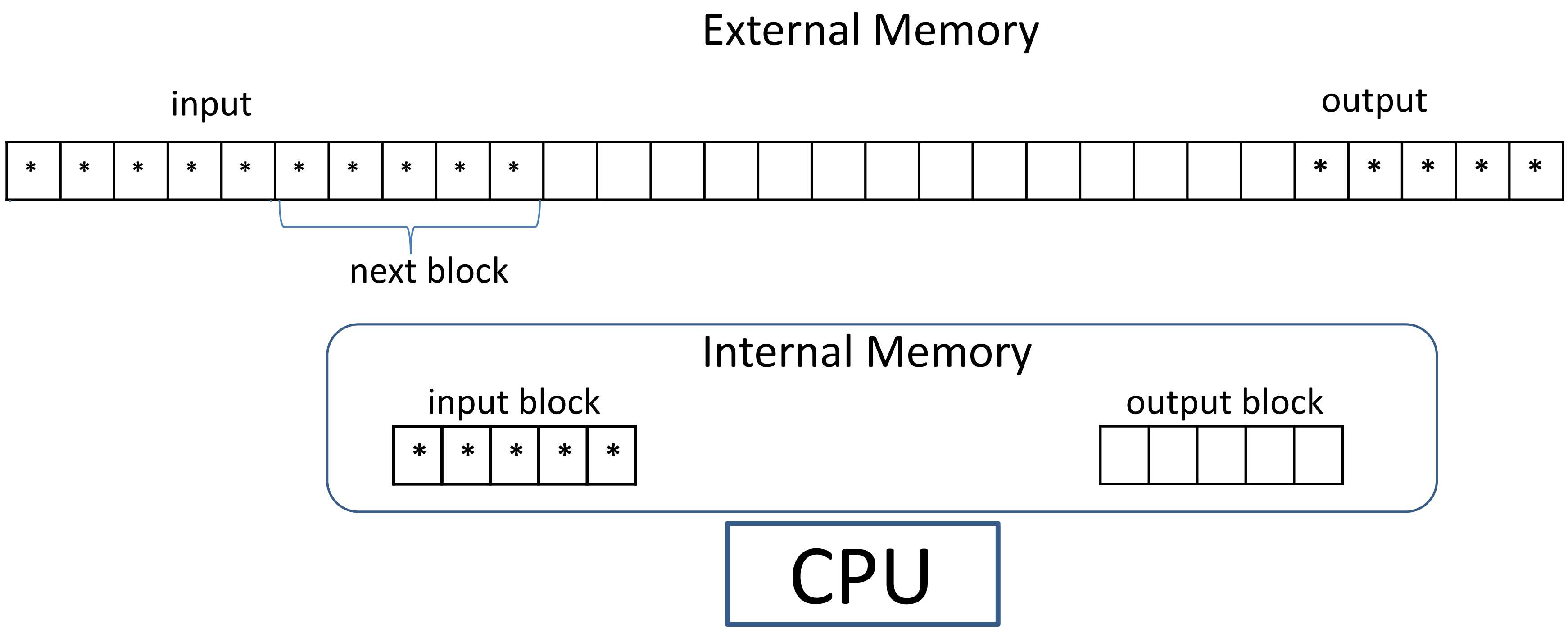
output block is full, transfer to external memory

Stream Based Algorithms in External Memory

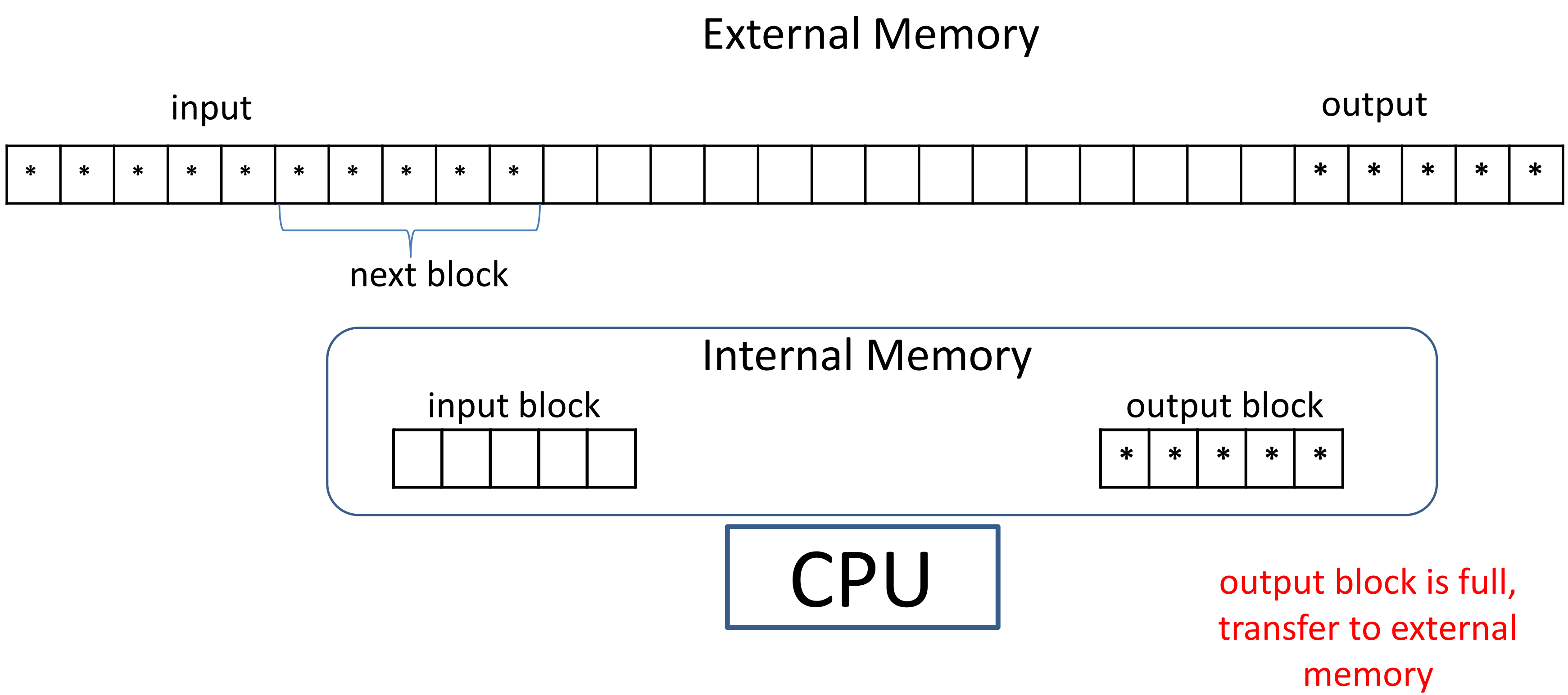


input block is empty,
transfer new input block
from external memory

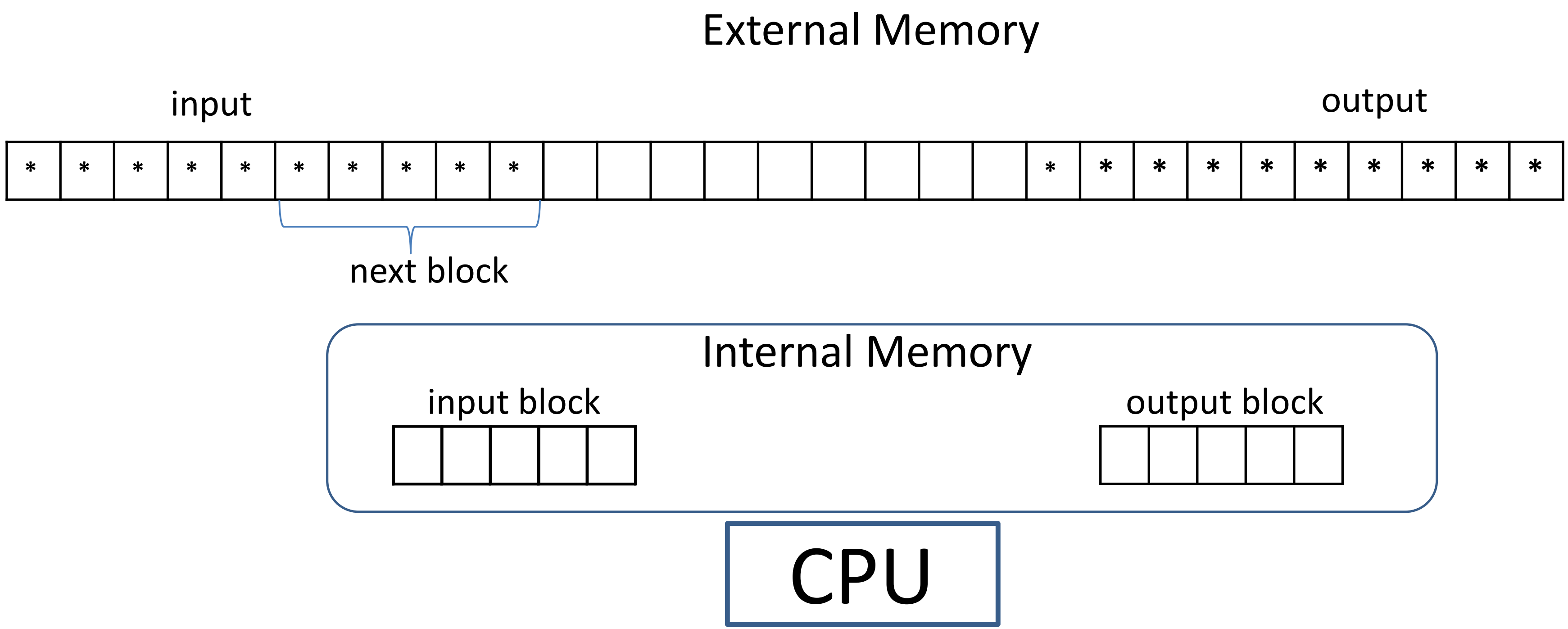
Stream Based Algorithms in External Memory



Stream Based Algorithms in External Memory



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

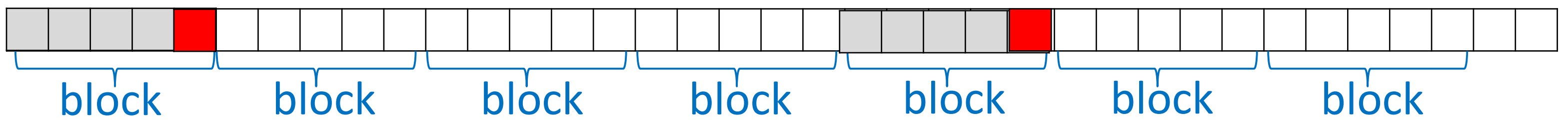
- Methods below use stream input/output model, therefore need $\Theta\left(\frac{n}{B}\right)$ block transfers, assuming output size is $O(n)$
  - Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore
    - assuming pattern P fits into internal memory
  - Text compression: Huffman, run-length encoding, Lempel-Ziv-Welch

Outline

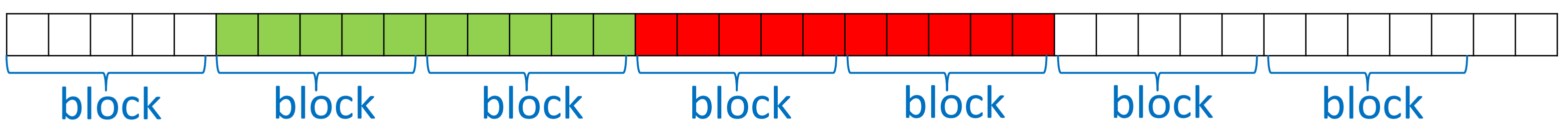
- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - (*a*, *b*)-Trees
    - B-Trees
  - Extendible Hashing

Sorting in external memory

- Sort array A of n numbers
  - n is huge so that A is stored in blocks in external memory
- Heapsort was optimal in time and space in RAM model
  - poor memory locality: accesses indices of A that are far apart

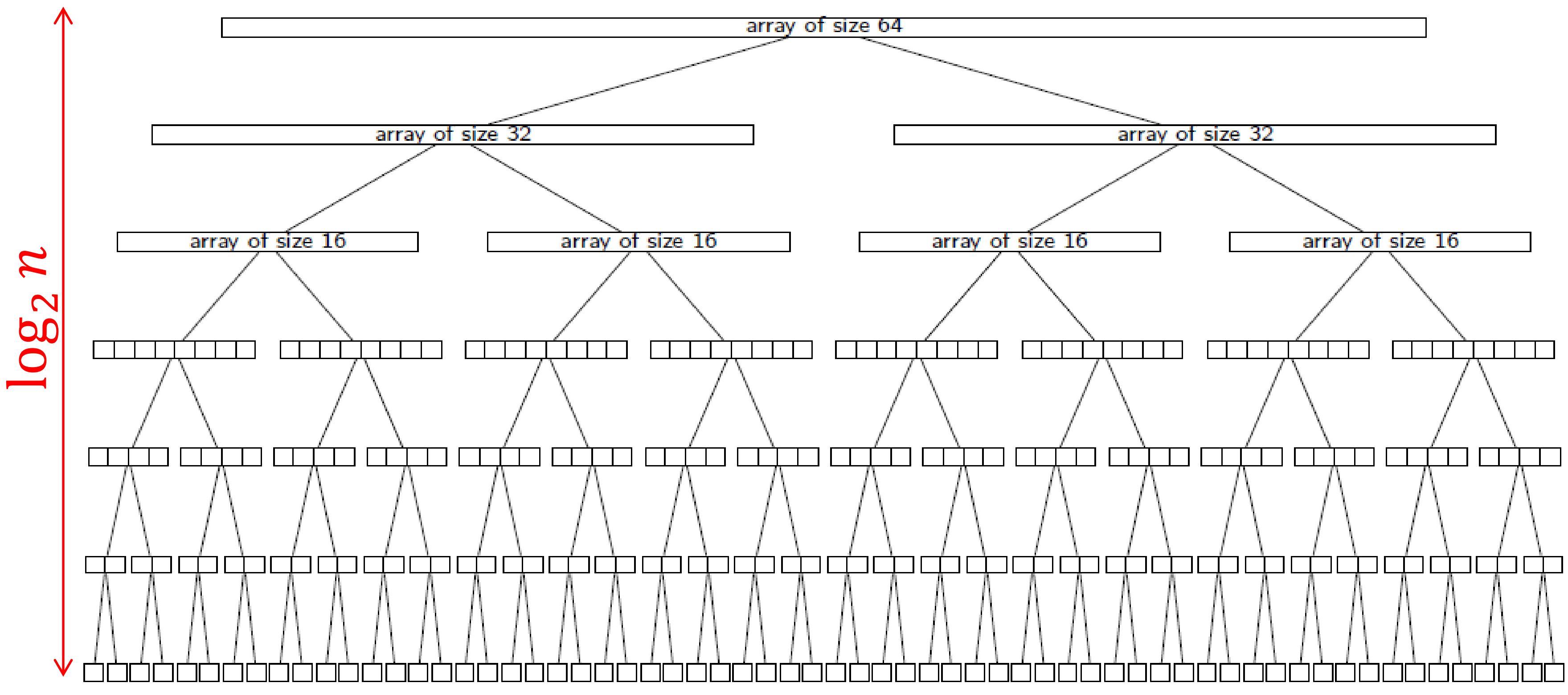


  - typically one block transfer per array access
    - access 2 blocks, but need only 2 elements in these blocks
    - all other data read in these 2 blocks is not used
  - does not adapt well to data stored in external memory, $\Theta(n \log n)$ block transfers
- Mergesort adapts well to array stored in external memory
  - based on merging already sorted parts of the array
  - access consecutive locations of A, ideal for reading in blocks



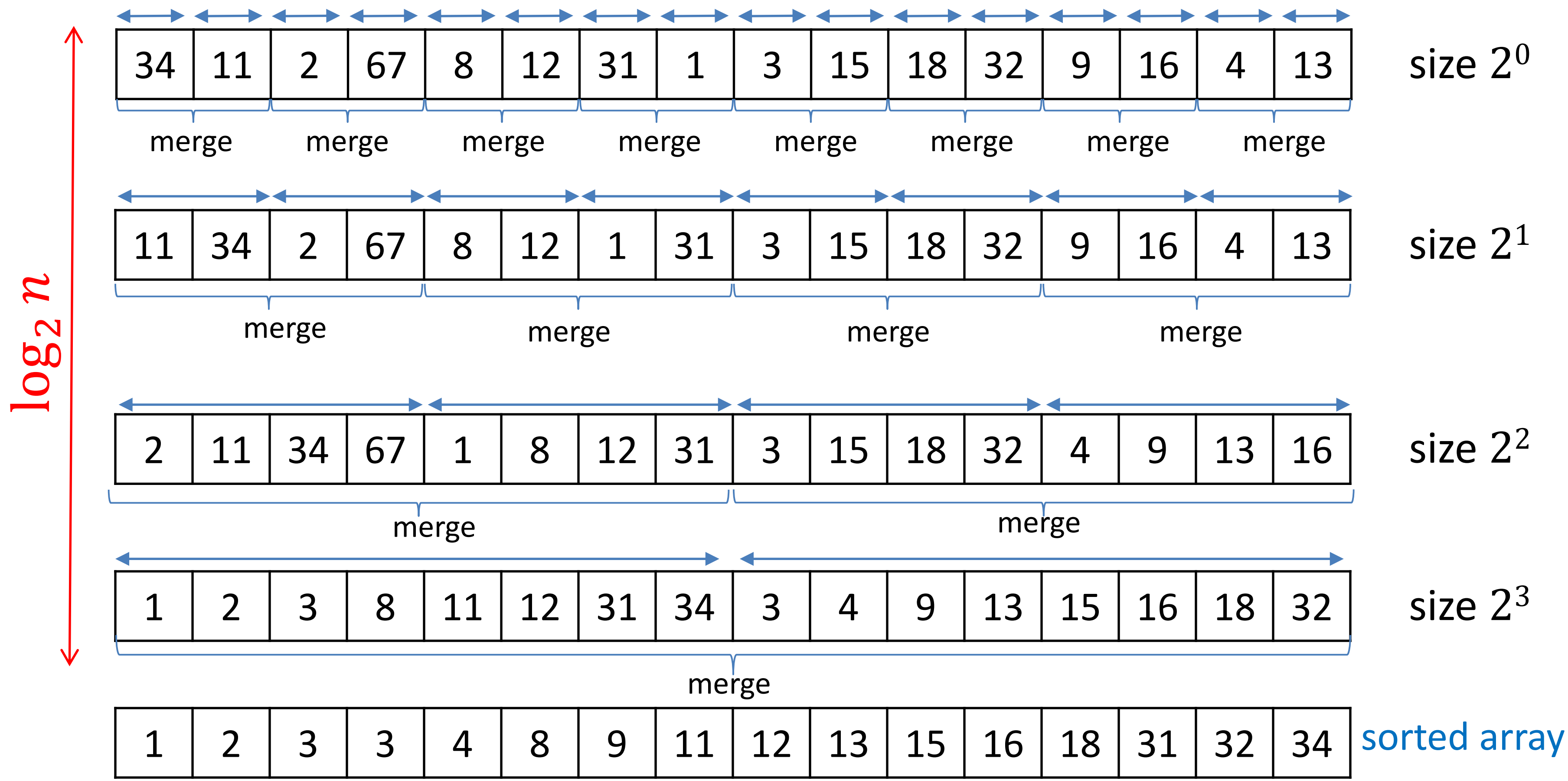
  - **key idea**: merge can be done with streams

Recall Mergesort



Mergesort: non-recusive Version

- Proceed bottom-up with while loops, rather than top-down with recursion
- Several rounds of merging adjacent pairs of sorted *runs* (run = subarray)
    - in round i, merge sorted runs of size 2^i
- Graphical notation sorted run



Merging with Streams in External Memory

Merge(S_1, S_2, S)

S_1, S_2 are input streams in sorted order, S is output stream

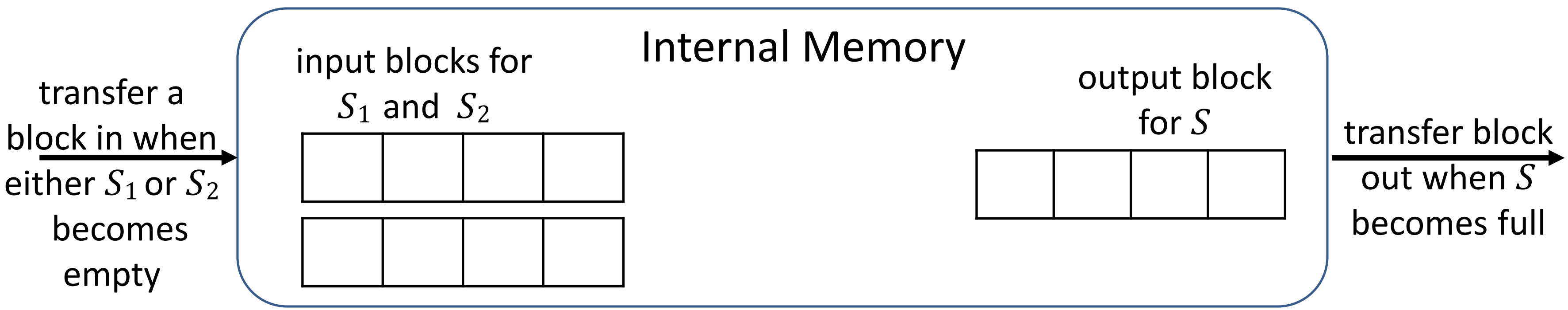
while S_1 or S_2 is not empty **do**

if S_1 is empty S.append$(S_2.pop(\,))$

else if S_2 is empty S.append$(S_1.pop(\,))$

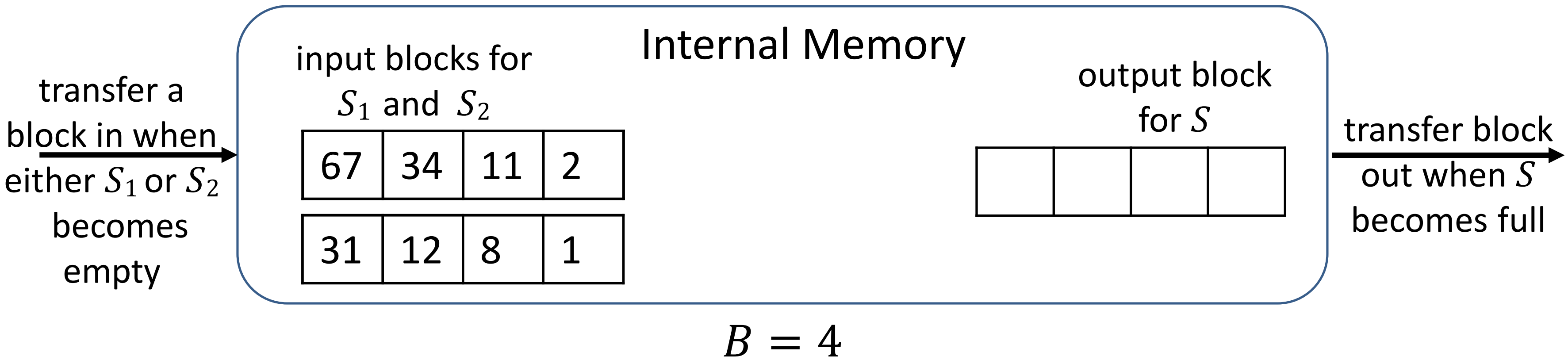
else if $S_1.top(\,) < S_2.top(\,)$ S.append$(S_1.pop(\,))$

else S.append$(S_2.pop(\,))$

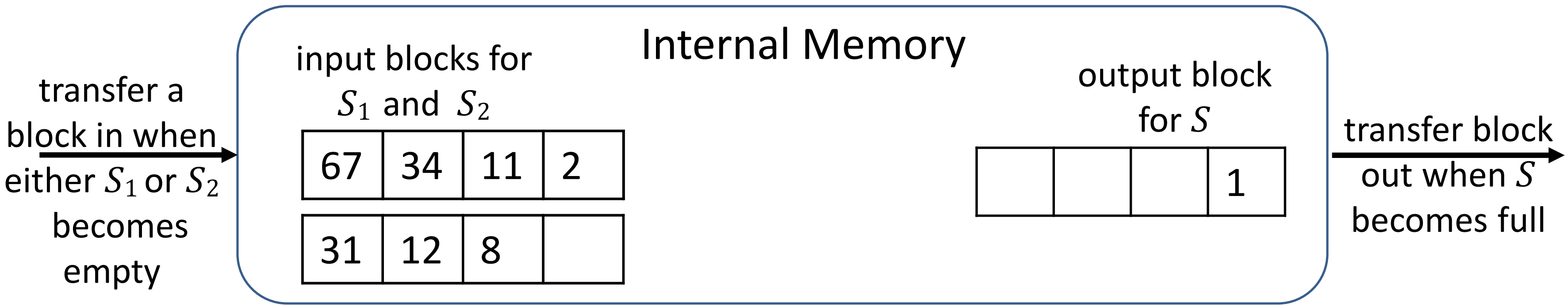


$B = 4$

Merging with Streams in External Memory

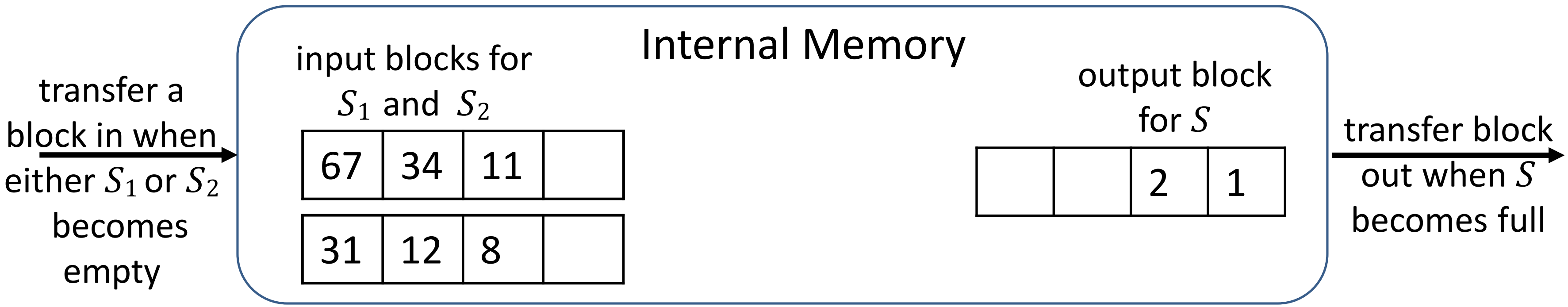


Merging with Streams in External Memory



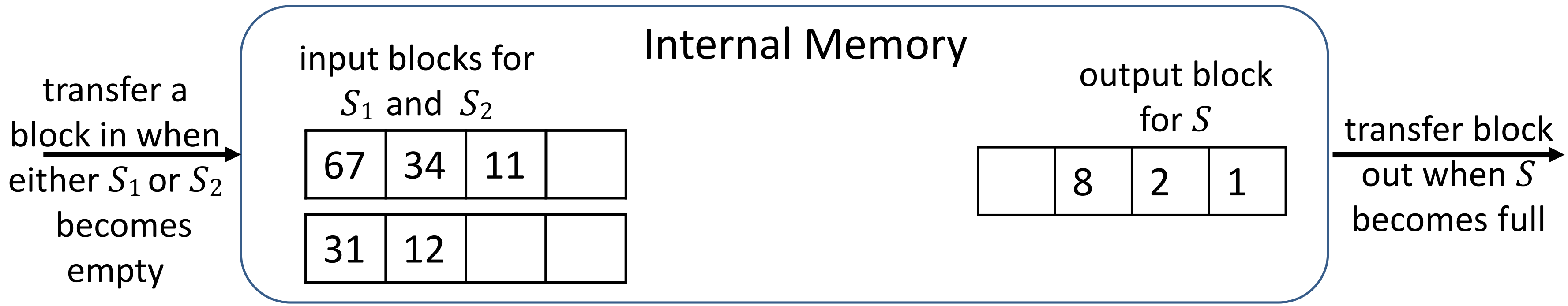
$B = 4$

Merging with Streams in External Memory



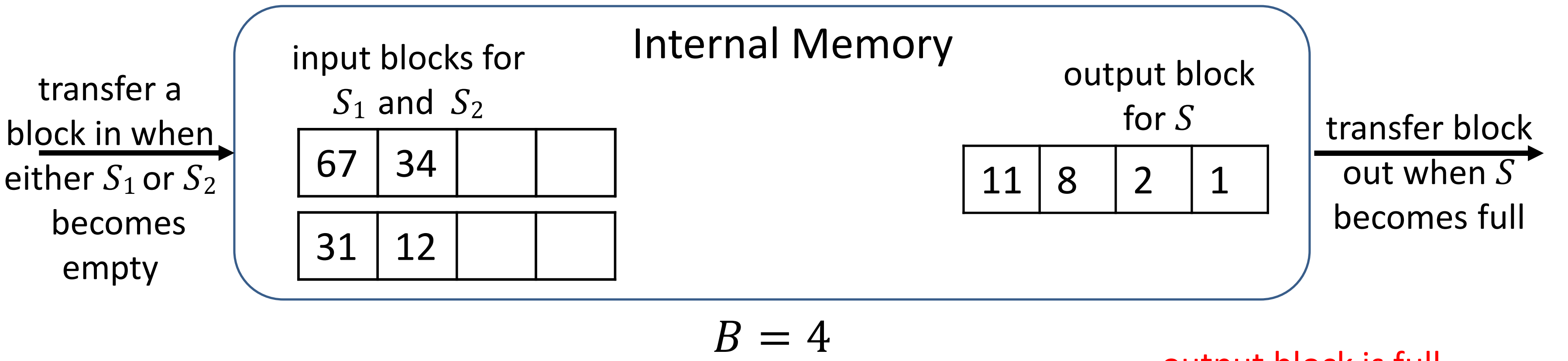
$B = 4$

Merging with Streams in External Memory



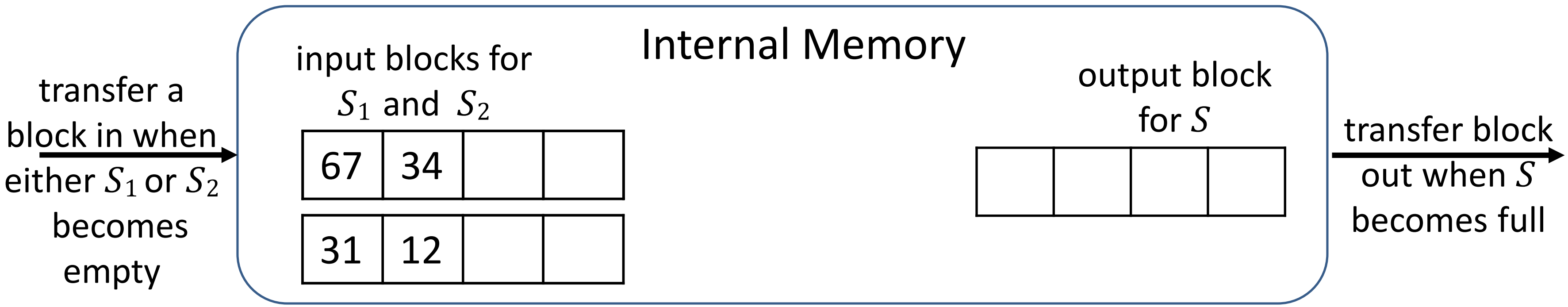
$B = 4$

Merging with Streams in External Memory



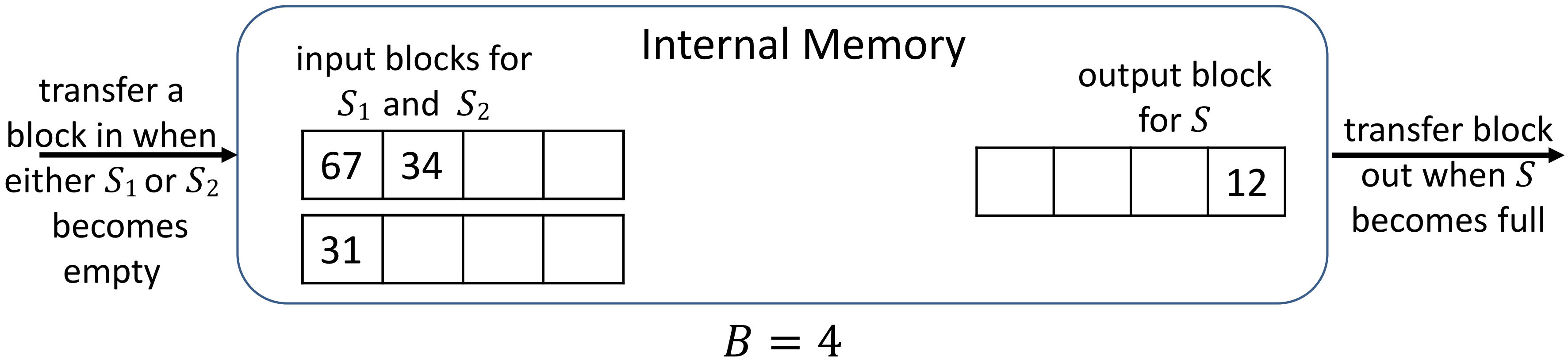
output block is full, transfer to external memory

Merging with Streams in External Memory

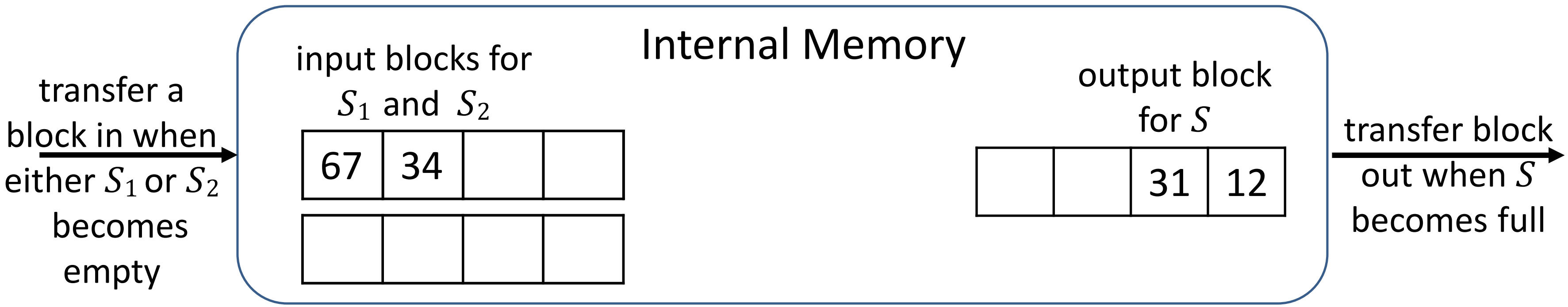


$$B = 4$$

Merging with Streams in External Memory



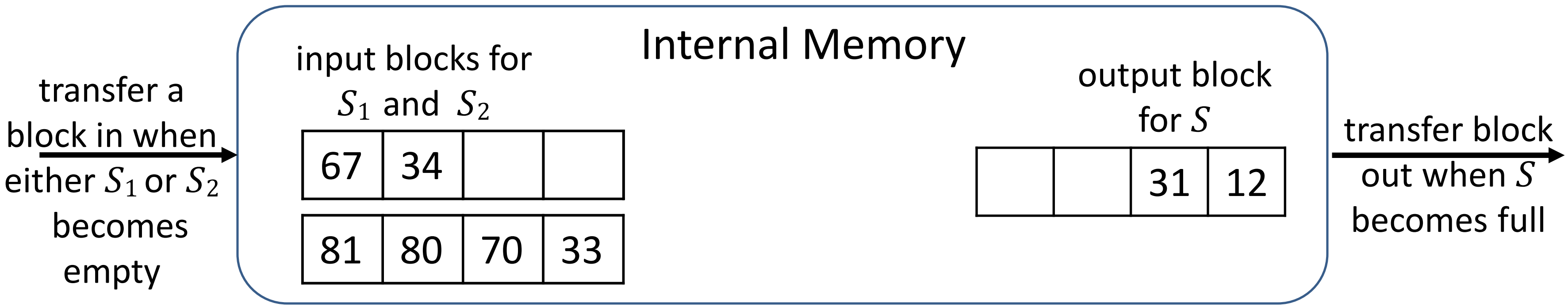
Merging with Streams in External Memory



$B = 4$

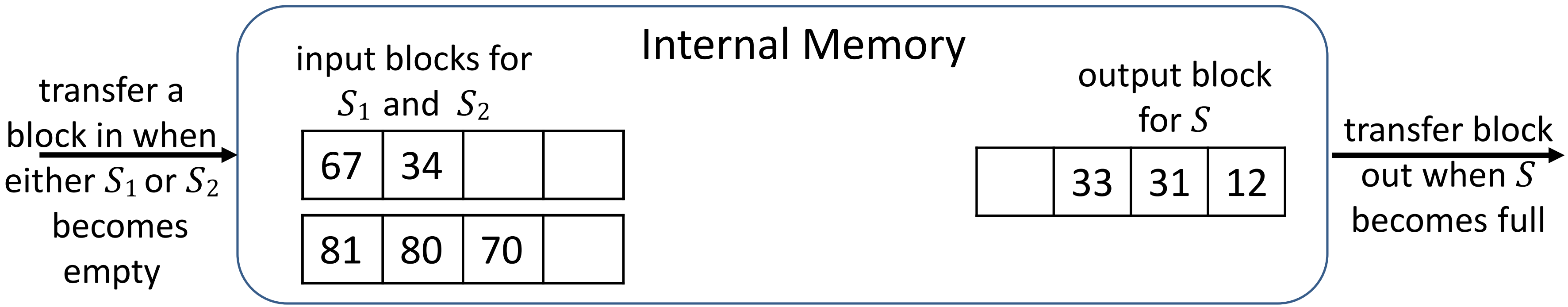
input block for S_2 is empty, transfer next block for S_2 from external memory

Merging with Streams in External Memory



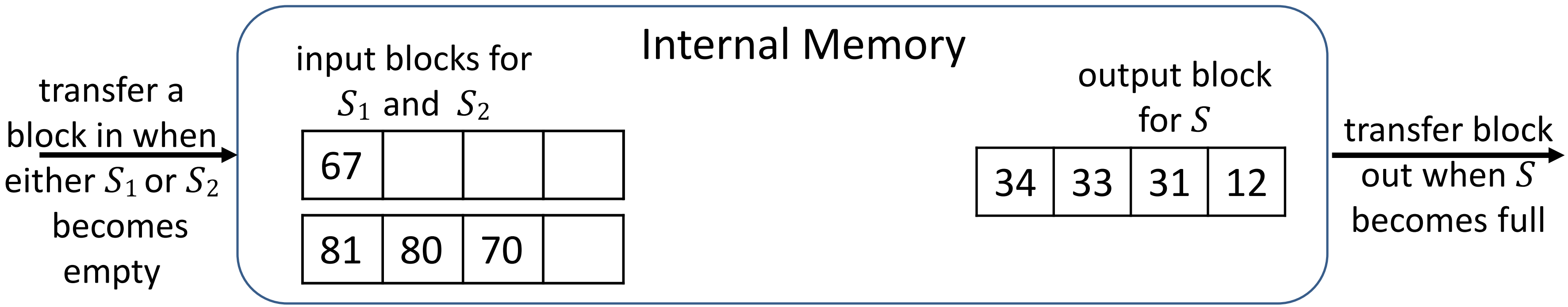
$B = 4$

Merging with Streams in External Memory



$B = 4$

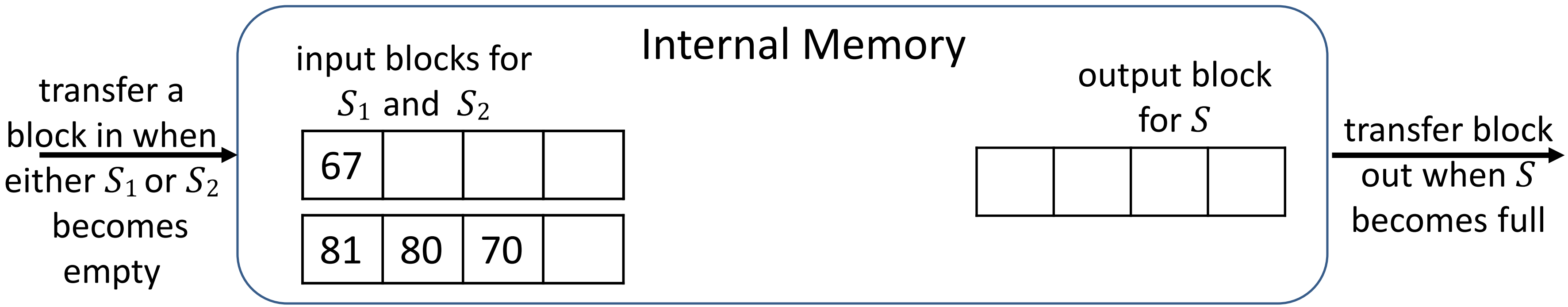
Merging with Streams in External Memory



$B = 4$

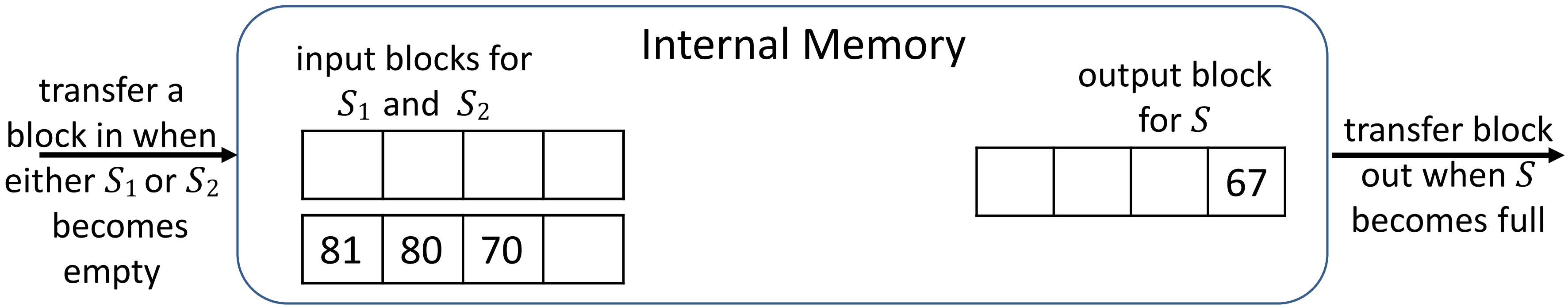
output block is full, transfer to external memory

Merging with Streams in External Memory



$B = 4$

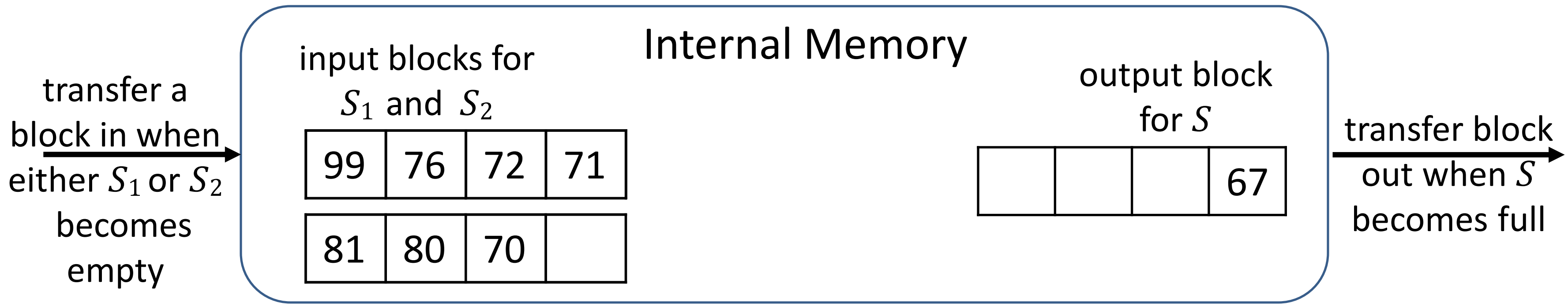
Merging with Streams in External Memory



$B = 4$

input block for S_1 is empty, transfer next block for S_1 from external memory

Merging with Streams in External Memory



$B = 4$

- *Merge* uses streams S_1, S_2, S
  - each block in the stream is transferred exactly once
- *Merge* takes $\frac{n}{B}$ block transfers for input streams and $\frac{n}{B}$ for output stream, total $\frac{2n}{B}$
- Recall that *MergeSort* uses $\log_2 n$ rounds of merging
- *MergeSort* run-time to sort is $\frac{2n}{B} \cdot \log_2 n$ block transfers
  - not bad but we can do better
  - idea: reduce the number of rounds
    - typically $M \gg 3B$, so can fit many blocks in the main memory
    - merge more than 2 sequences at a time!

d-way Mergesort

- Merge d sorted runs at one time
    - $d = 2$ gives standard mergesort
- Example: $d = 4$

$\log_d n$

34	11	2	67	8	12	31	1	3	15	18	32	9	16	4	13	size 4^0

d-way merge d-way merge d-way merge d-way merge

2	11	34	67	1	8	12	31	3	15	18	32	4	9	13	16	size 4^1

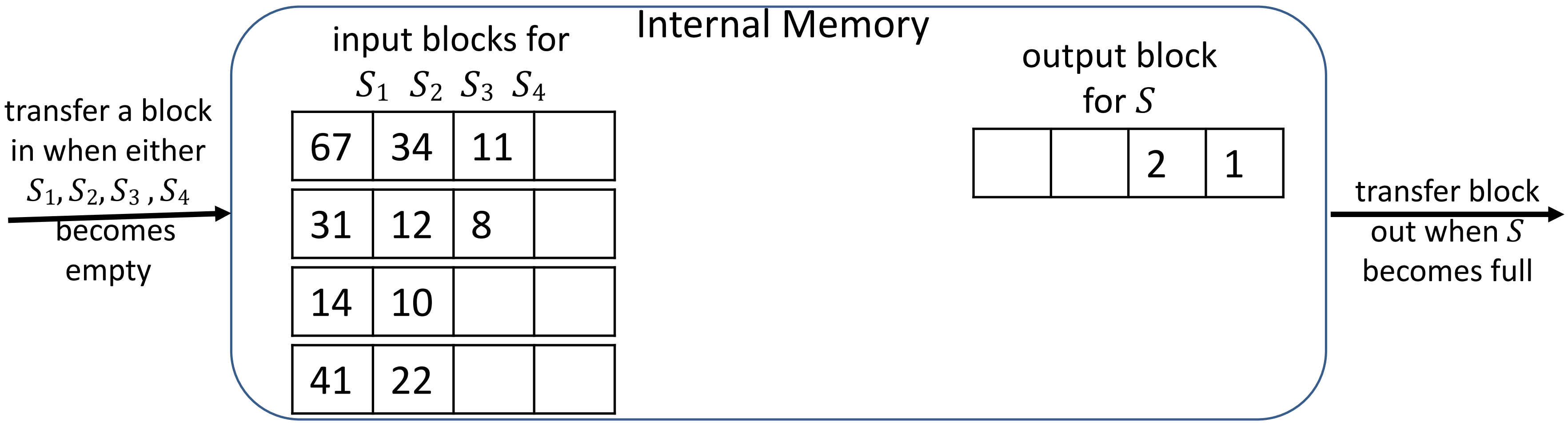
d-way merge

1	2	3	3	4	8	9	11	12	13	15	16	18	31	32	34	sorted array

- $\log_d n = \frac{\log_2 n}{\log_2 d}$ rounds
    - the larger is d the less rounds
    - each round still takes $\frac{2n}{B}$ of block transfers

d-way Mergesort

- Merge d sorted runs at once, and it still takes $\frac{2n}{B}$ of block transfers
- Let M be the size of the internal memory
- Choose d so that $d + 1$ blocks fit into internal memory
  - $d + 1 \approx \frac{M}{B}$



d-way Merge

- $d = 3$

2	11	34	8	9	12	1	11	31

- $d = 5$

2	11	34	8	9	12	1	11	31	15	18	32	9	12	13

- $d = 16$

34	11	2	67	8	12	31	1	3	15	18	32	9	16	4	13

- Need efficient data structure to find the minimum among d current tops
    - although it does not effect efficiency in terms of block transfers

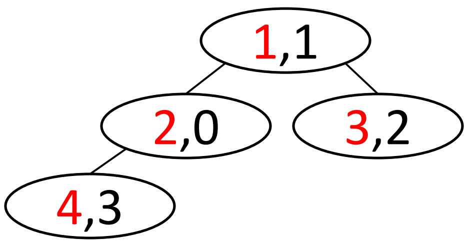
d-way Merge with Min-Heap

- Use min heap to find the smallest element among of d current tops
    - (key,value) = (element, sorted run)
- $d = 4$

2	11	34	67	1	8	12	31	3	15	18	32	4	9	13	16

merged output

1) insert(2,0), insert(1,1),
insert(3,2), insert(4,3)



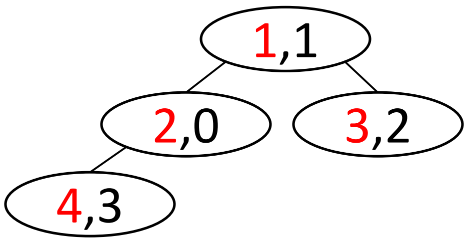
d-way Merge with Min-Heap

2	11	34	67	1	8	12	31	3	15	18	32	4	9	13	16

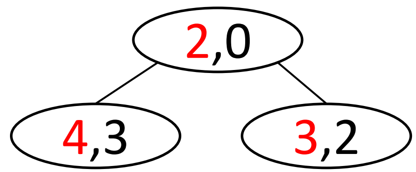
merged output

1															

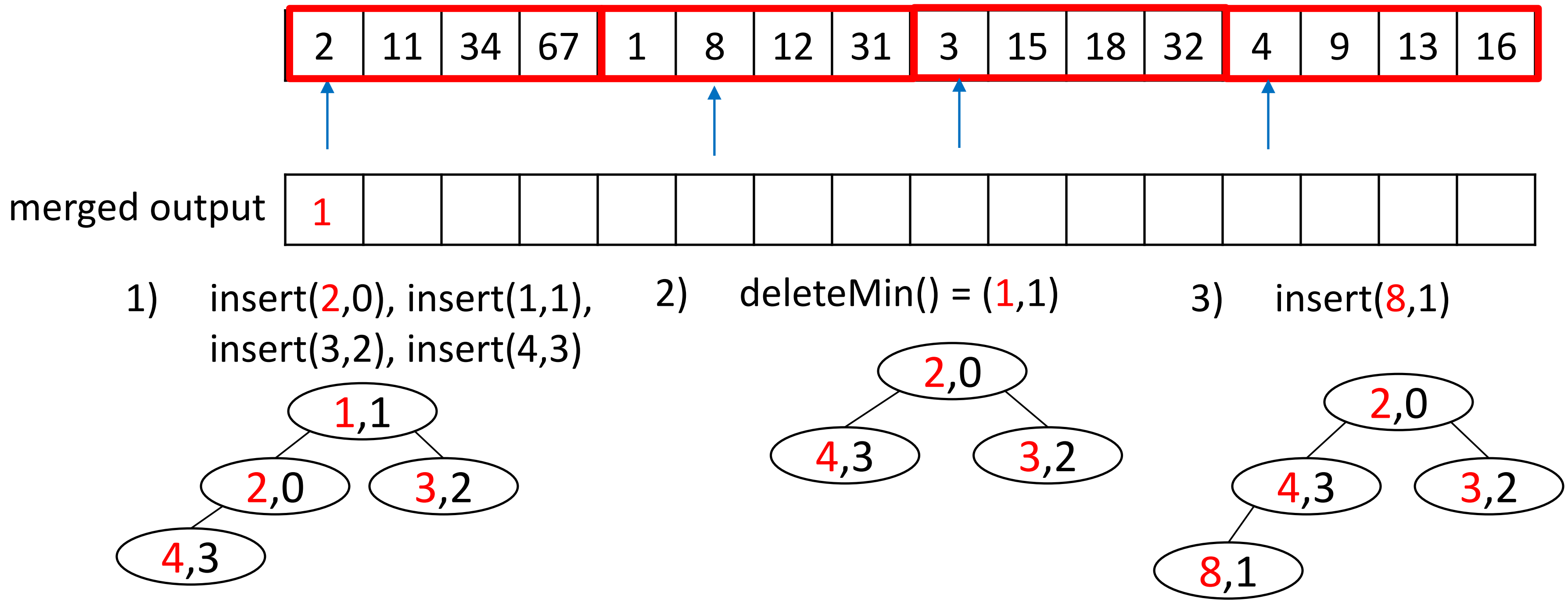
1) insert(2,0), insert(1,1), insert(3,2), insert(4,3)



2) deleteMin() = (1,1)

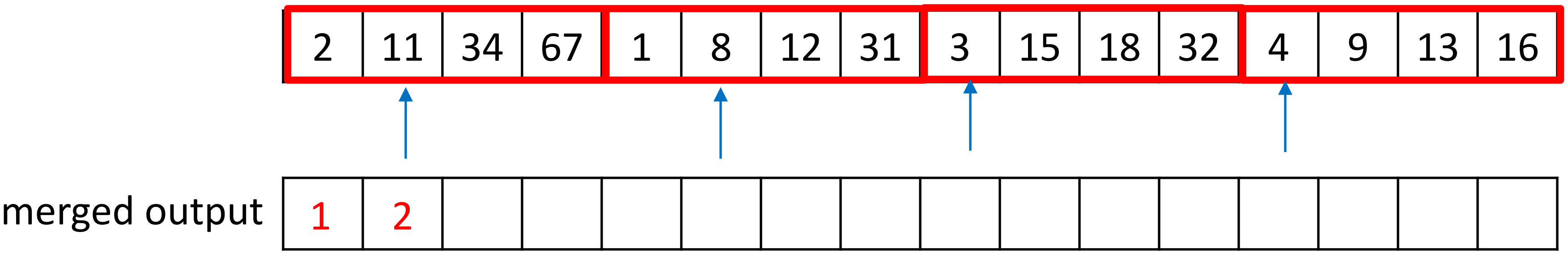


d-way Merge with Min-Heap

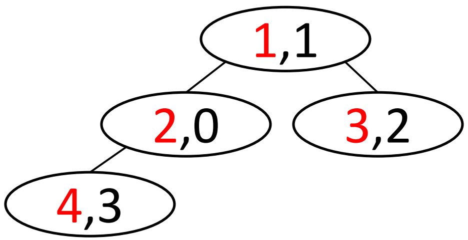


- Heap must have current fronts from all sorted runs
  - unless some sorted run ends

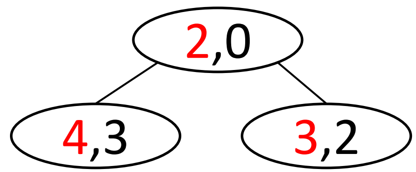
d-way Merge with Min-Heap



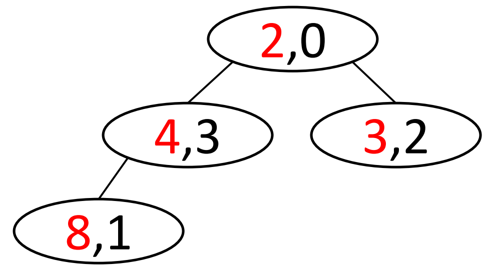
1) insert(2,0), insert(1,1), insert(3,2), insert(4,3)



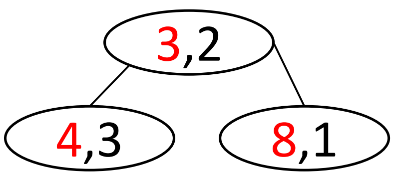
2) deleteMin() = (1,1)



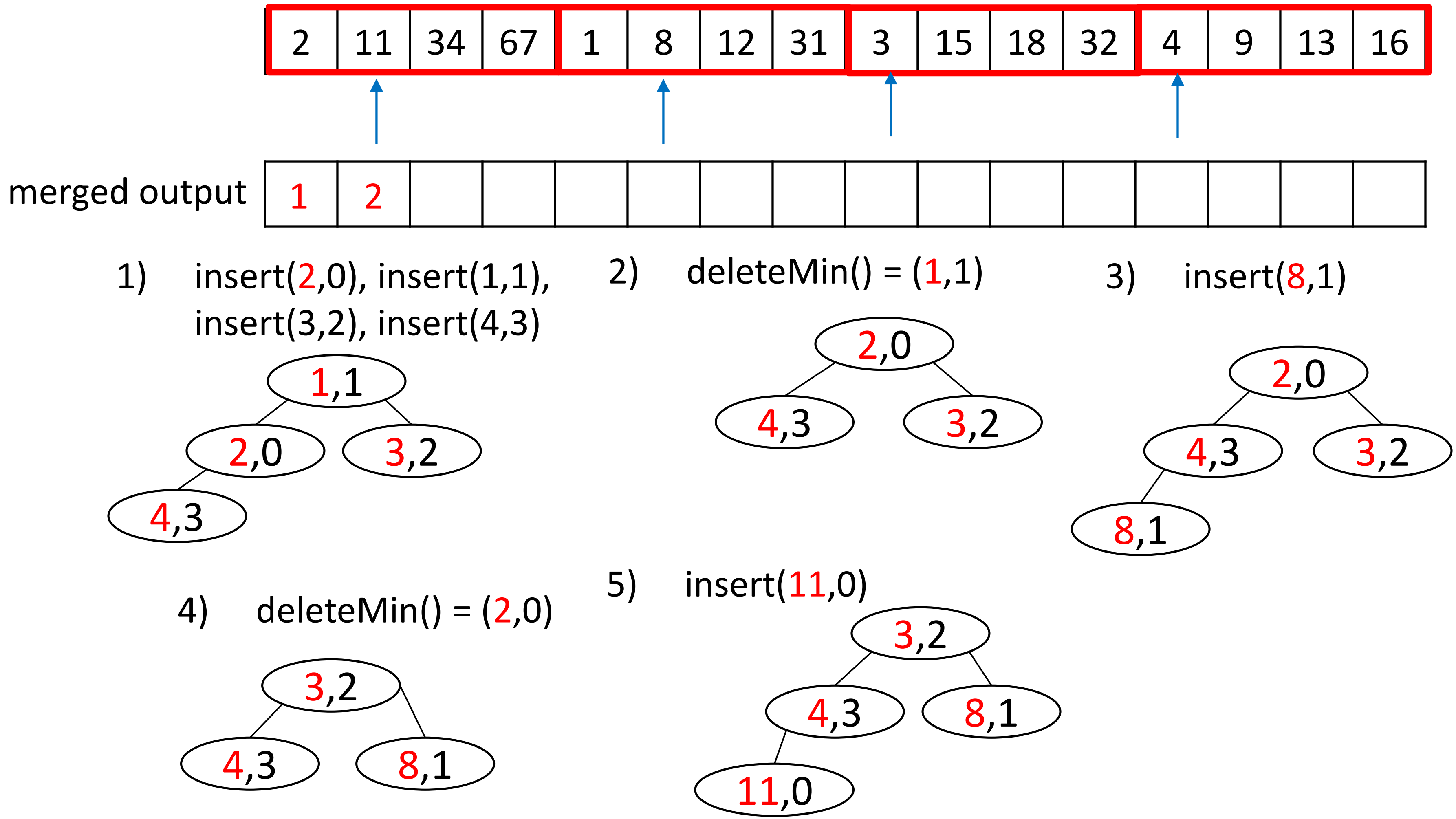
3) insert(8,1)



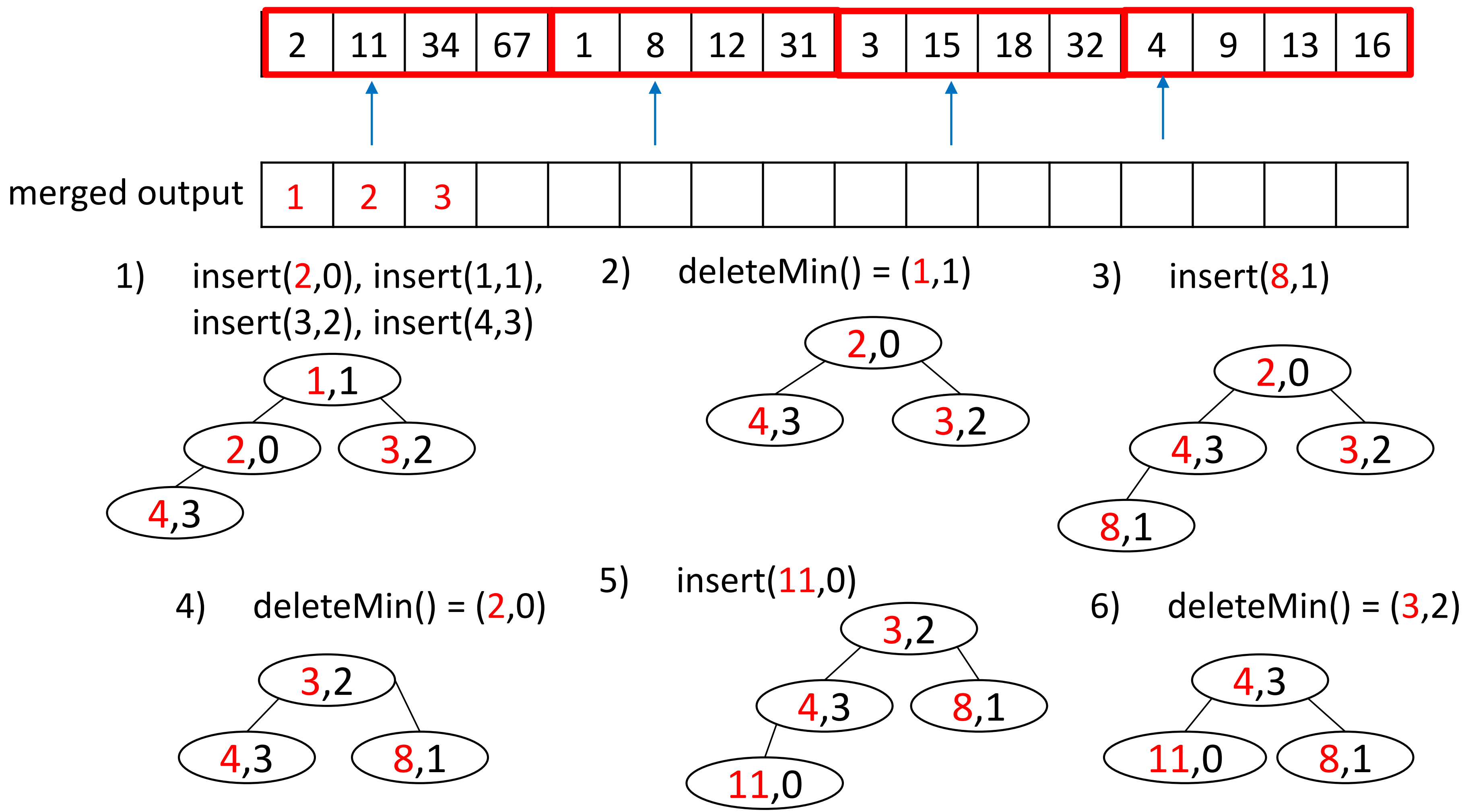
4) deleteMin() = (2,0)



d-way Merge with Min-Heap



d-way Merge with Min-Heap



d-way Merge with Min Heap Pseudo Code

d-Way-Merge$(S_1, \ldots, S_d, S)$

$S_1, \ldots, S_d$ are input sorted runs, each is a stream, S is output stream

$P \leftarrow$ empty min-priority queue

// P always holds current top elements of $S_1, \ldots, Sd$

$\Theta(d \log_2 d)$

for $i \leftarrow 1$ to d **do**

P.insert($S_i.top(\,), i$)

$\Theta(m \log_2 d)$

while P is not empty **do**

$(x, i) \leftarrow P$.*deleteMin*() // removes current top of S_i from P

S.append(x)

if S_i is not empty **do**

// current top of S_i is not represented in P, add it

P.insert$(S_i.top(\,), i)$

- Running time of operations in internal memory
  - priority queue P has size at most d at all times
  - while loop runs for m iterations, where $m = |S_1| + \cdots + |S_d|$
  - at each iteration
    - one *deleteMin*() on heap of size d, time is $\Theta(\log_2 d)$
    - one *insert*() on heap of size d, time is $\Theta(\log_2 d)$
  - Total time is $\Theta(m \log_2 d)$

d-way Merge with Min Heap Pseudo Code

```
d-Way-Merge(S_1, ..., S_d, S)
S_1, ..., S_d are input sorted runs, each is a stream, S is output stream
        P ← empty min-priority queue
        // P always holds current top elements of S_1, ..., Sd
Θ(d log_2 d)    for i ← 1 to d do
                        P.insert( S_i.top( ),, i)
Θ(m log_2 d)    while P is not empty do
                        (x, i) ← P.deleteMin() // removes current top of S_i from P
                        S.append(x)
                        if S_i is not empty do
                        // current top of S_i is not represented in P, add it
                                P.insert(S_i.top( ), i)
```

- Running time of operations in internal memory
  - priority queue P has size at most d at all times
  - while loop runs for m iterations, where $m = |S_1| + \cdots + |S_d|$
  - at each iteration
    - one *deleteMin*() on heap of size d, time is $\Theta(\log_2 d)$
    - one *insert*() on heap of size d, time is $\Theta(\log_2 d)$
  - Total time is $\Theta(m \log_2 d)$
- Number of block transfers is $\frac{2m}{B}$, assuming $d+1$ blocks and P fit into main memory

One Round of d-way Mergesort Running time

- In internal memory, d-way merge is $\Theta(m \log_2 d)$
    - m is the total number of elements in d sorted runs
- We need to d-way merge multiple number of times for one round of d-way Mergesort

34	11	2	67	8	12	31	1	3	15	18	32	9	16	4	13

d-way merge m_1 elements | d-way merge m_2 elements | d-way merge m_3 elements | d-way merge m_4 elements

- let m_1 be the number of elements in the first set of d sorted runs we merge
    - time to merge is $\Theta(m_1 \log_2 d)$
- let m_2 be the number of elements in the second set of d sorted runs we merge
    - time to merge is $\Theta(m_2 \log_2 d)$
- ……….
- let m_k be the number of elements in the last set of d sorted runs we merge
    - time to merge is $\Theta(m_k \log_2 d)$
- Total time to merge is

$$\Theta(m_1 \log_2 d + m_2 \log_2 d + \ldots + m_k \log_2 d) = \Theta((\underbrace{m_1 + m_2 + \cdots + m_k}_{n})\log_2 d)$$

    - where n is the size of the whole sequence
- The number of block transfers is $\frac{2n}{B}$

d-way Mergesort Complexity In Internal Memory

- $\log_d n$ rounds
- Running time for one round is $\Theta(n \log_2 d)$
- Total time $\Theta(\log_d n \cdot n \log_2 d) = \Theta\left(\frac{\log_2 n}{\cancel{\log_2 d}} \cdot \cancel{n \log_2 d}\right) = \Theta(n \log_2 n)$
- In internal memory, d-way merge sort has the same running time theoretically
  - in practice , d-way merge is slower due to the overhead of maintaining a heap

d-way Mergesort Complexity In External Memory

- Only block transfers count, each round is $\Theta\left(\frac{n}{B}\right)$ block transfers, no matter what d is
    - assuming d is such that $d+1$ blocks plus priority queue fit into internal memory
- $\log_d n$ rounds, time for each round is $\Theta\left(\frac{n}{B}\right)$ block transfers
- Total time $\Theta\left(\frac{n}{B} \cdot \log_d n\right)$
    - for large d, better than $\Theta\left(\frac{n}{B} \cdot \log_2 n\right)$ of standard MergeSort

d-way Mergesort Complexity In External Memory

- Further improvements
  - reduce number of rounds by starting immediately with runs of length M

- Suppose $M = 256$ and $d = 4$
  - previously, iterate with sorted runs of length
    - $1,\ 4,\ 16,\ 64,\ 256, 1024, \ldots$
  - Now, first sort subarrays of size 256 by bringing them into main memory
    - cost is equal to 1 round of merging
  - Now, iterate with sorted runs of length
    - $256, 1024, \ldots$
    - saves 3 iterations

d-Way merge in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53

Internal memory $M = 8$

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

39	5	28	22	10	33	29	37	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15

Internal (M = 8):

39	5	28	22	10	33	29	37

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

39	5	28	22	10	33	29	37	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15

Internal (M = 8):

5	10	22	28	29	33	37	39

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15

sorted run

Internal (M = 8):

5	10	22	28	29	33	37	39

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15

sorted run

Internal (M = 8):

8	30	54	40	31	52	21	45

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15

sorted run

Internal (M = 8):

8	21	30	31	40	45	52	54

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15

sorted run   sorted run

Internal (M = 8):

8	21	30	31	40	45	52	54

1. Create $\frac{n}{M}$ sorted runs of length M
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory: Initialization

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run sorted run sorted run sorted run

Internal (M = 8):

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run | sorted run | sorted run | sorted run

Internal (M = 8):

S_1		S_2		S_3		S	
5	10	8	21	11	12		

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run | sorted run | sorted run | sorted run

Internal (M = 8):

S_1		S_2		S_3		S	
	10	8	21	11	12	5	

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run   sorted run   sorted run   sorted run

Internal (M = 8):

S_1		S_2		S_3		S	
	10		**21**	**11**	**12**	**5**	**8**

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run   sorted run   sorted run   sorted run

5	8																														

Internal (M = 8):

S_1		S_2		S_3		S	
	10		**21**	**11**	**12**		

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run | sorted run | sorted run | sorted run

5	8																														

Internal (M = 8):

S_1		S_2		S_3		S	
			21	11	12	10	

($d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run   sorted run   sorted run   sorted run

5	8																														

Internal (M = 8):

S_1		S_2		S_3		S	
22	28		21	11	12	10	

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run | sorted run | sorted run | sorted run

5	8																														

Internal (M = 8):

S_1		S_2		S_3		S	
22	28		21		12	10	11

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run ⟷ sorted run ⟷ sorted run ⟷ sorted run

5	8	10	11																												

Internal (M = 8):

S_1		S_2		S_3		S	
22	28		21		12		

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

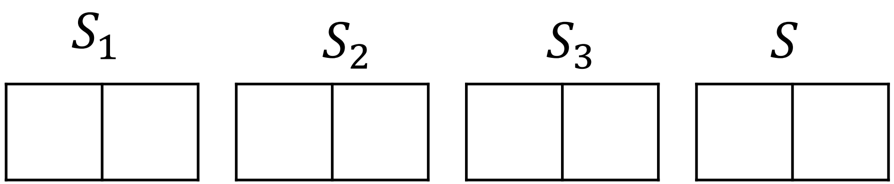
- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run sorted run sorted run sorted run

5	8	10	11	12	13	21	22	28	29	30	31	33	35	36	37	39	40	42	45	49	52	53	54								

sorted run



$(d = 3$, priority queue not shown$)$

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*

d-Way Mergesort in External Memory

- External $(B = 2)$

5	10	22	28	29	33	37	39	8	21	30	31	40	45	52	54	11	12	13	35	36	42	49	53	3	4	9	14	15	27	32	44

sorted run　sorted run　sorted run　sorted run

5	8	10	11	12	13	21	22	28	29	30	31	33	35	36	37	39	40	42	45	49	52	53	54	3	4	9	14	15	27	32	44

sorted run　sorted run

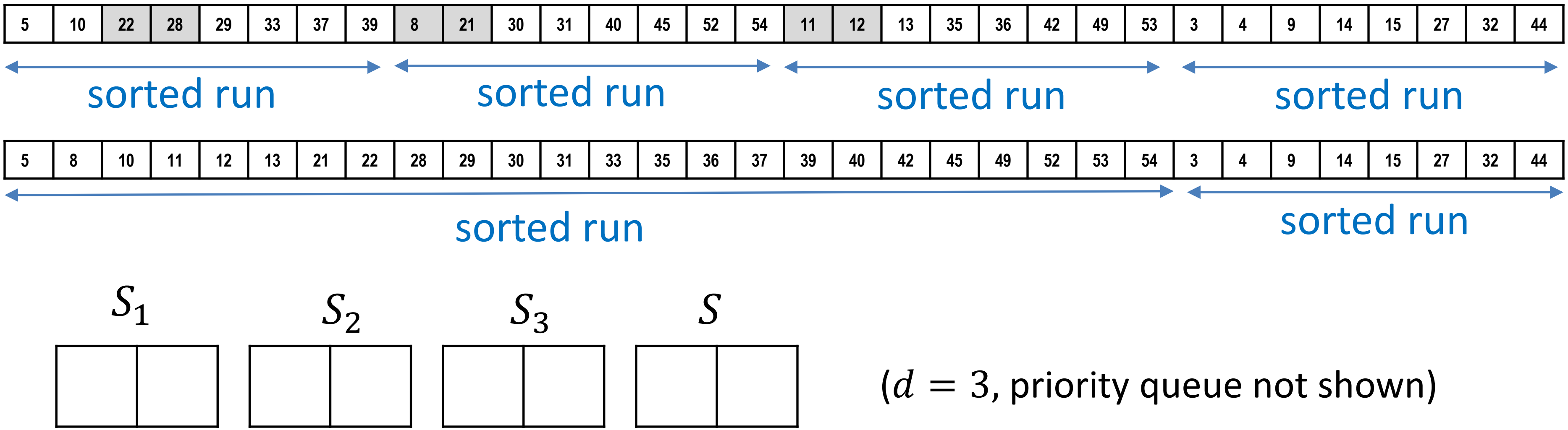
S_1　S_2　S_3　S

$(d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*
3. Keep merging the next runs to complete one round. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - after one round of merging, number of sorted runs reduced by a factor of d

d-Way Mergesort in External Memory

- External $(B = 2)$



($d = 3$, priority queue not shown)

1. Create $\frac{n}{M}$ sorted runs of length M. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - bring consecutive chunks of size M into internal memory
   - sort each chunk with an efficient sorting algorithm
2. Merge first $d \approx \frac{M}{B} - 1$ sorted runs using *d-way-Merge*
3. Keep merging the next runs to complete one round. Takes is $\Theta\left(\frac{n}{B}\right)$ block transfers
   - after one round of merging, number of sorted runs reduced by a factor of d
4. Keep doing rounds until we get just one sorted run

d-Way Mergesort in External Memory: Running time

- How many rounds?
  - $\frac{n}{M}$ runs after initialization
  - each round decreases the number of sorted runs by a factor of d
  - $\frac{n}{M}/d$ runs after one round
  - $\frac{n}{M}/d^k$ runs after k rounds
  - stop when $\frac{\frac{n}{M}}{d^k} = 1 \Longrightarrow k = \log_d \frac{n}{M}$
  - $\log_d \frac{n}{M}$ rounds of merging
- Each round takes $\Theta\left(\frac{n}{B}\right)$ block transfers

since $d \approx \frac{M}{B} - 1$

- Total number of bock transfers is proportional to $\frac{n}{B} \cdot \log_d \frac{n}{M} \in O\left(\frac{n}{B} \cdot \log_{M/B} \frac{n}{M}\right)$
- One can prove lower bound in external memory model for comparison sorting

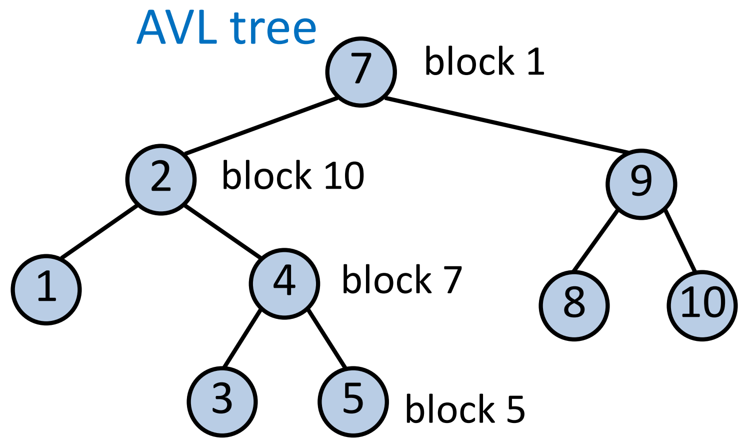
$$\Omega\left(\frac{n}{B} \cdot \log_{M/B} \frac{n}{M}\right)$$

- Thus d-way mergesort is optimal (up to constant factors)

Outline

- External Memory
  - Motivation
  - Stream Based Algorithms
  - External sorting
  - External Dictionaries
    - 2-4 Trees
    - (*a*, *b*)-Trees
    - B-Trees

Dictionaries in External Memory: Motivation

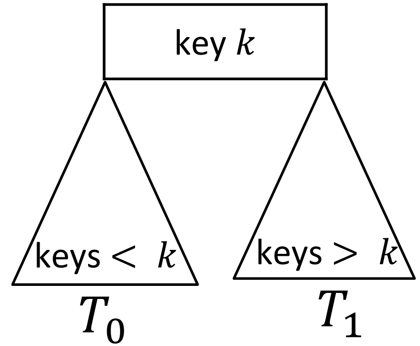


- AVL tree based dictionary implementations have poor *memory locality*
  - tree nodes are in non-contiguous memory locations
  - for any tree path, each node is usually in a different block
- In an AVL tree $\Theta(\log n)$ blocks are loaded in the worst case
- Idea: define multi-way tree
  - one node stores many KVPs
  - for multi-way trees, $b - 1$ KVPs $\Leftrightarrow$ b subtrees
- For efficient insert/delete, we permit a varying number of KVPs in nodes
- This gives much smaller height than AVL-trees
  - smaller height implies fewer 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

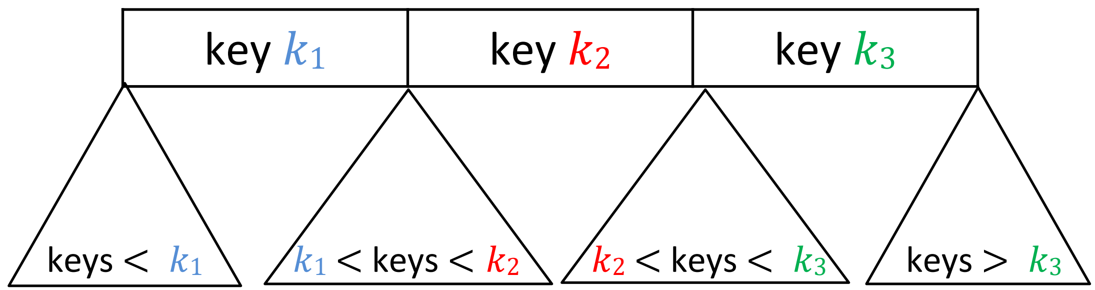
- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - (*a*, *b*)-Trees
    - B-Trees

2-4 Trees Motivation



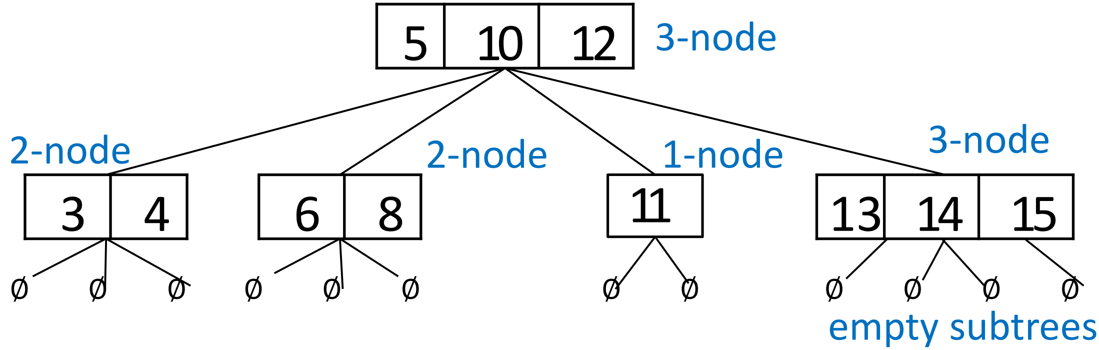
- Binary Search Tree supports efficient search with special key ordering

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

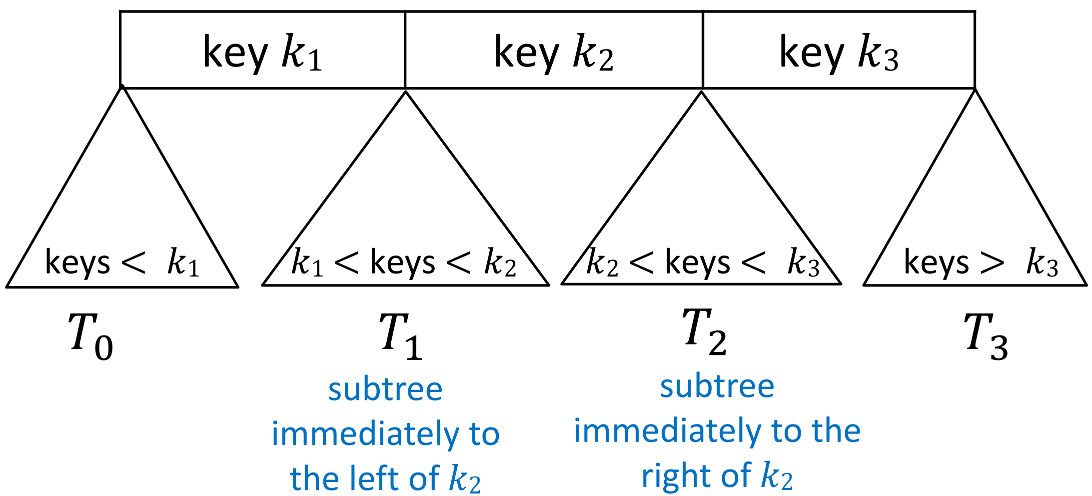


- Need additional properties to ensure tree is balanced and therefore *insert*, *delete* are efficient

2-4 Trees



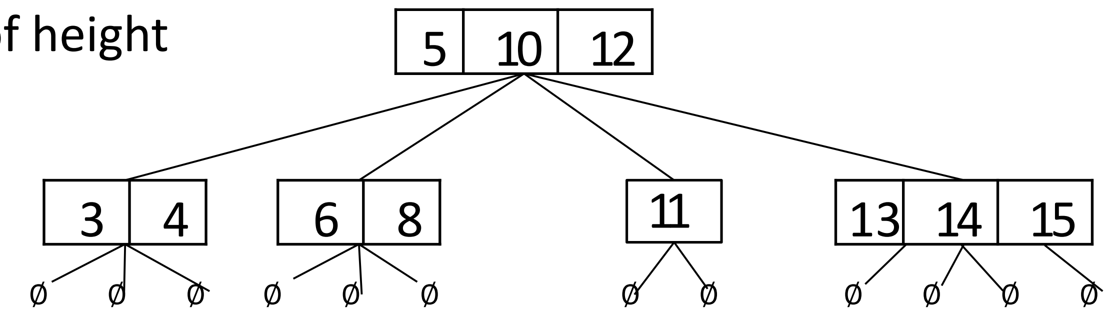
- **Structural properties**
  - Every node is either
    - 1-node: *one KVP* and *two subtrees* (possibly empty), or
    - 2-node: *two KVPs* and *three subtrees* (possibly empty), or
    - 3-node: *three KVPs* and *four subtrees* (possibly empty)
    - allowing 3 types of nodes simplifies insertion/deletion
  - 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
$< T_0, k_1, T_1, k_2, T_2, k_3, T_3, k_1 >$

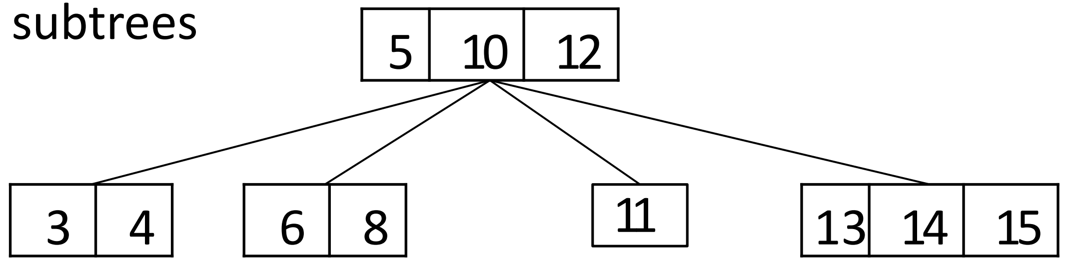
2-4 Tree Example

- Empty subtrees are not part of height computation



tree of height 1

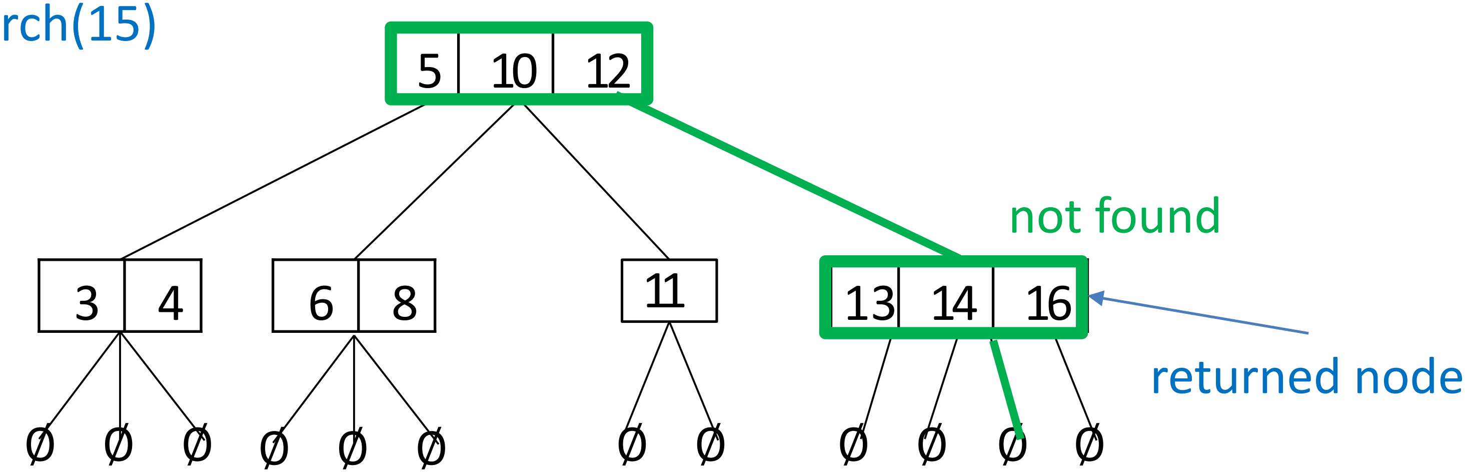
- Often do not even show empty subtrees



- 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$(k, v \leftarrow$root$, p \leftarrow$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 $< T_0, k_1, \ldots, k_d, T_d >$ be key-subtrees list at v

if $k \geq k_1$

$i \leftarrow$ maximal index such that $k_i \leq k$

if $k_i = k$

return "at ith key in v"

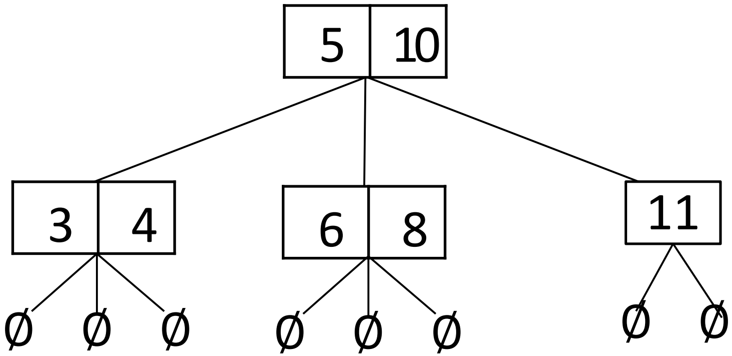
else *24Tree::search*(k, T_i, v)

else *24Tree::search*(k, T_0, v)

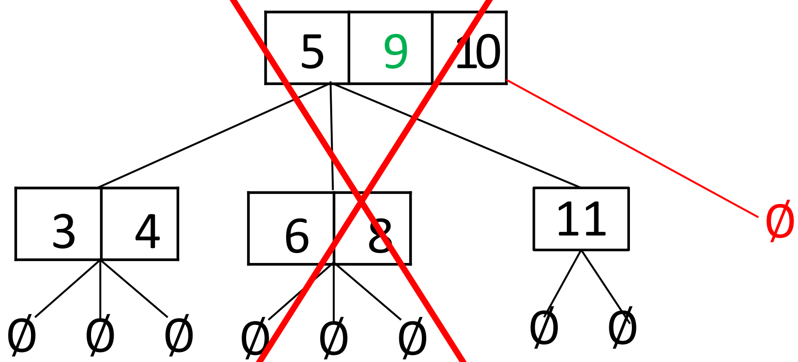
Example: 2-4 tree Insert

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

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



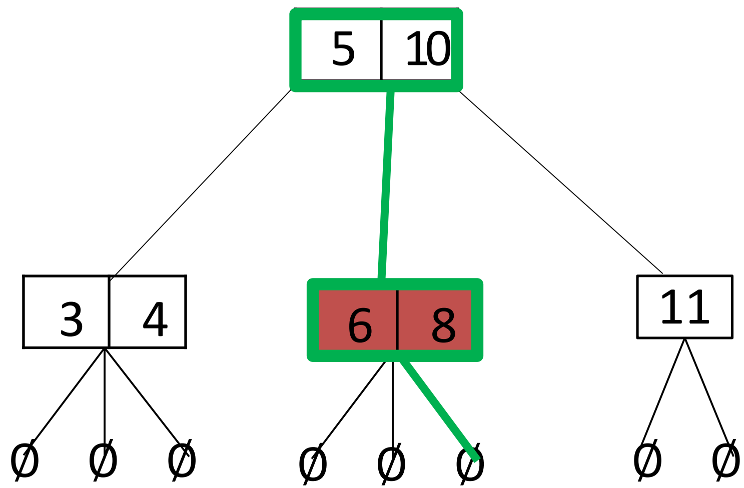
however, need 1 more subtree,
since node has 3 keys now!



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

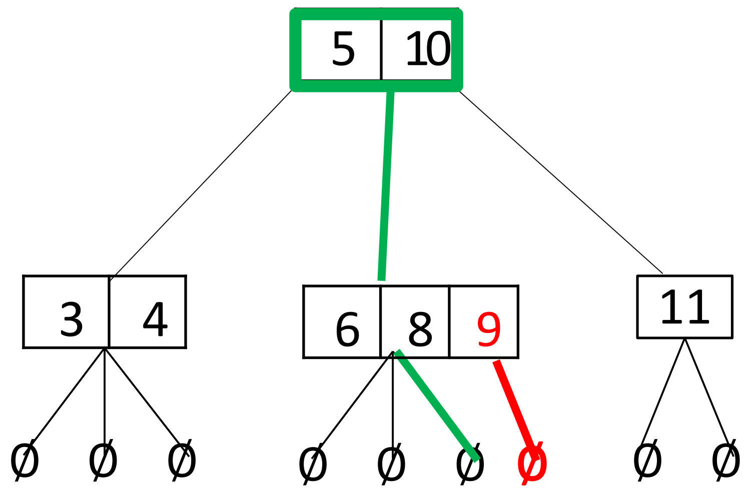
Example: 2-4 tree Insert

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



Example: 2-4 tree Insert

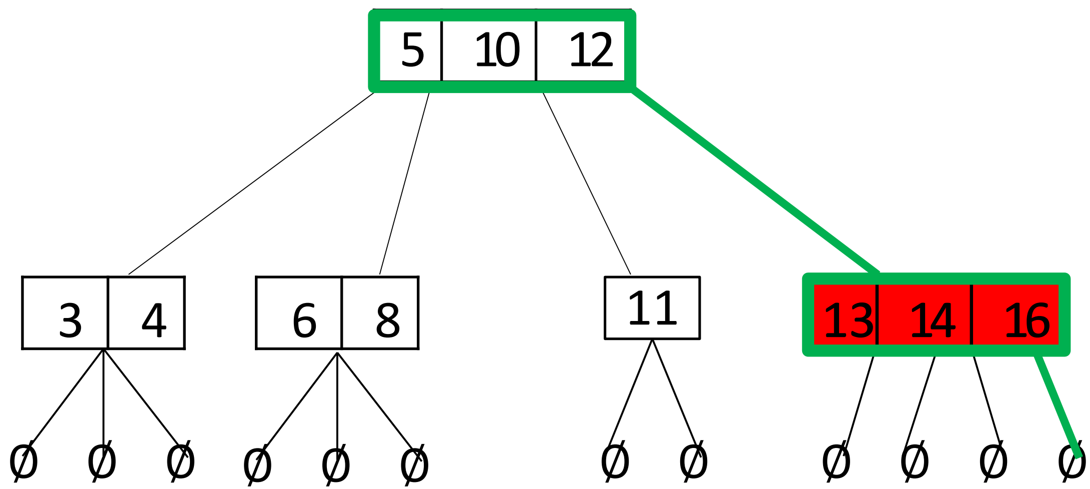
- **Example**: *24TreeInsert*(9)
  - first step: 24Tree::search(9)
  - second step: insert at the leaf node returned by search



- 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

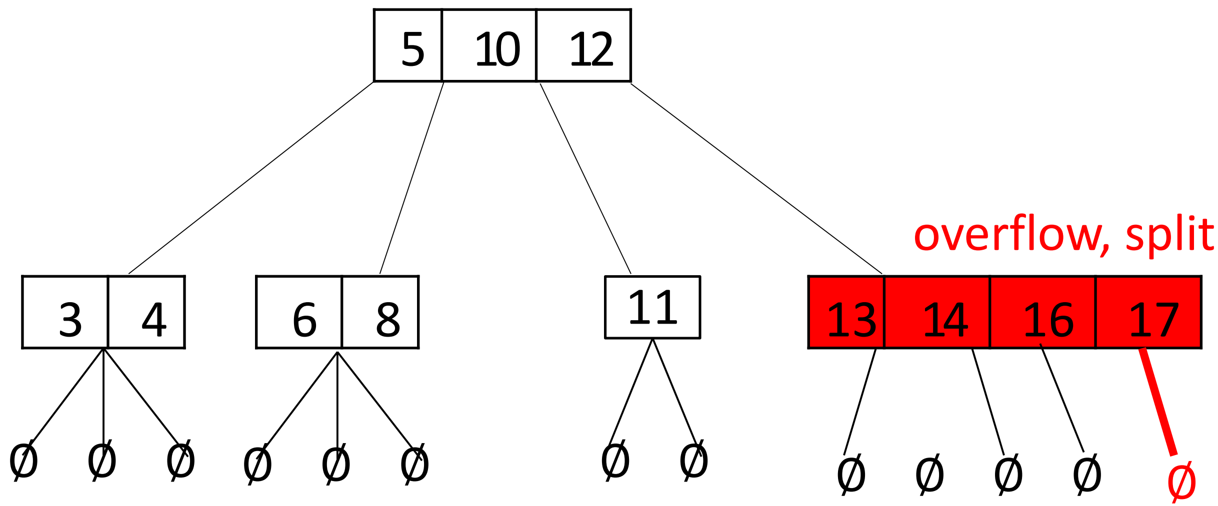
Example: 2-4 tree Insert

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



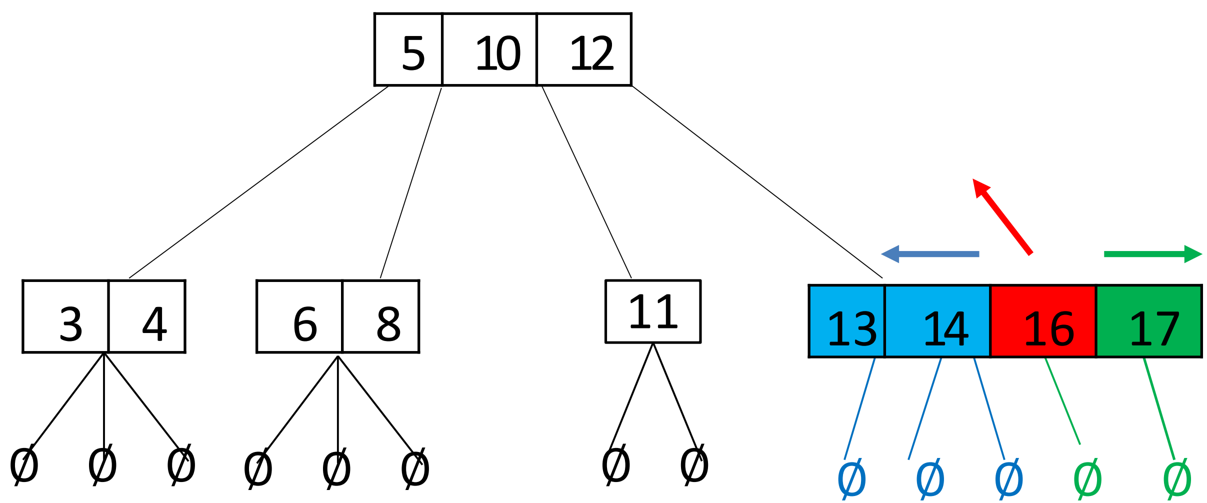
Example: 2-4 tree Insert

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



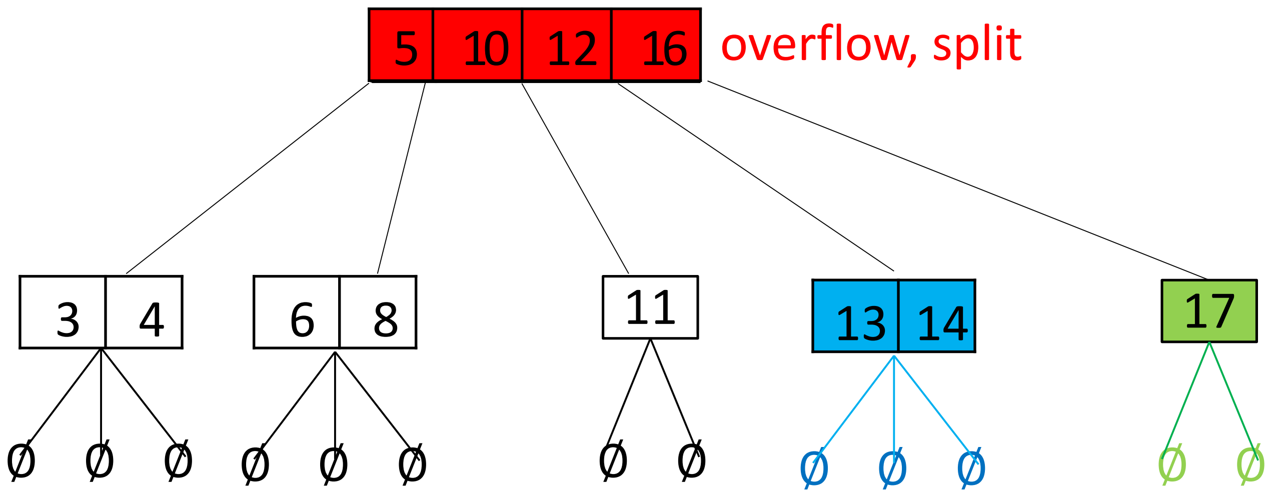
Example: 2-4 tree Insert

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



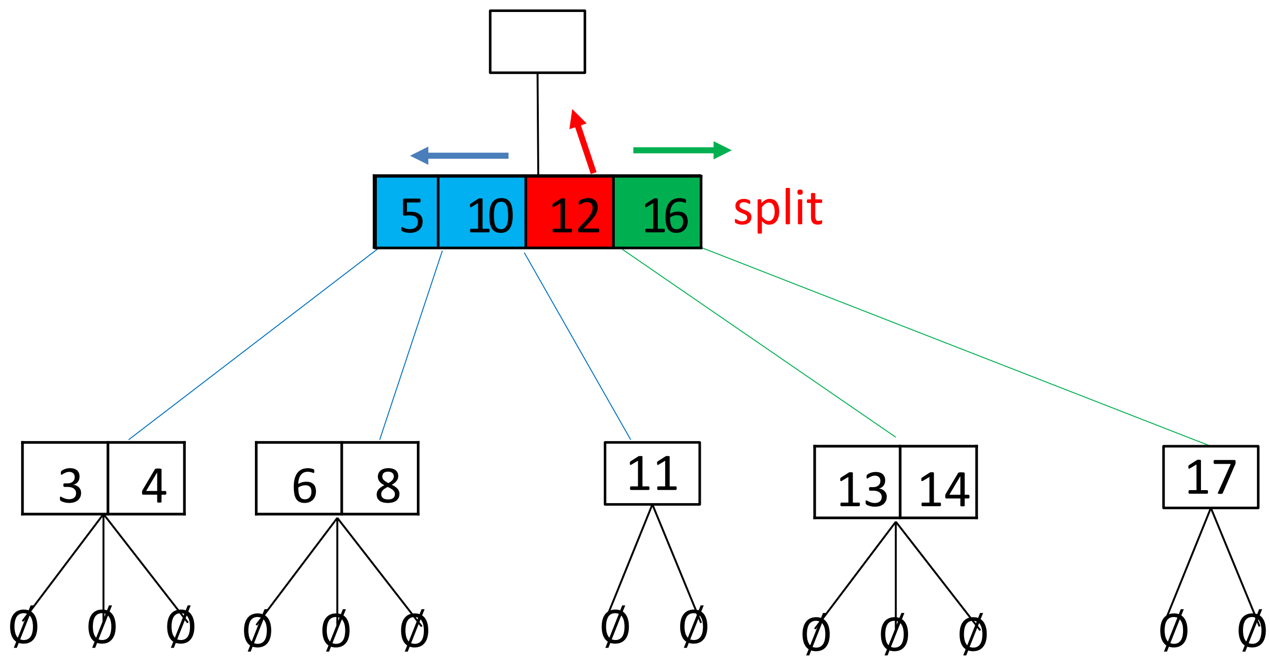
Example: 2-4 tree Insert

- **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



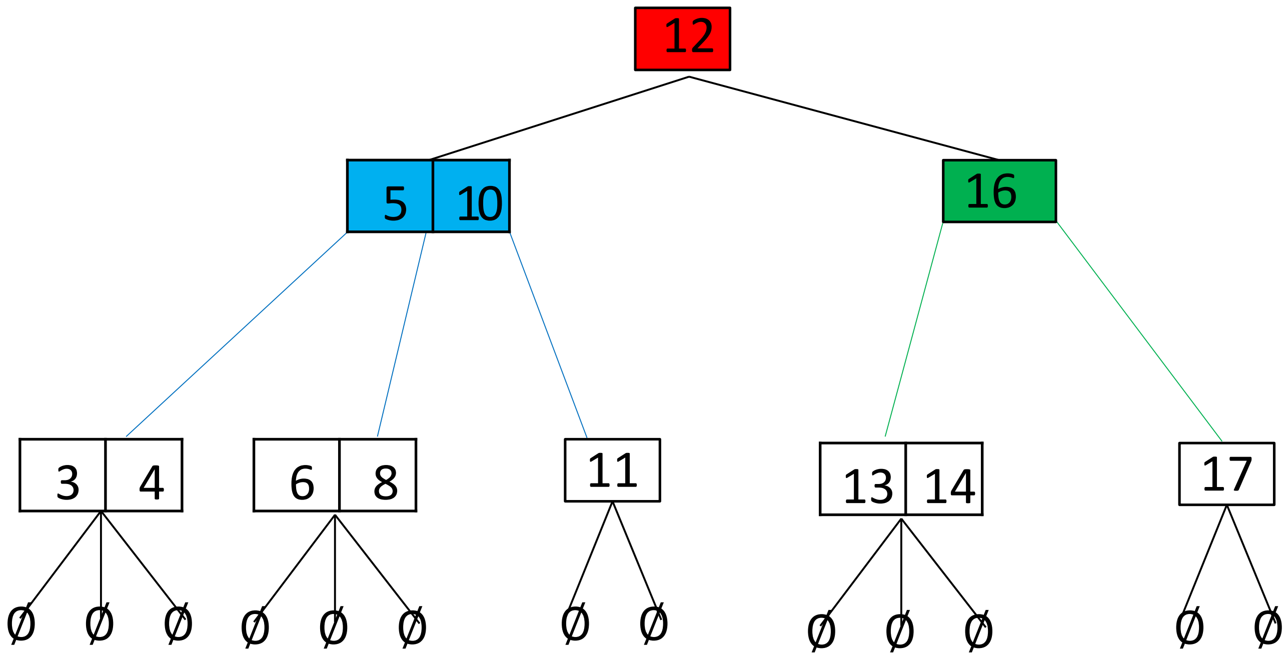
Example: 2-4 tree Insert

- **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

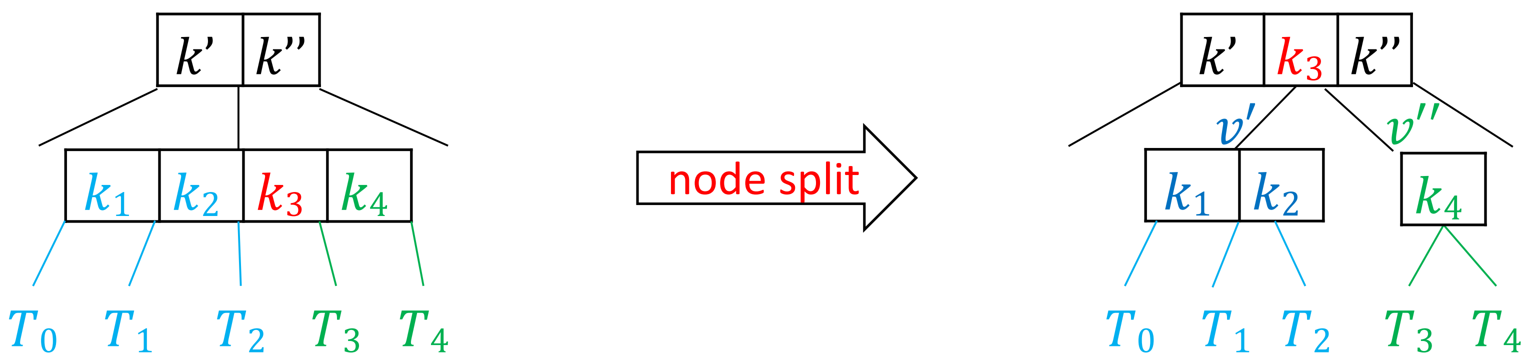
$p \leftarrow$ parent of v

$v' \leftarrow$ 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

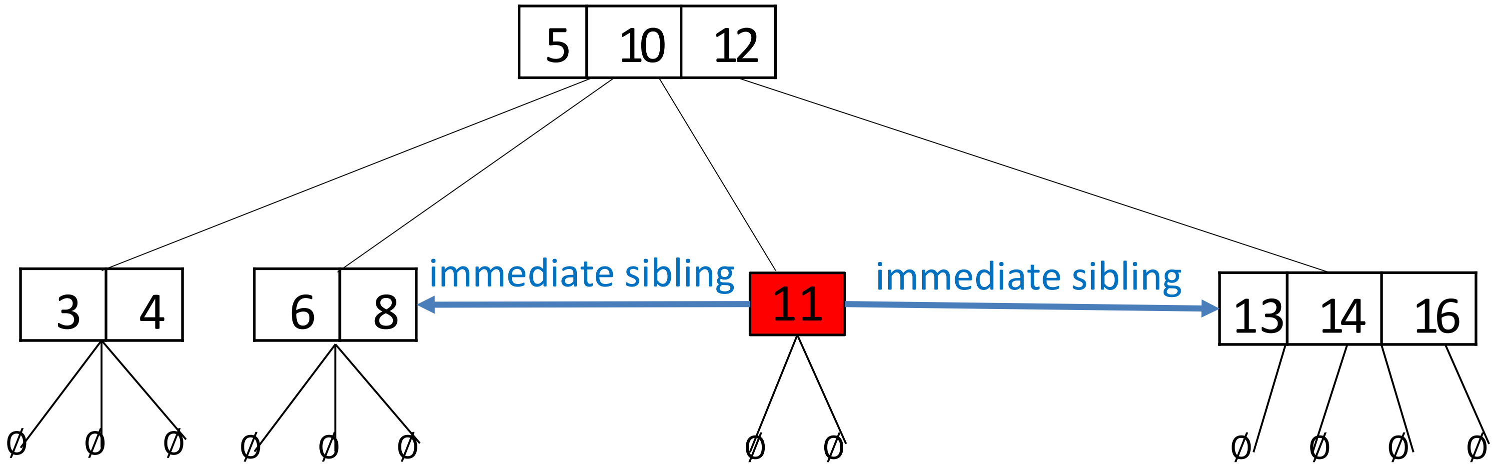
replace $< v >$ by $< v', k_3, v'' >$ in key-subtree-list of p

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

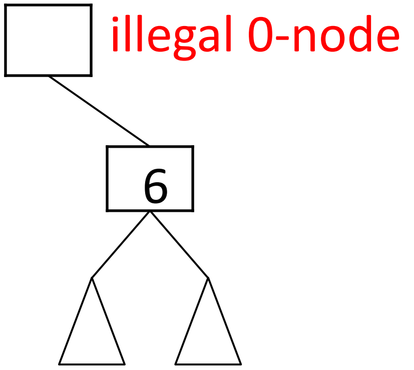


2-4 Tree: Immediate Sibling

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

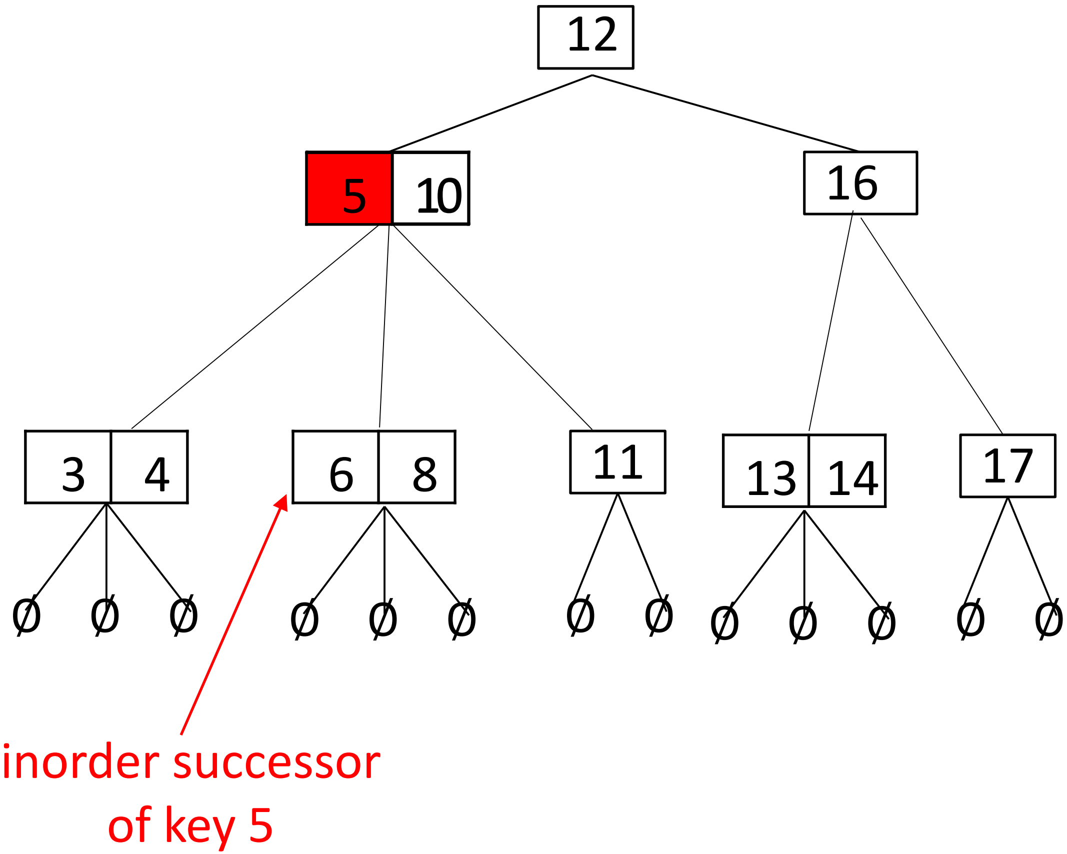


- Any node except the root must have an immediate sibling



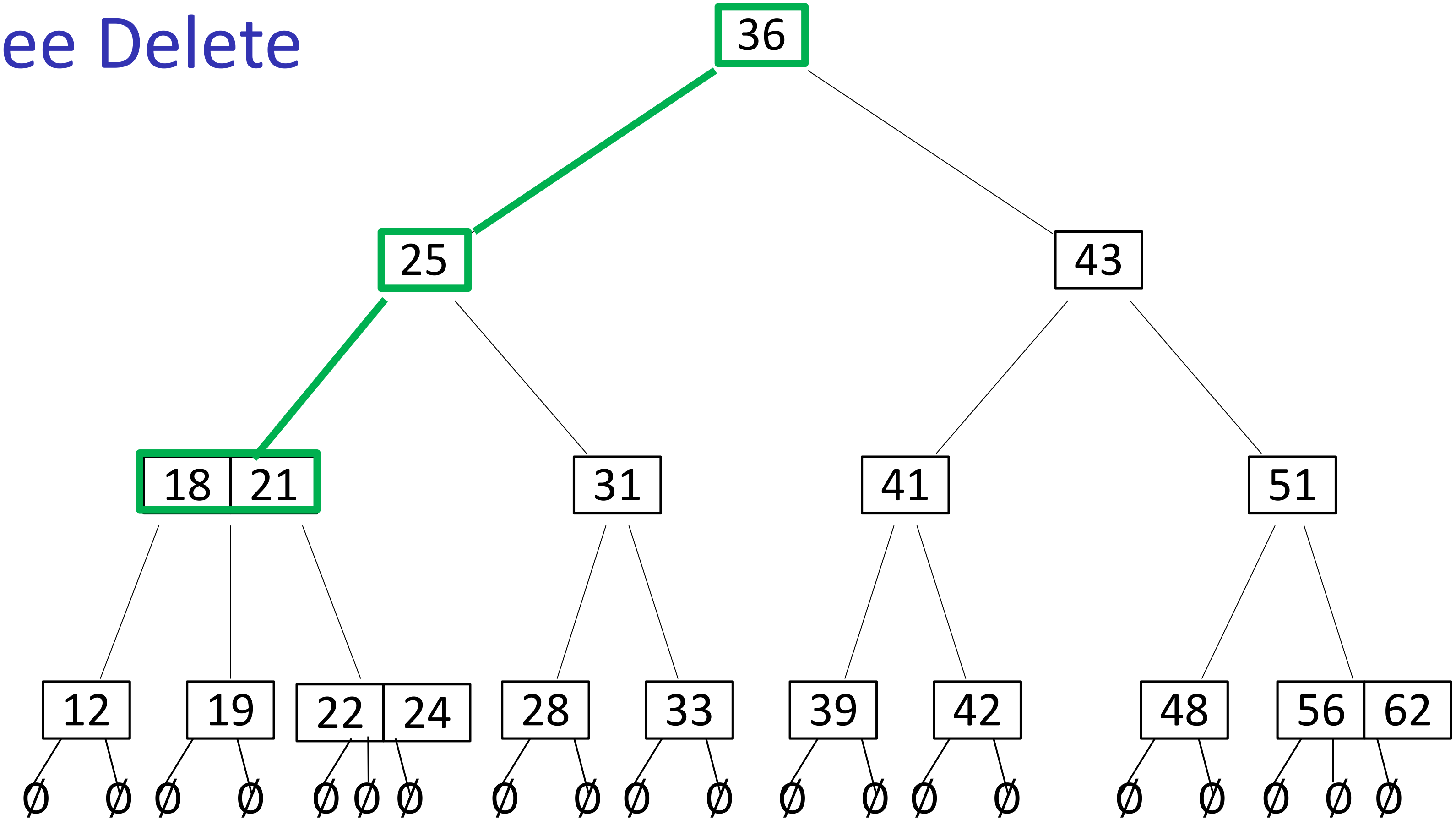
2-4 Tree: Inorder Successor

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



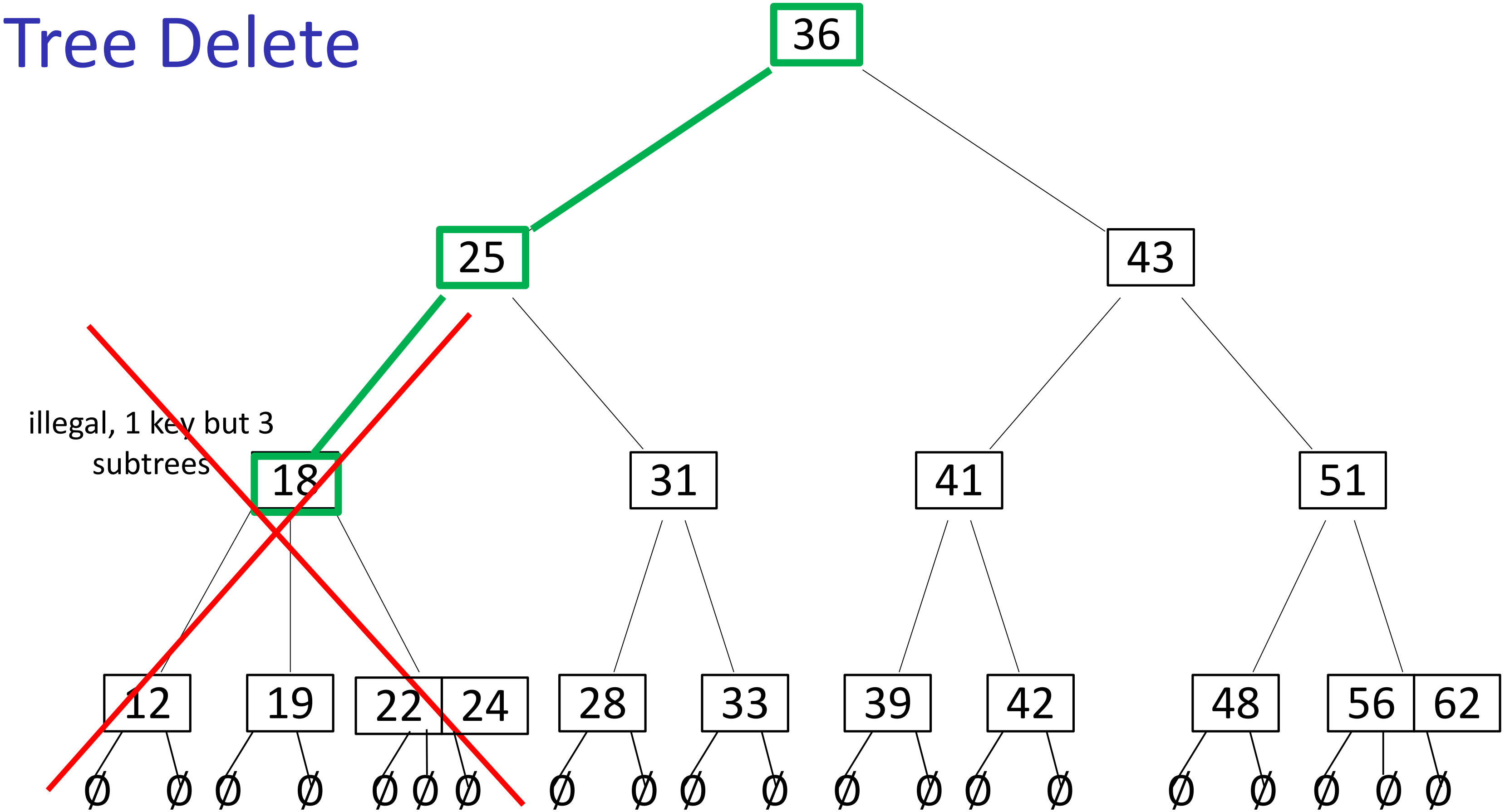
- Inorder successor is guaranteed to be at a leaf node
    - otherwise would have something smaller in the leftmost subtree

2-4 Tree Delete



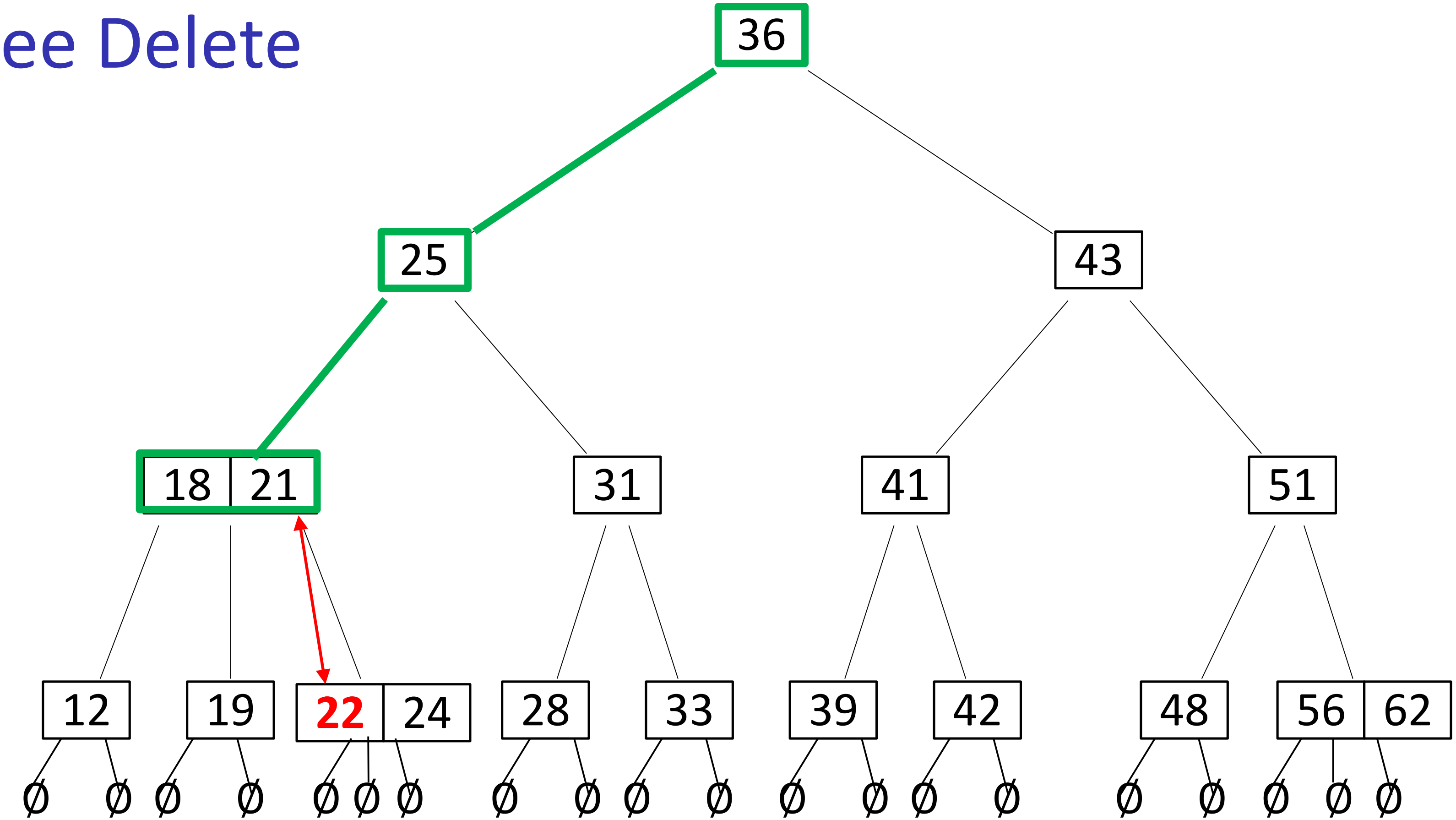
- Example: *delete*(21)
- Search for key to delete
  - if a node found has more than 1 key, it is tempting to delete it directly

2-4 Tree Delete



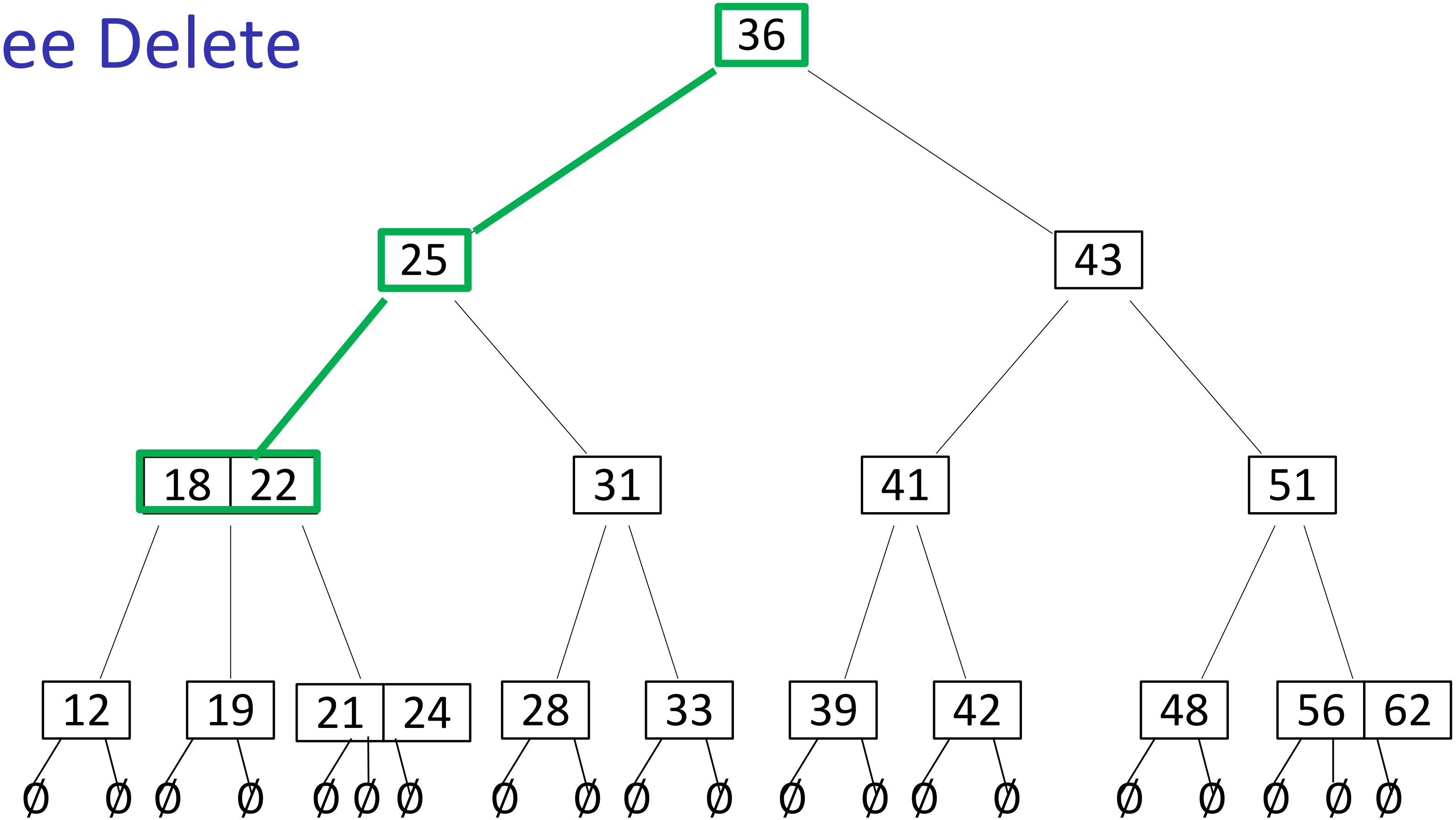
- Example: *delete*(21)
- Search for key to delete
  - if a node found has more than 1 key, it is tempting to delete it directly
  - however, can delete the key directly only if a node is a leaf
    - when we delete a key, we need to delete 1 subtree, easy only at a leaf

2-4 Tree Delete



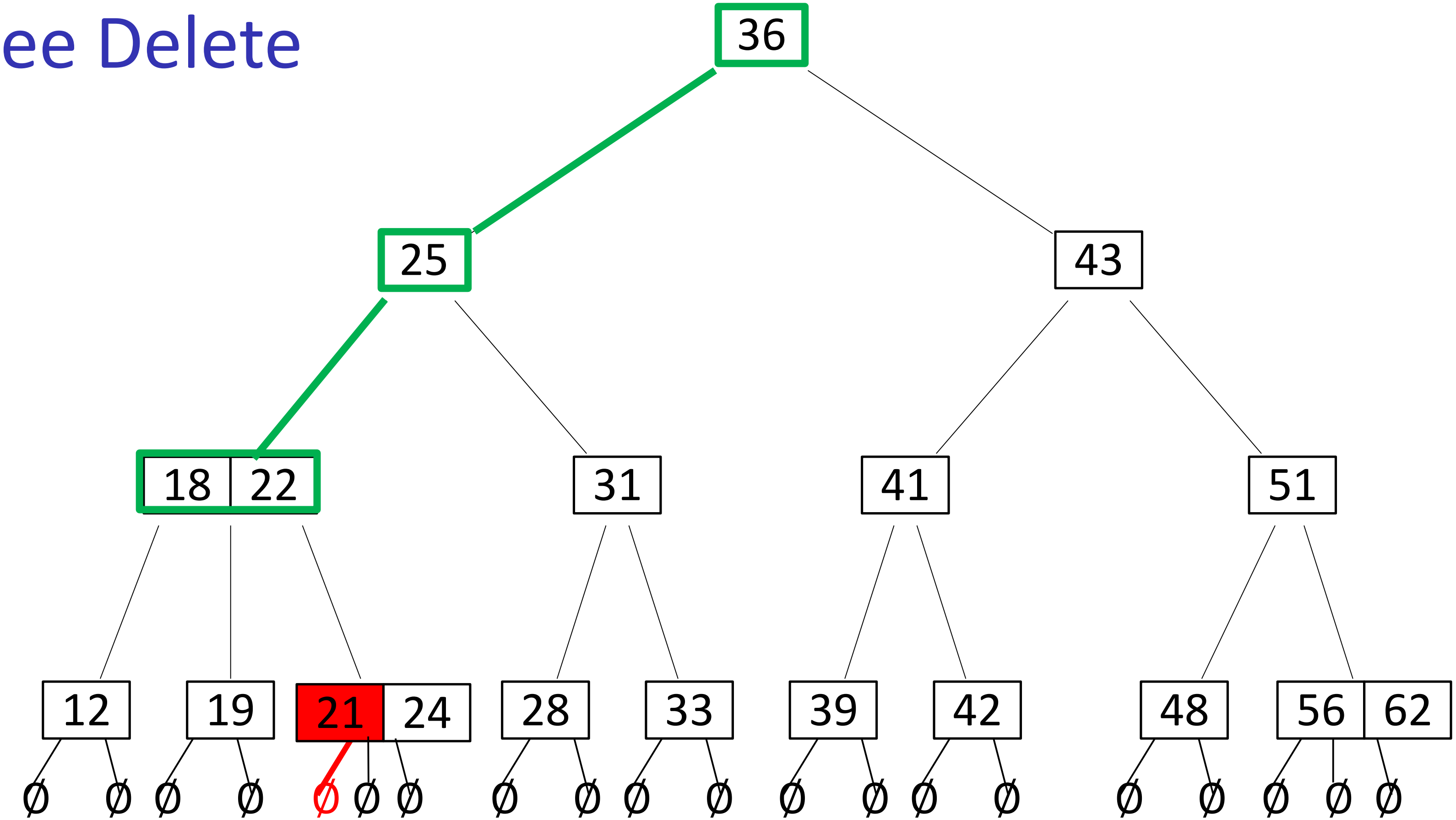
- Example: *delete*(21)
- Search for key to delete
  - can delete keys only from a leaf node, as need to delete a subtree as well
  - if the key is in a node which is not a leaf, replace key with its inorder successor

2-4 Tree Delete



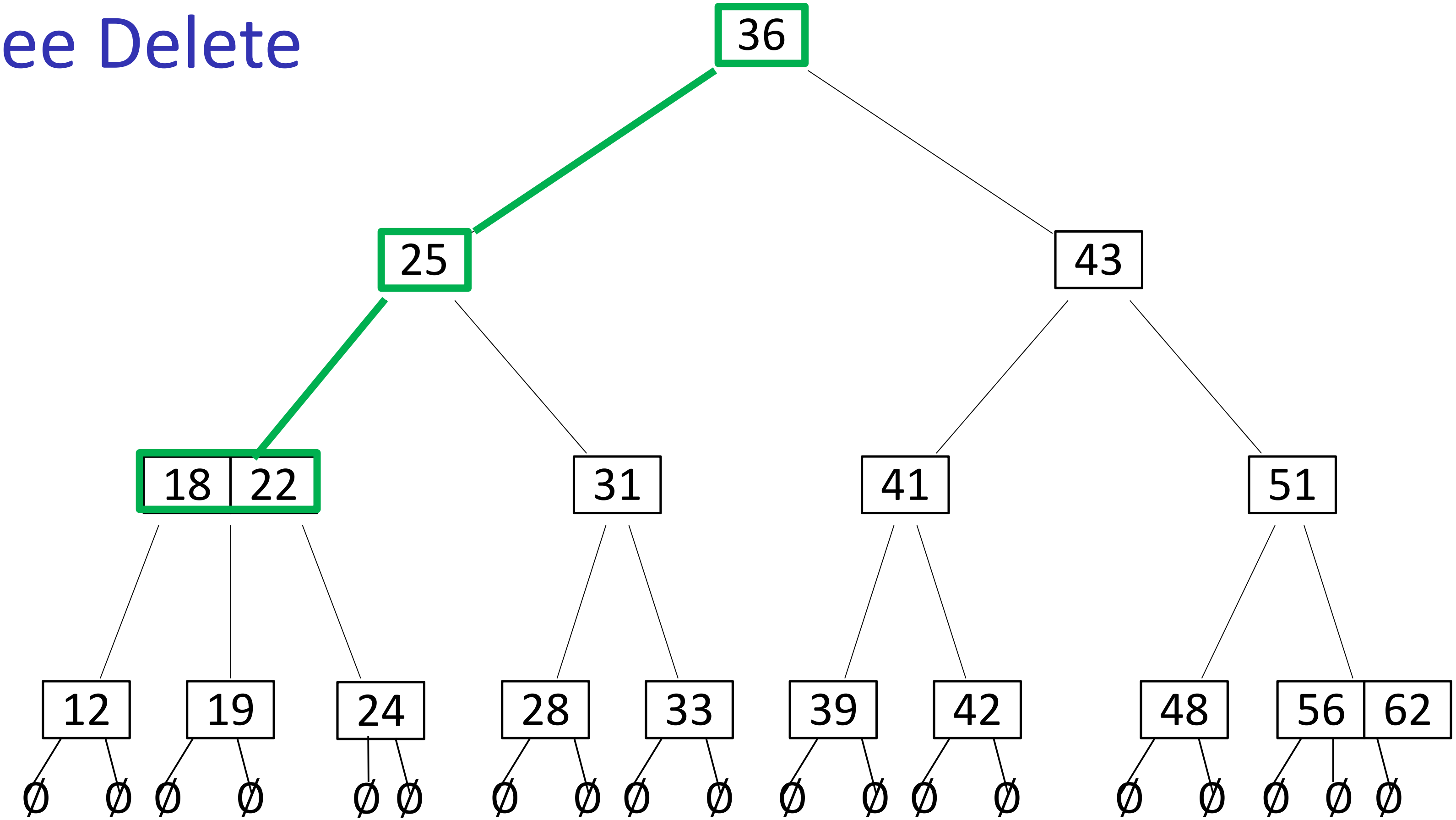
- Example: *delete*(21)
- Search for key to delete
  - can delete keys only from a leaf node, as need to delete a subtree as well
  - if the key is in a node which is not a leaf, replace key with its inorder successor

2-4 Tree Delete



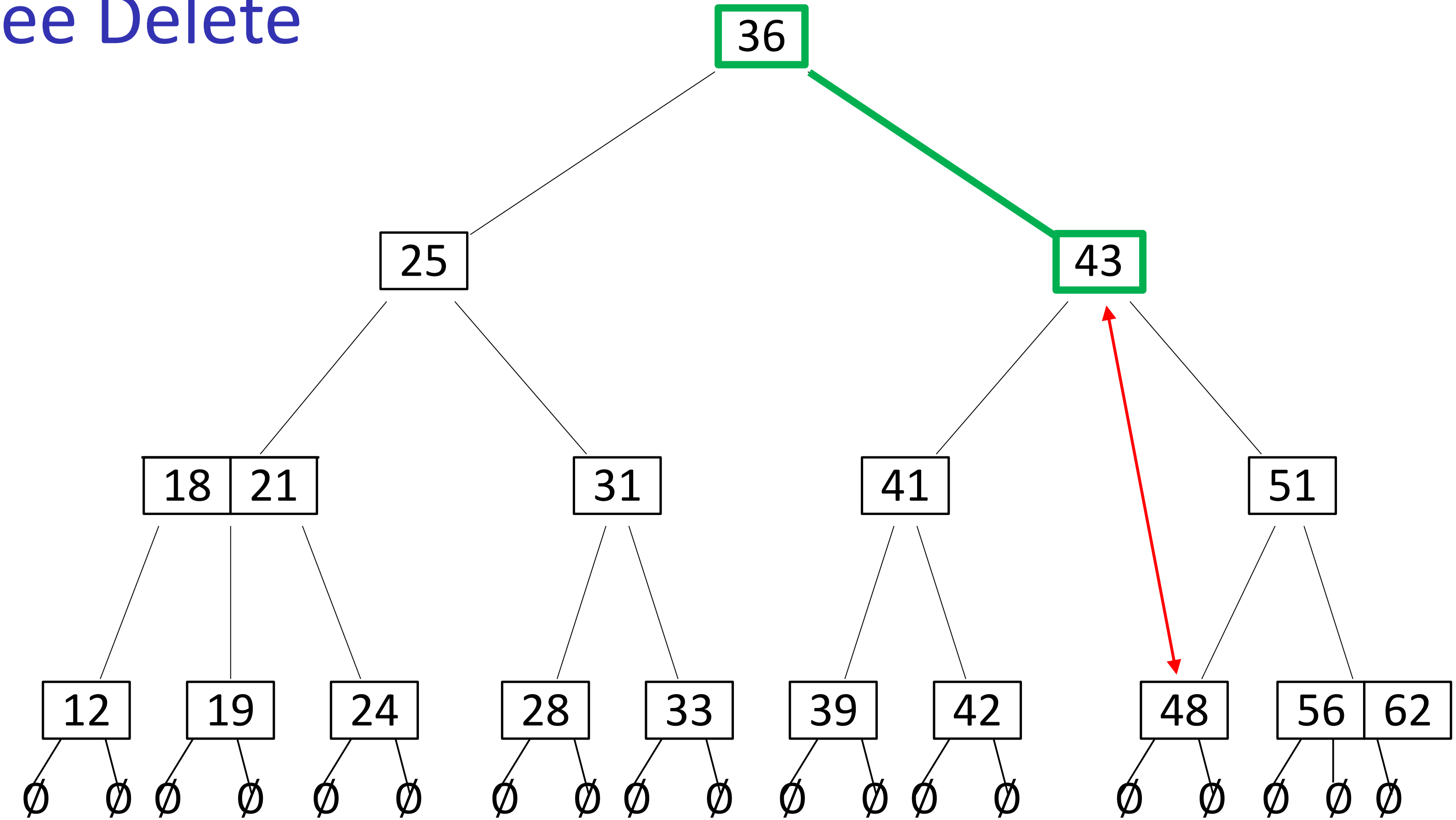
- Example: *delete*(21)
- Search for key to delete
  - can delete keys only from a leaf node, as need to delete a subtree as well
  - if the key is in a node which is not a leaf, replace key with its inorder successor
  - delete key 21 and an empty subtree

2-4 Tree Delete



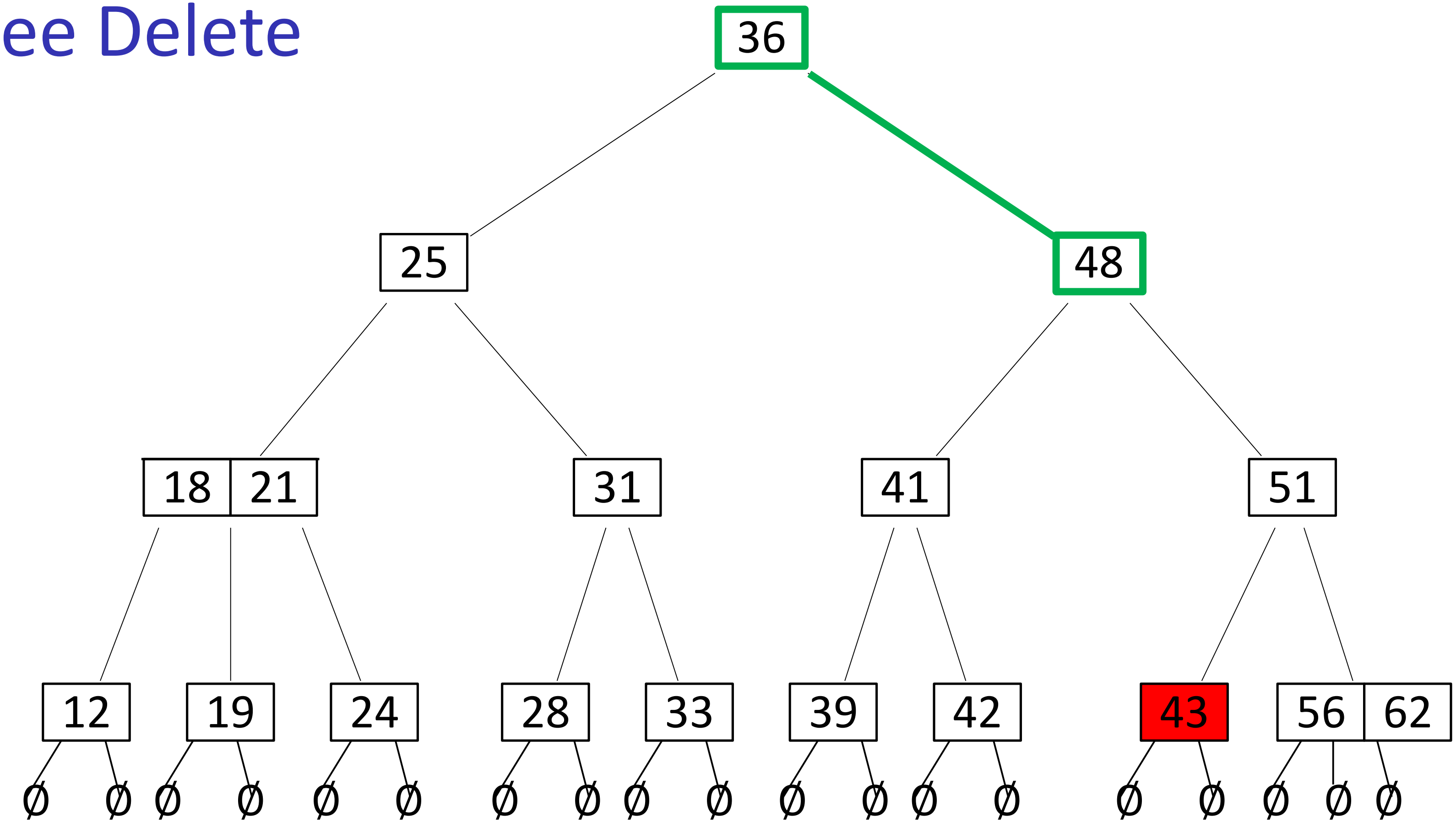
- Example: *delete*(21)
- Search for key to delete
    - can delete keys only from a leaf node, as need to delete a subtree as well
    - if the key is in a node which is not a leaf, replace key with its inorder successor
    - delete key 21 and an empty subtree
    - order property is preserved and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

2-4 Tree Delete



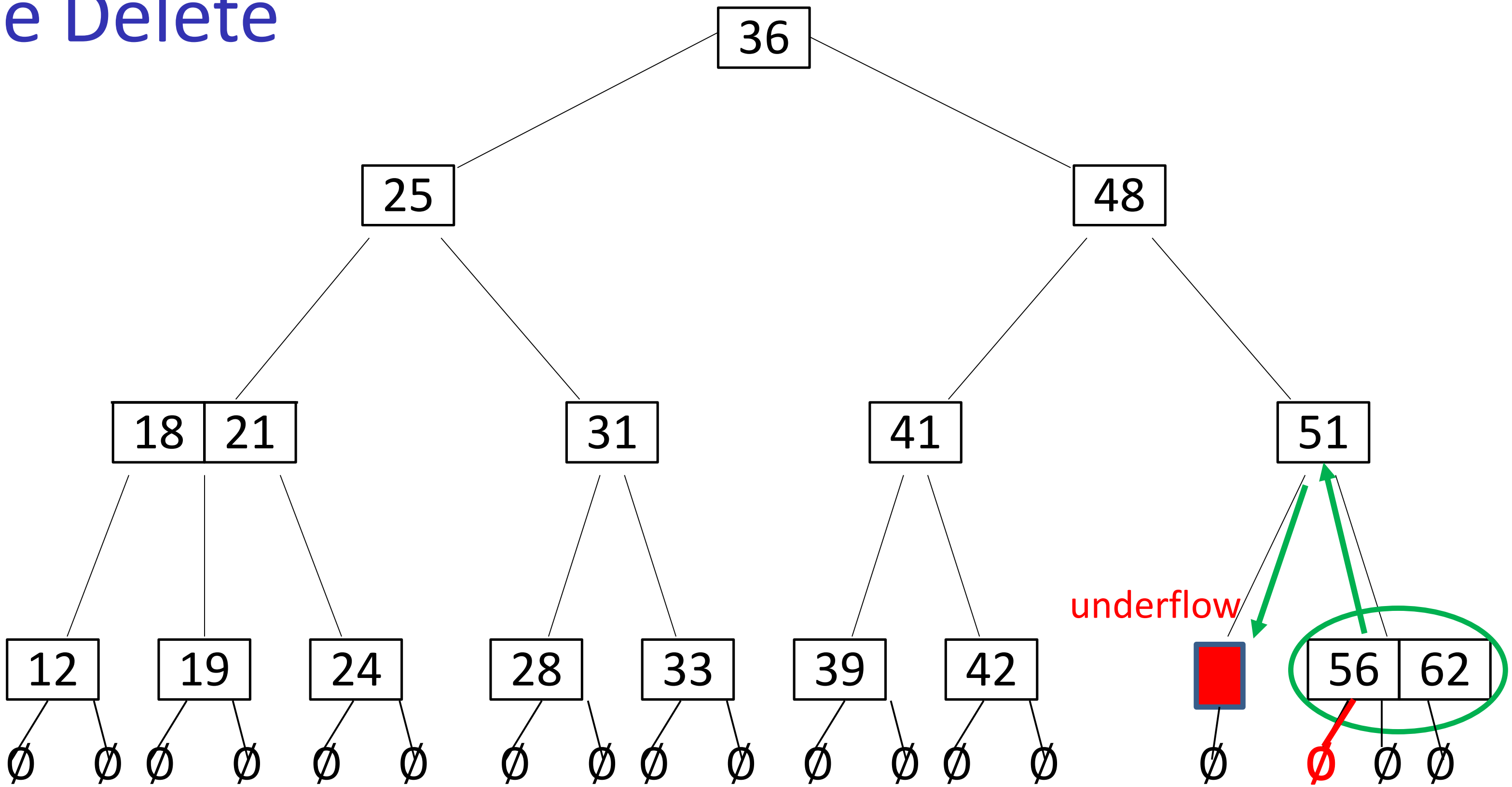
- Example: *delete*(43)
- Search for key to delete
  - can delete keys only from a leaf node
  - replace key with in-order successor

2-4 Tree Delete



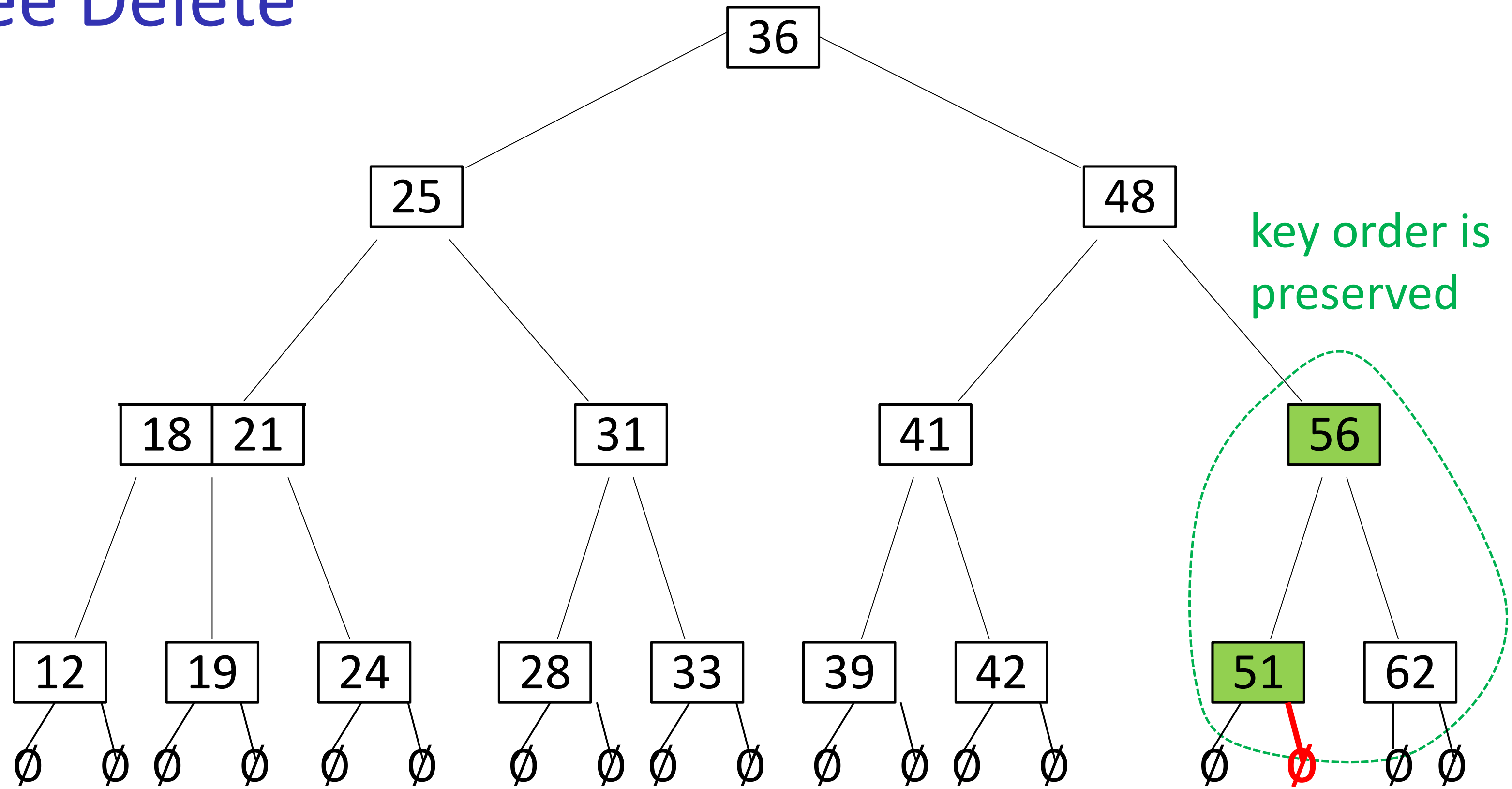
- Example: *delete*(43)
- Search for key to delete
  - can delete keys only from a leaf node
  - replace key with in-order successor
  - delete key 43 and a subtree

2-4 Tree Delete



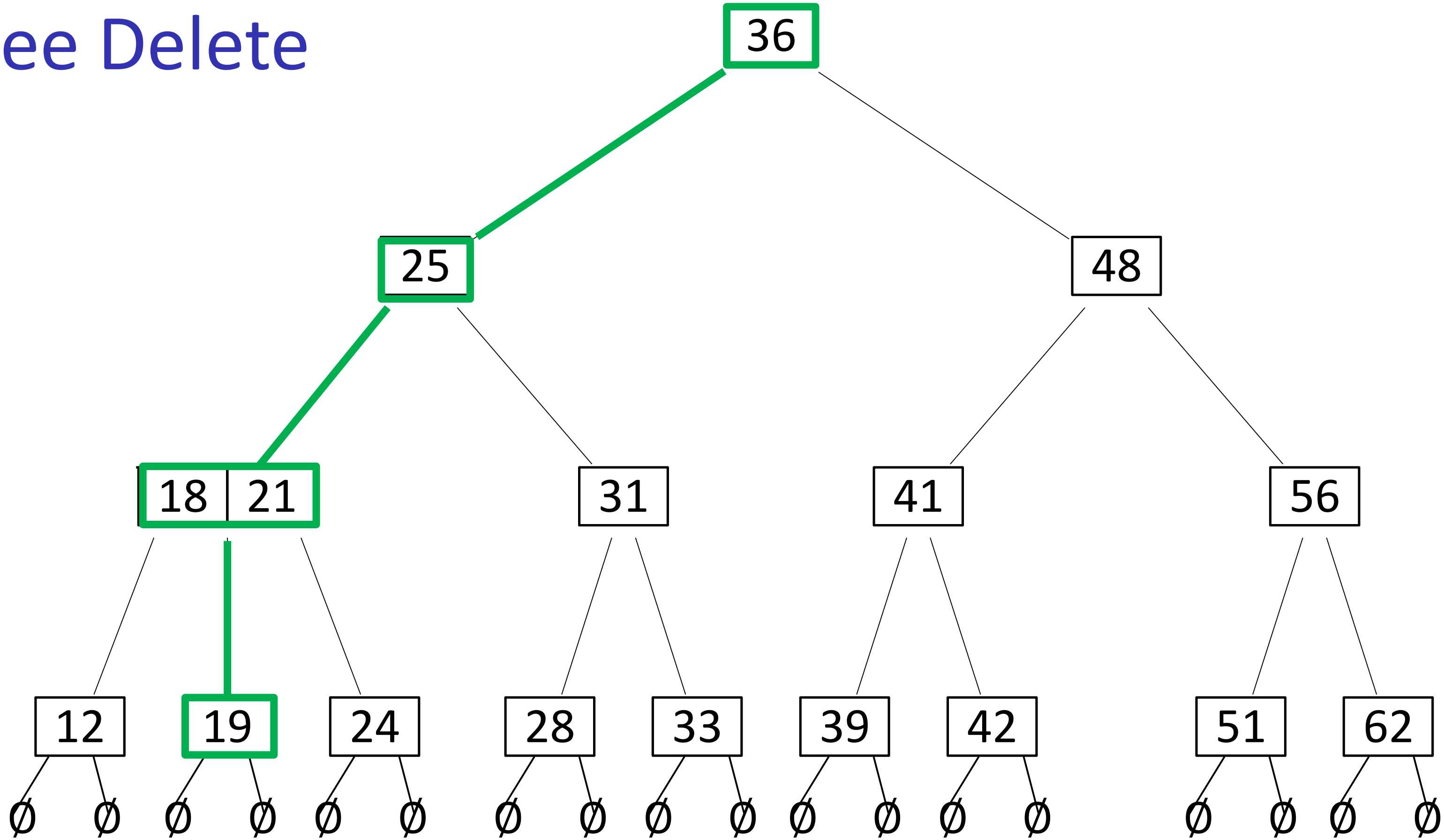
- Example: *delete*(43)
  - *rich* immediate sibling, transfer key from sibling, with help from the parent
    - sibling is *rich* if it is a 2-node or 3-node
    - adjacent subtree from sibling is also transferred

2-4 Tree Delete



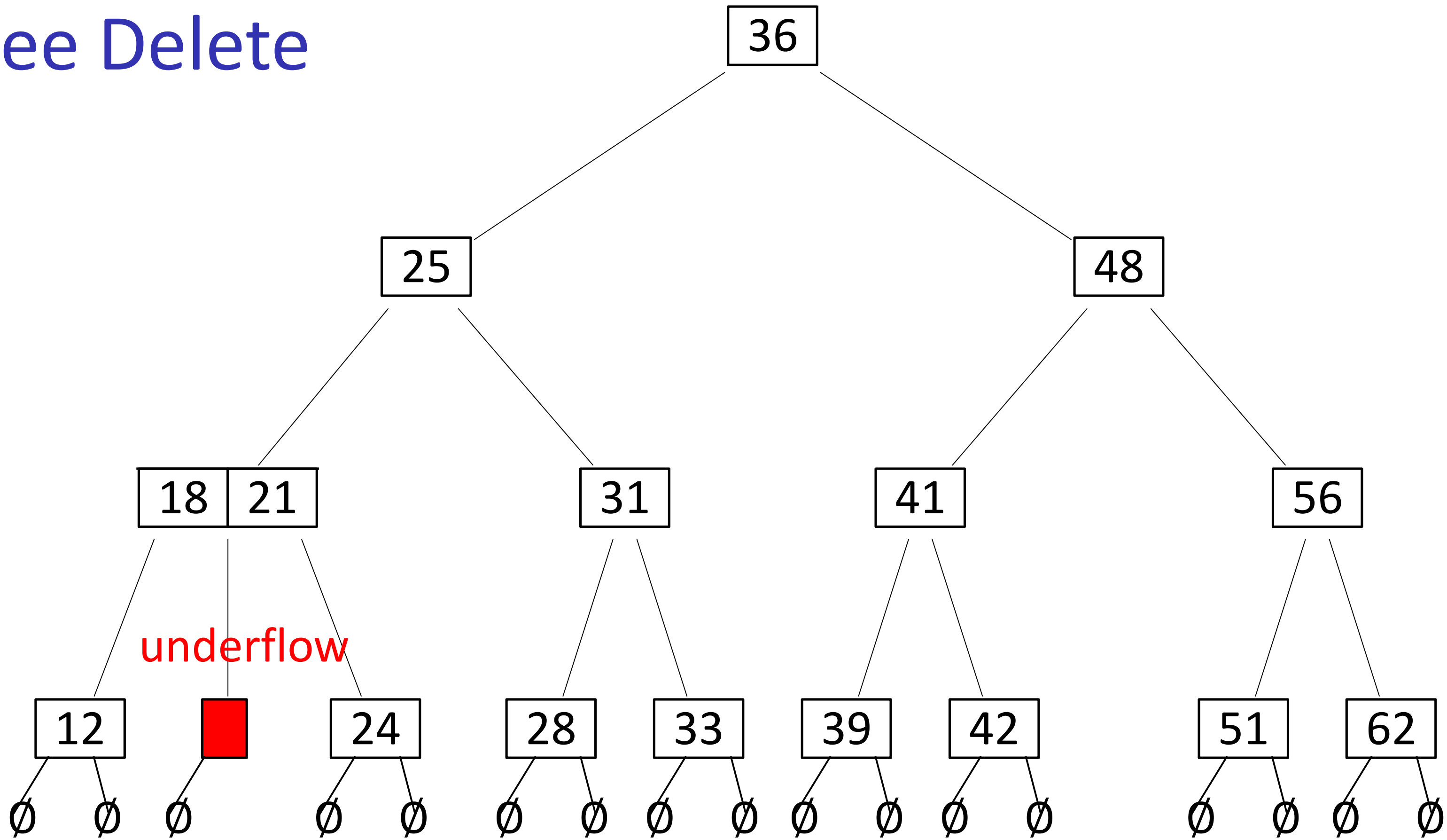
- Example: *delete*(43)
  - *rich* immediate sibling, transfer key from sibling, with help from the parent
    - sibling is *rich* if it is a 2-node or 3-node
    - adjacent subtree from sibling is also transferred
    - order property is preserved

2-4 Tree Delete



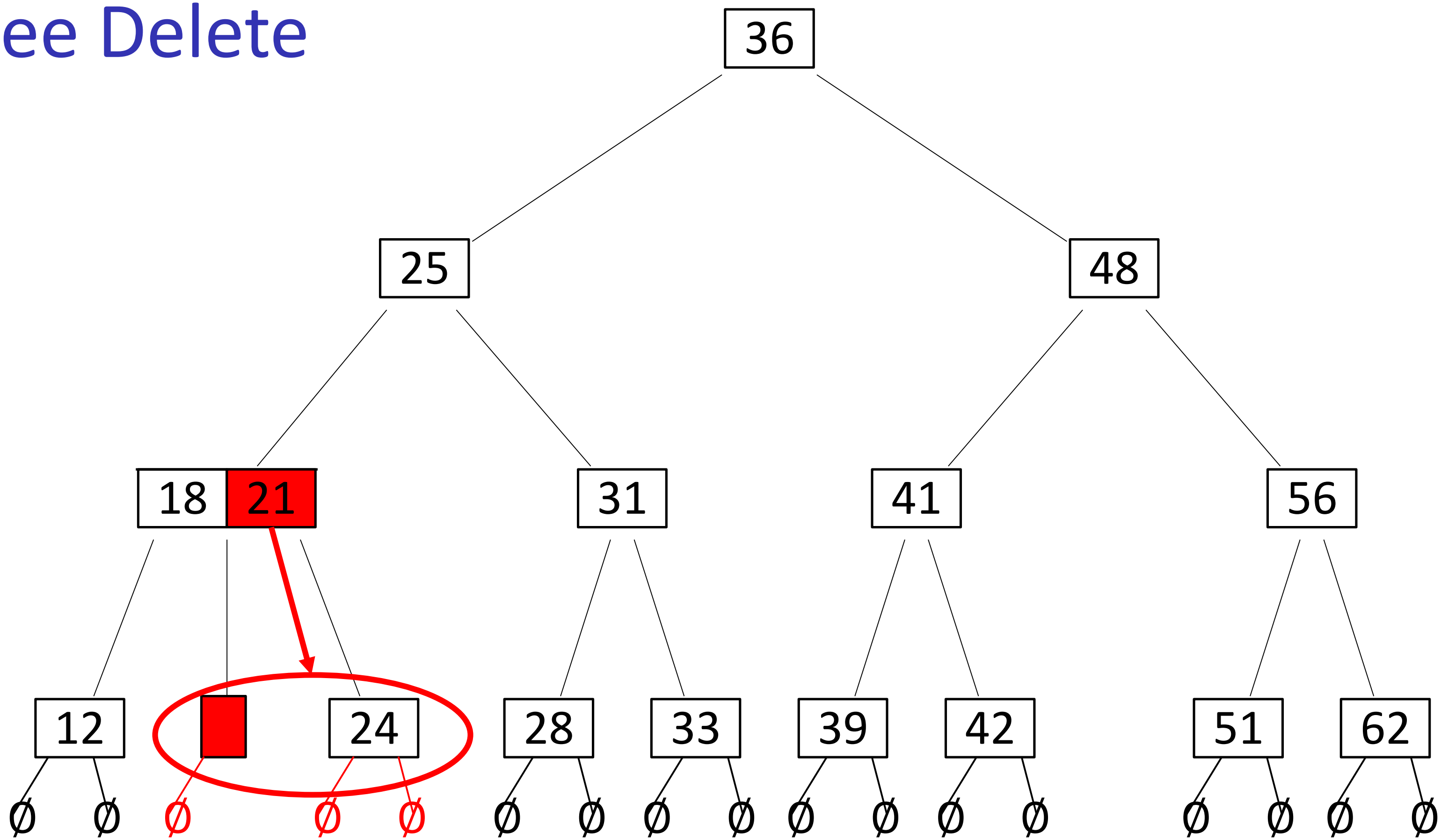
- Example: *delete*(19)
  - first search(19)

2-4 Tree Delete



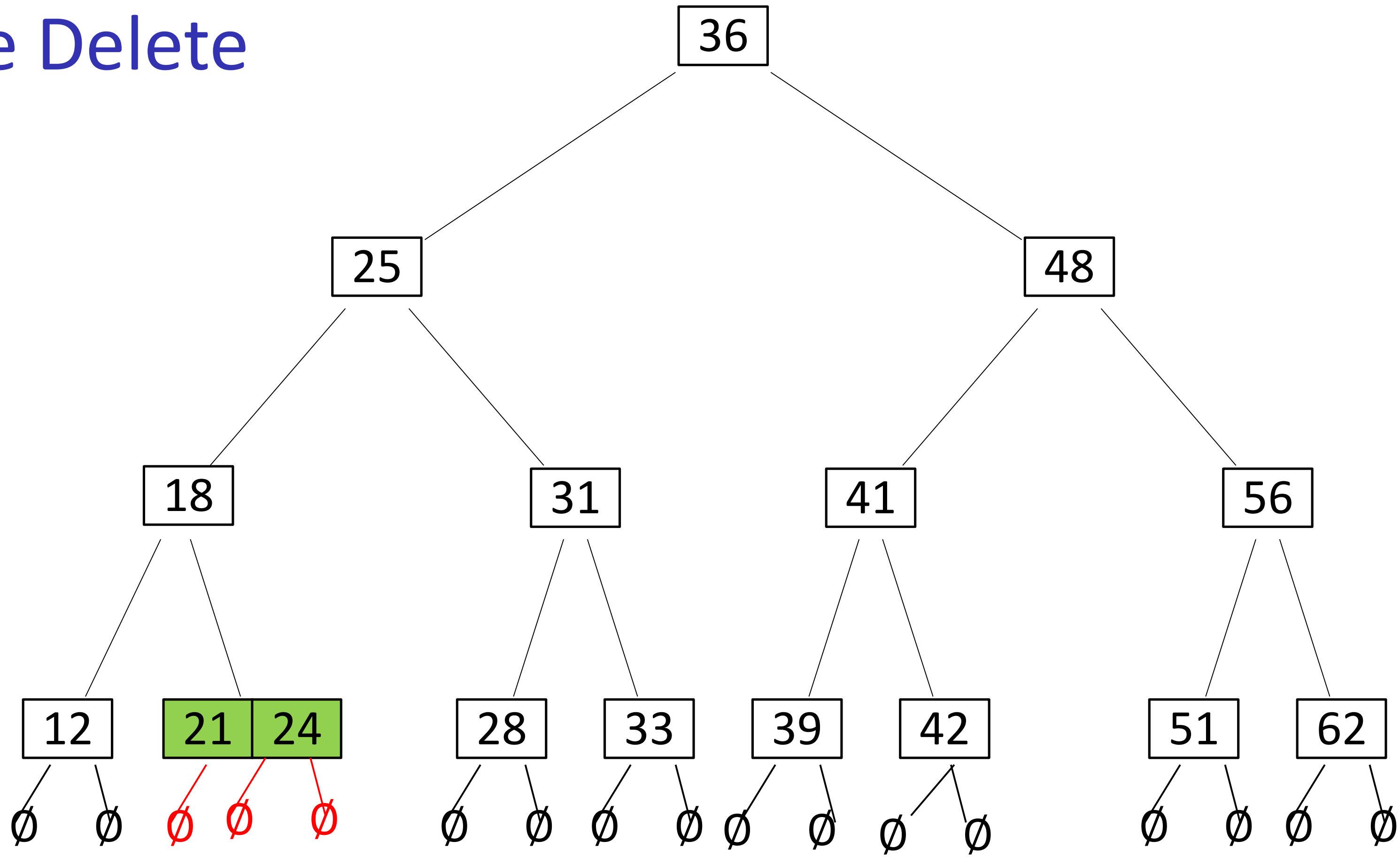
- Example: *delete*(19)
  - first search(19)
  - then delete key 19 (and an empty subtree) from the node
  - immediate siblings exist, but not rich, cannot transfer

2-4 Tree Delete



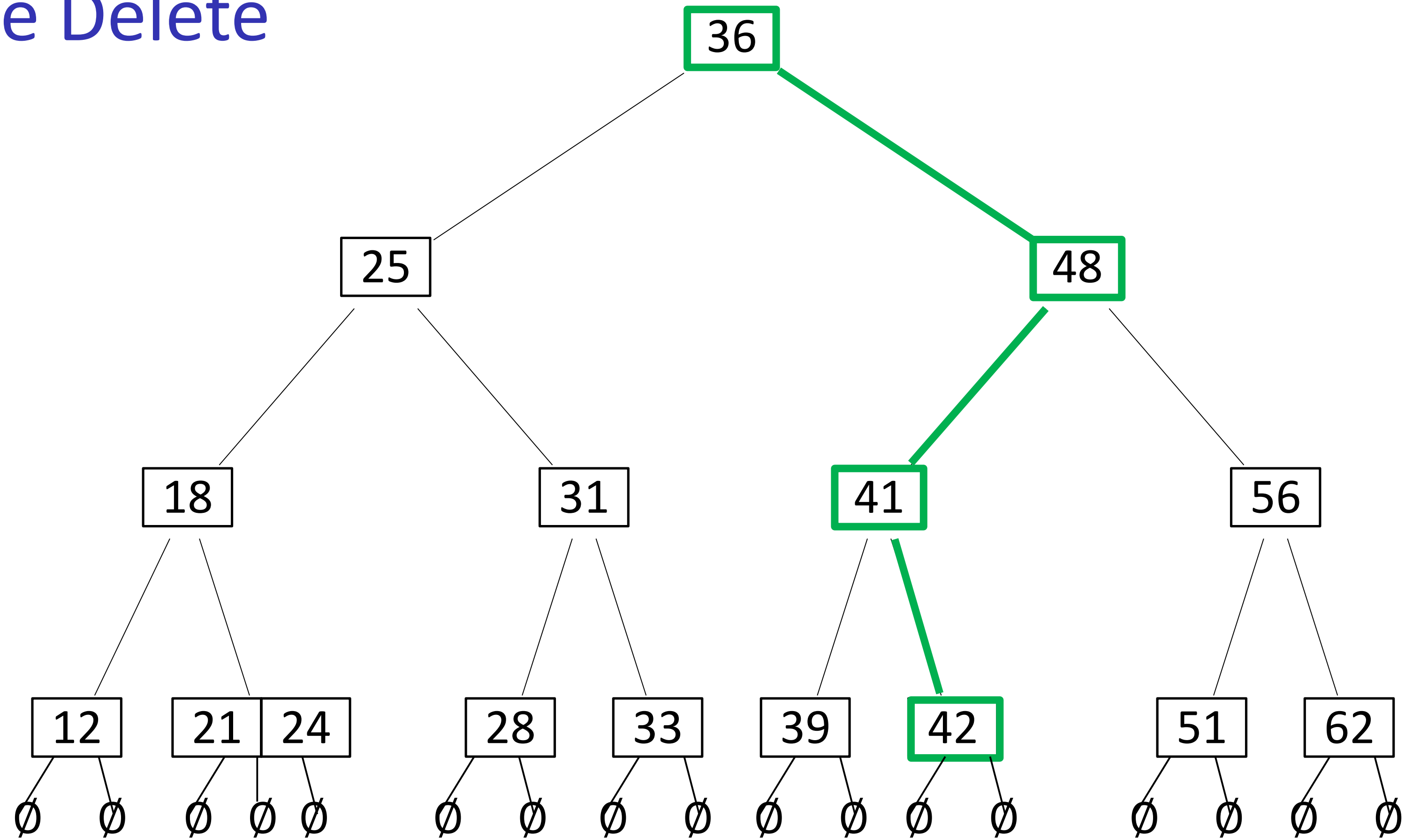
- Example: *delete*(19)
  - immediate siblings exist, but not rich, cannot transfer
    - merge with right immediate sibling with help from parent

2-4 Tree Delete



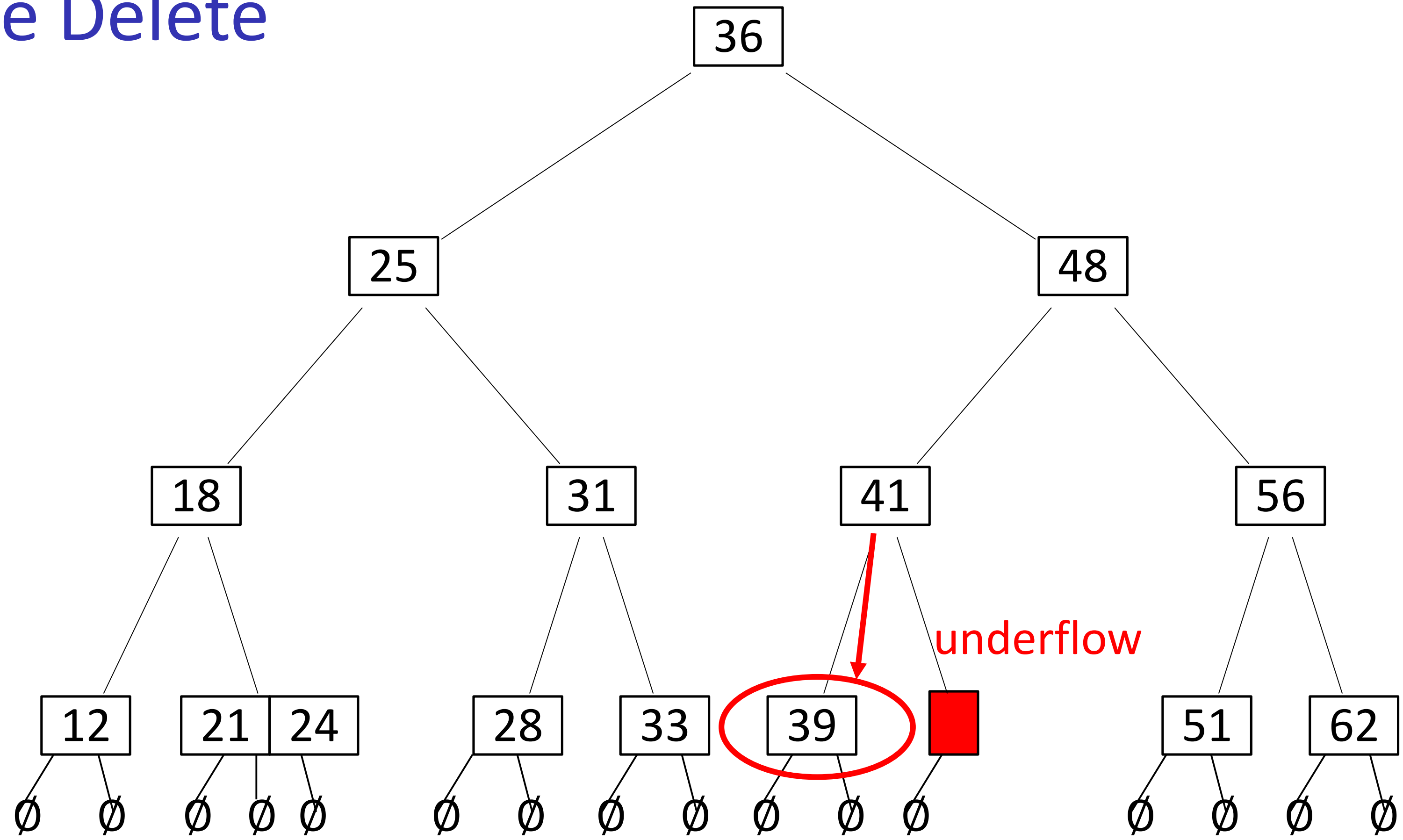
- Example: *delete*(19)
  - immediate siblings exist, but not rich, cannot transfer
    - merge with right immediate sibling with help from parent
      - all subtrees merged together as well
  - structural and order properties are preserved

2-4 Tree Delete



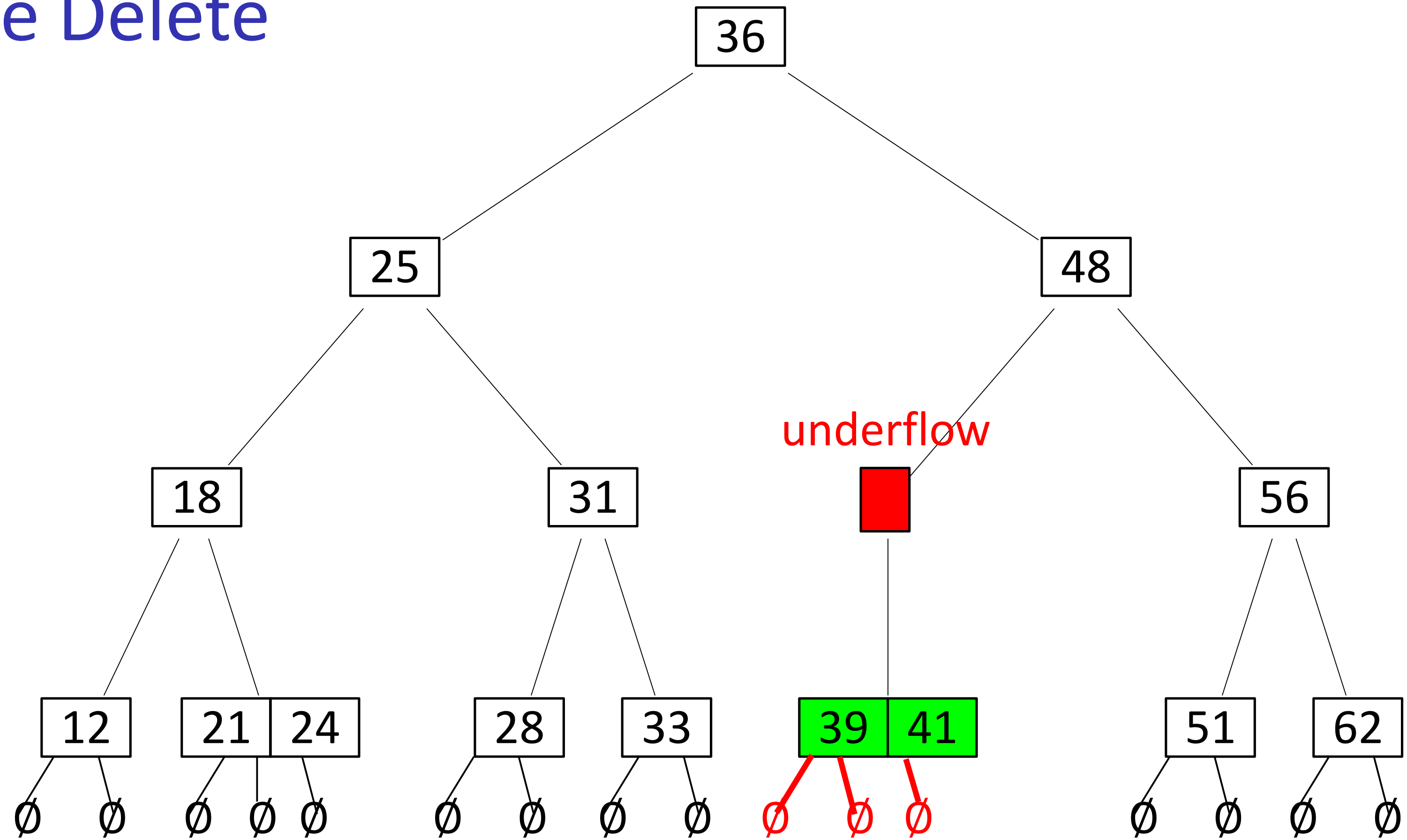
- Example: *delete*(42)
  - first search(42)
  - delete key 42 with one empty subtree

2-4 Tree Delete



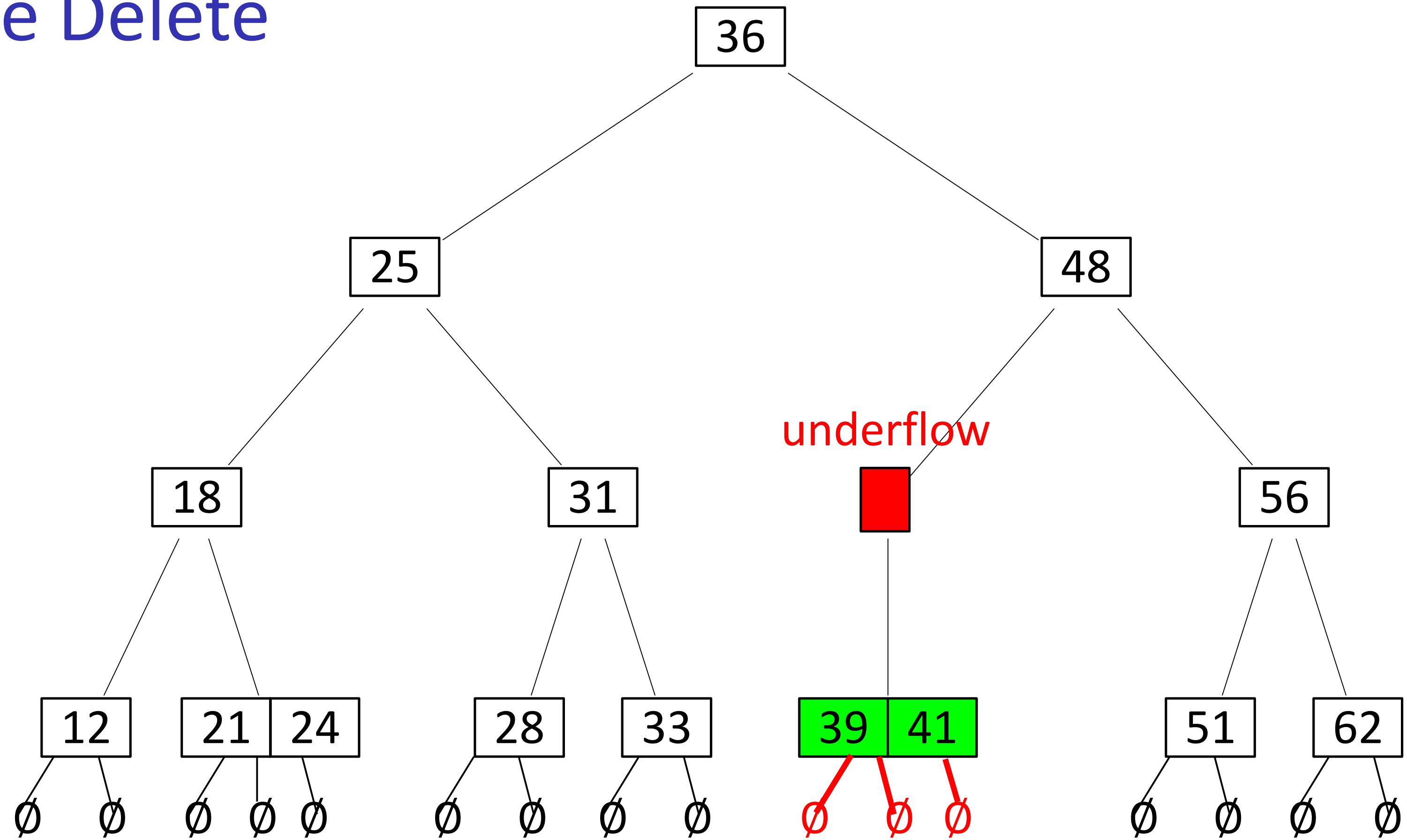
- Example: *delete*(42)
  - first search(42)
  - the only immediate sibling is not rich, perform merge

2-4 Tree Delete



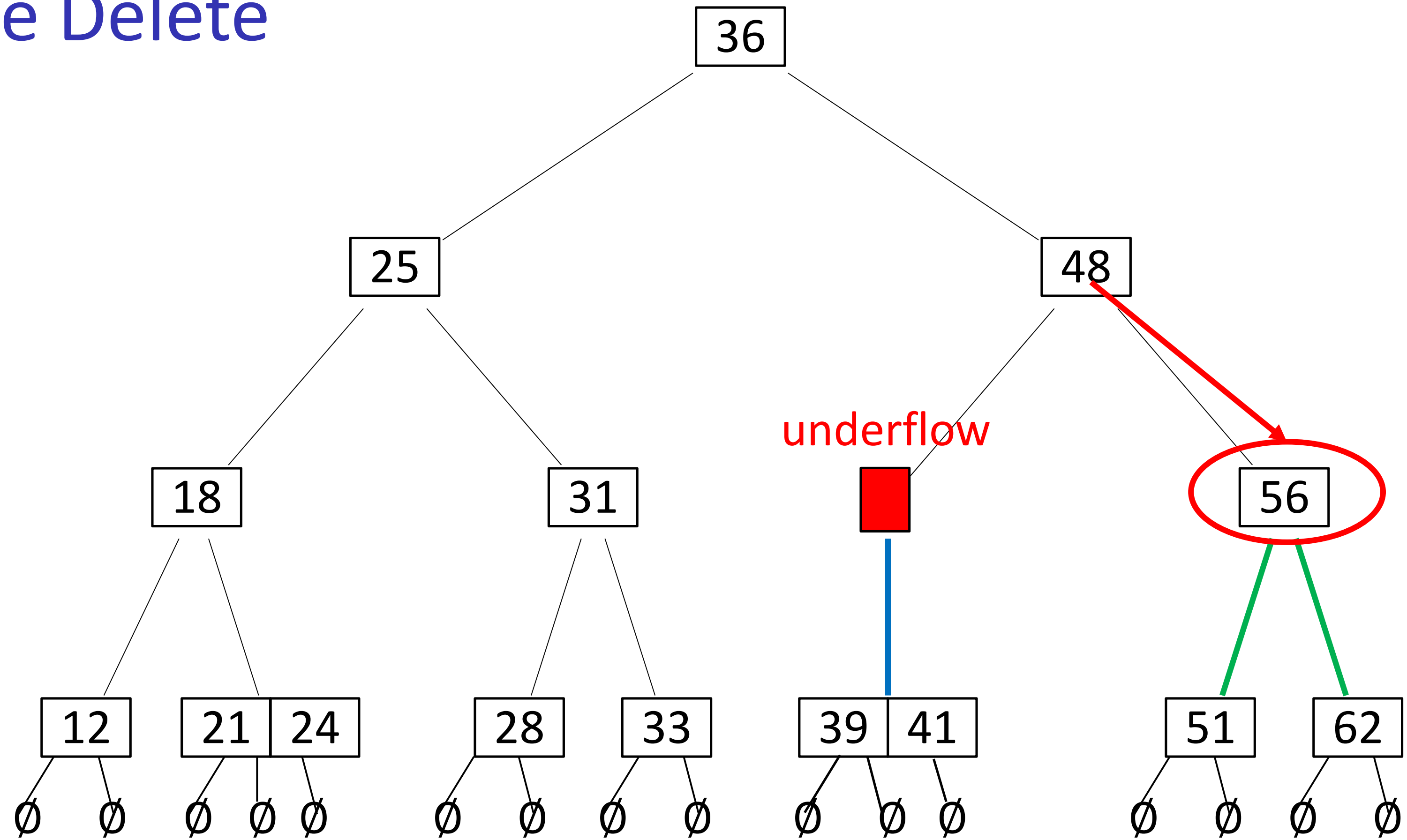
- Example: *delete*(42)
  - first search(42)
  - the only immediate sibling is not rich, perform merge
    - all subtrees merged together as well

2-4 Tree Delete



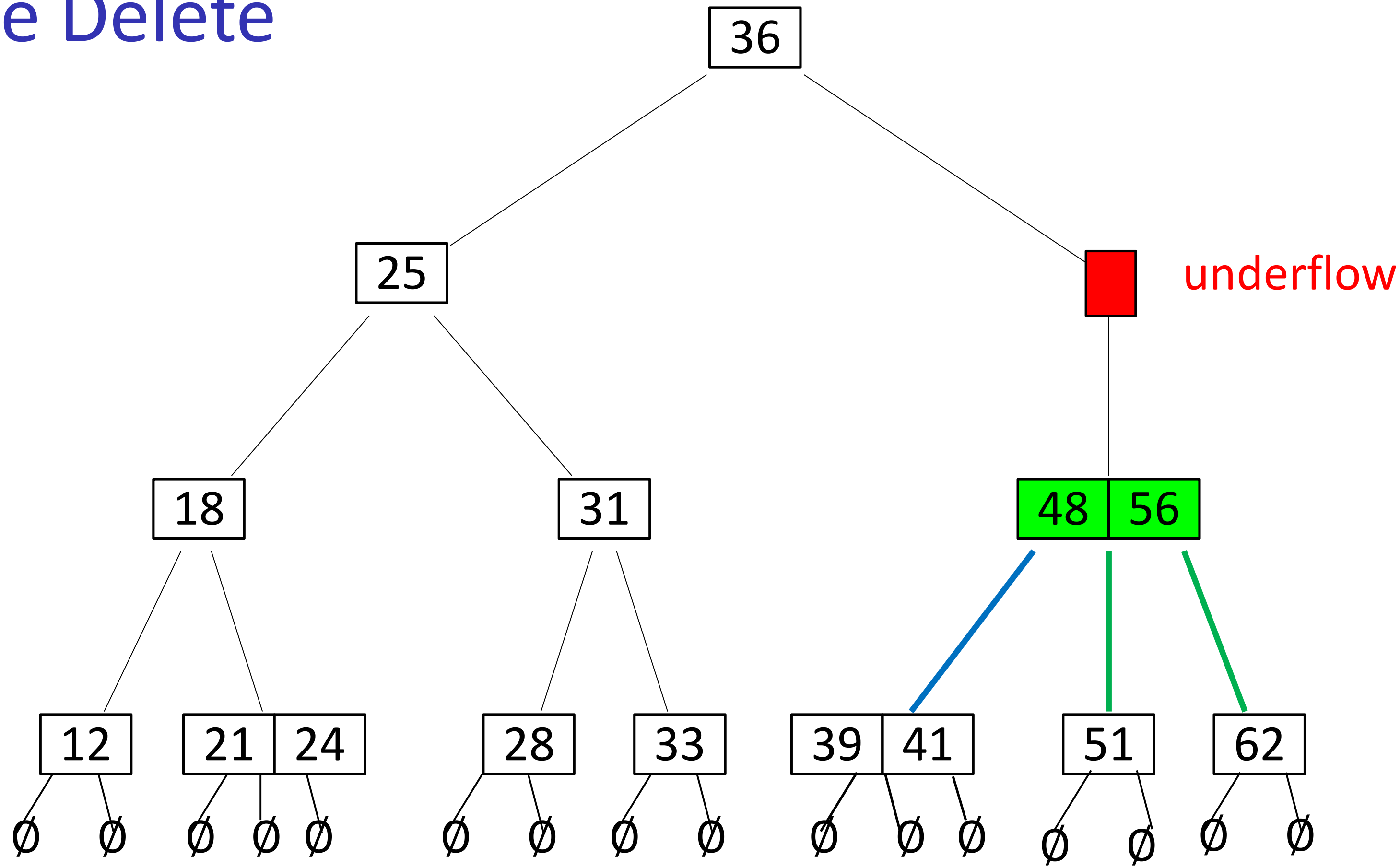
- Example: *delete*(42)
  - merge operation can cause underflow at the parent node
  - while needed, continue fixing the tree upwards
    - possibly all the way to the root

2-4 Tree Delete



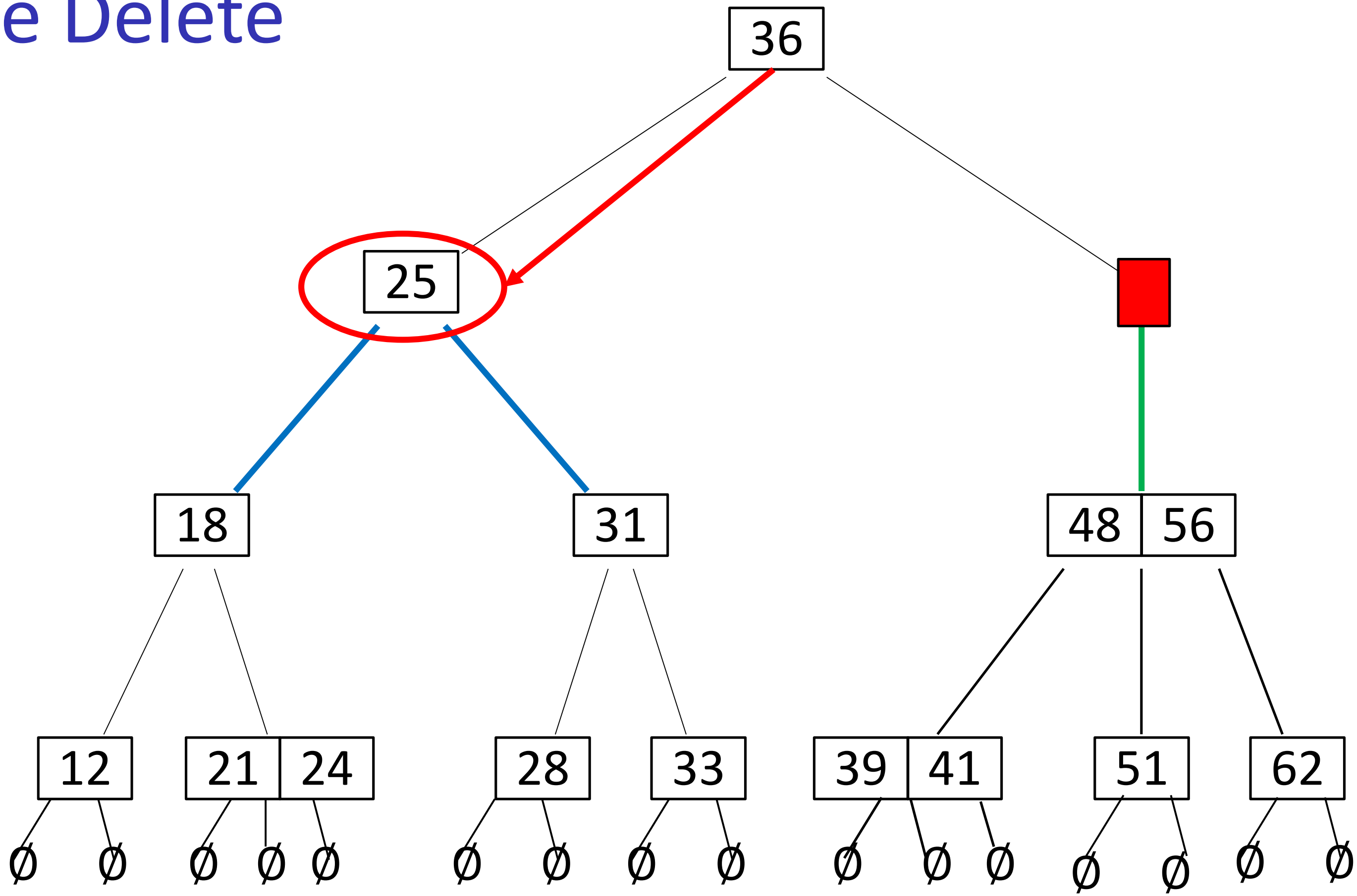
- Example: *delete*(42)
    - the only sibling is not rich, perform a merge

2-4 Tree Delete



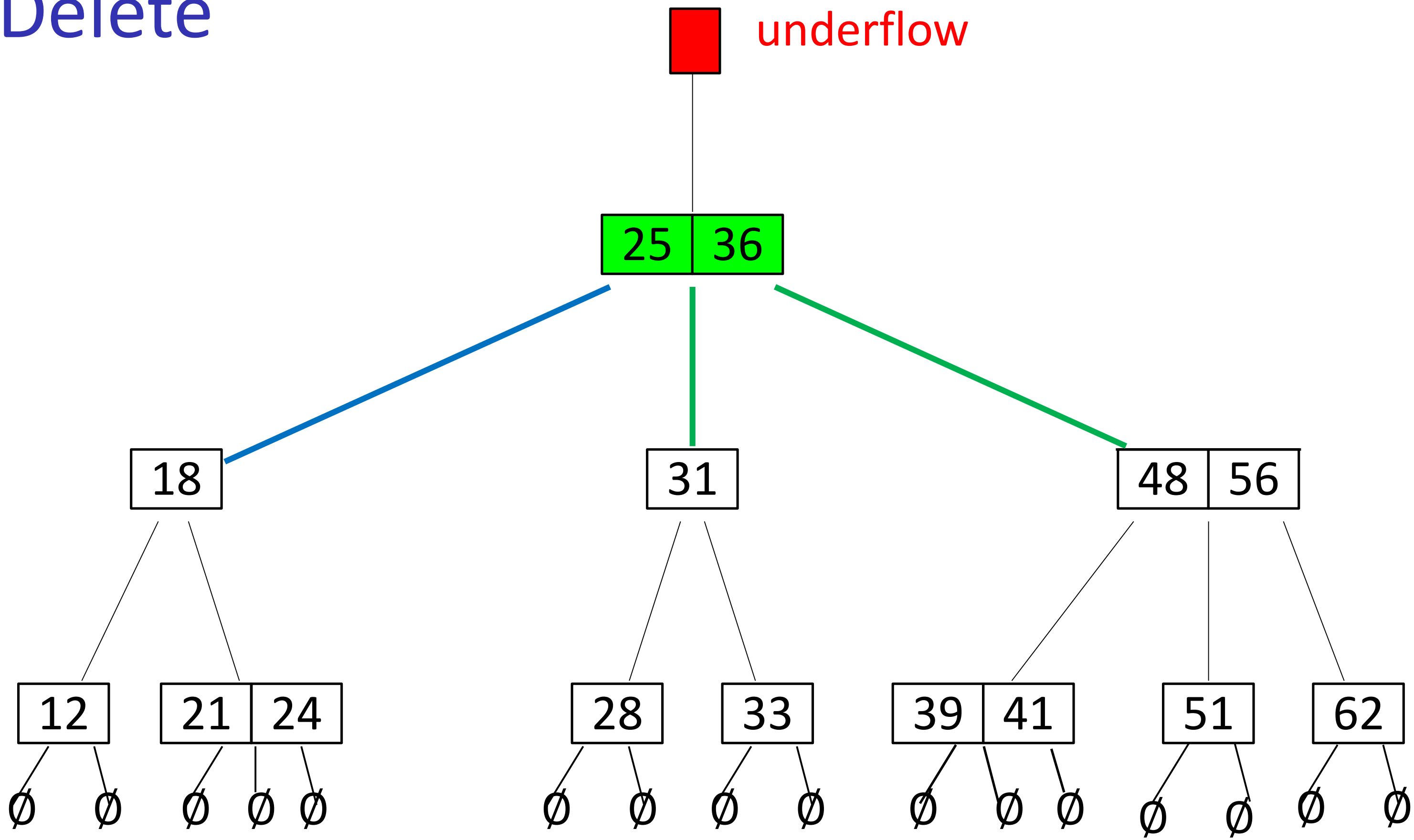
- Example: *delete*(42)
  - the only sibling is not rich, perform a merge
  - subtrees are merged as well
  - continue fixing the tree upwards

2-4 Tree Delete



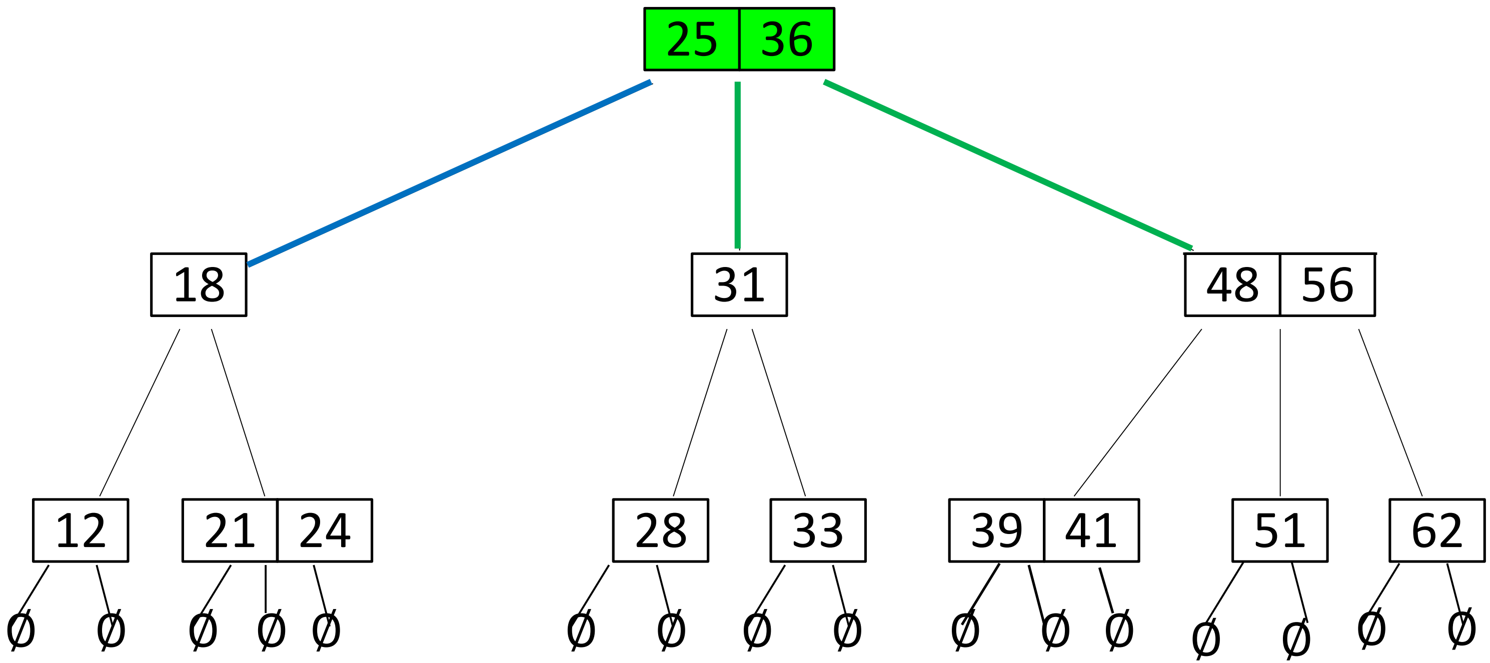
- Example: *delete*(42)
  - the only sibling is not rich, perform a merge

2-4 Tree Delete



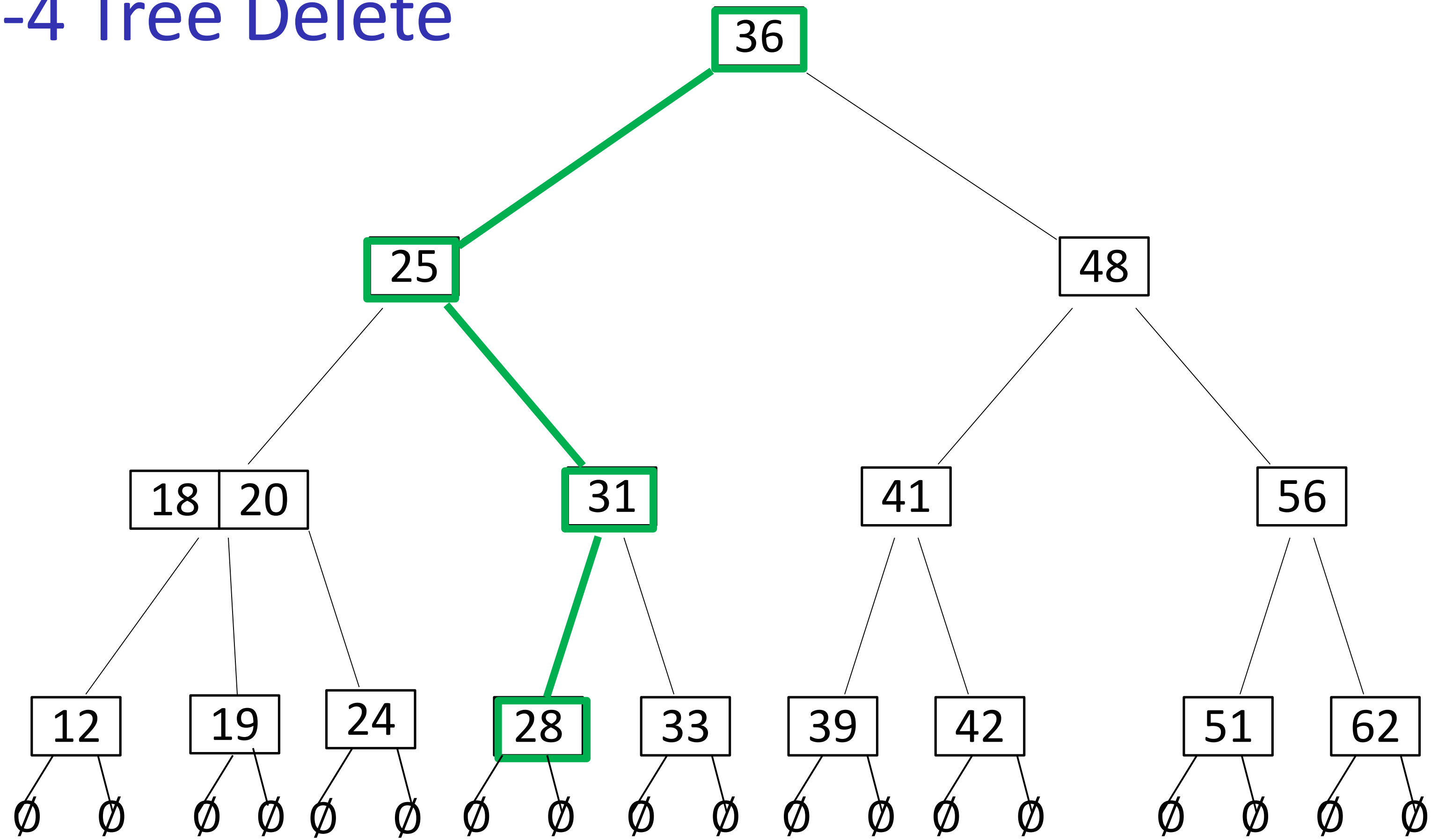
- Example: *delete*(42)
  - the only sibling is not rich, perform merge
  - underflow at parent node
    - it is the root, delete root

2-4 Tree Delete



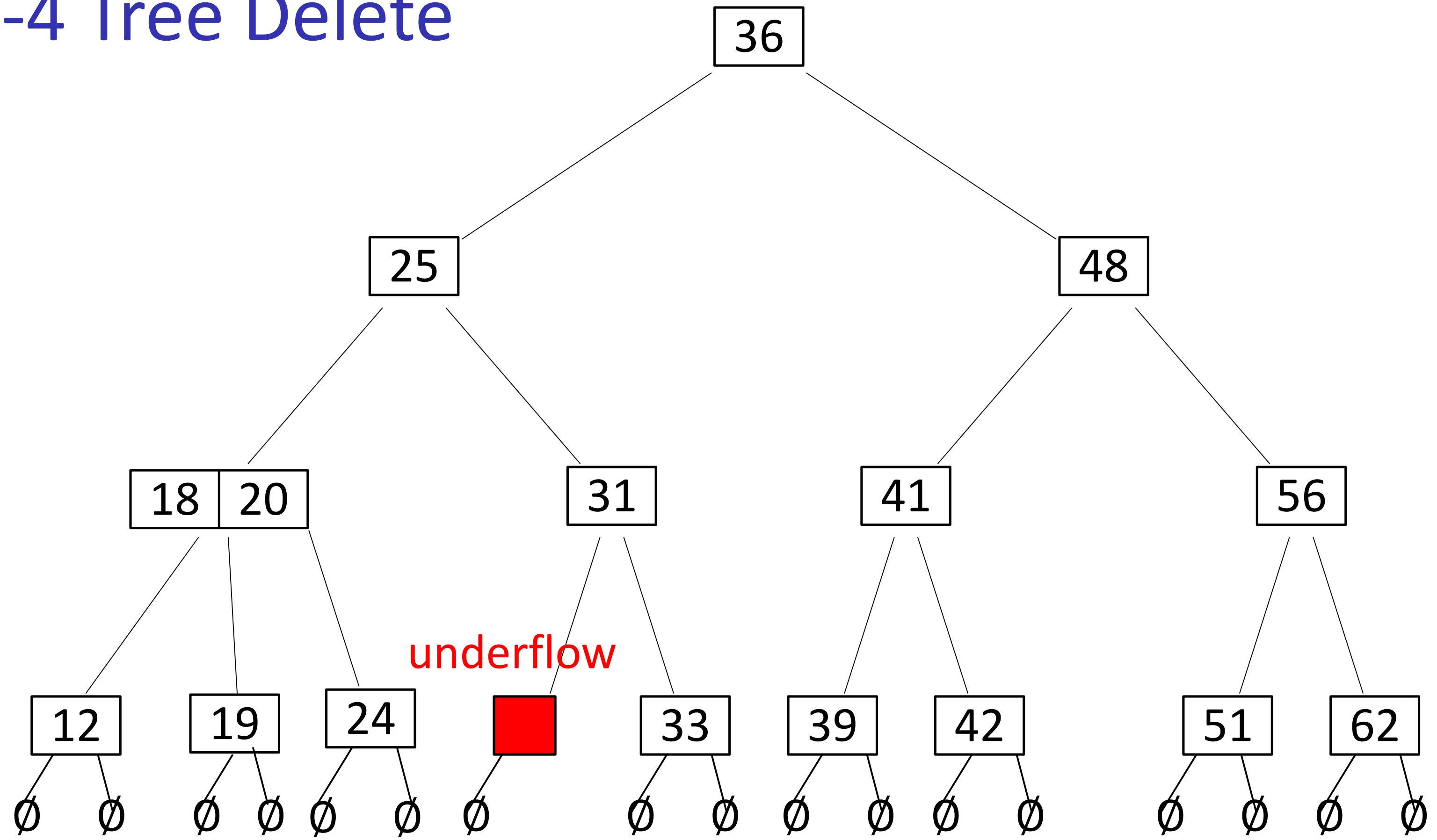
- Example: *delete*(42)
  - the only sibling is not rich, perform merge
  - underflow at parent node
    - it is the root, delete root

2-4 Tree Delete



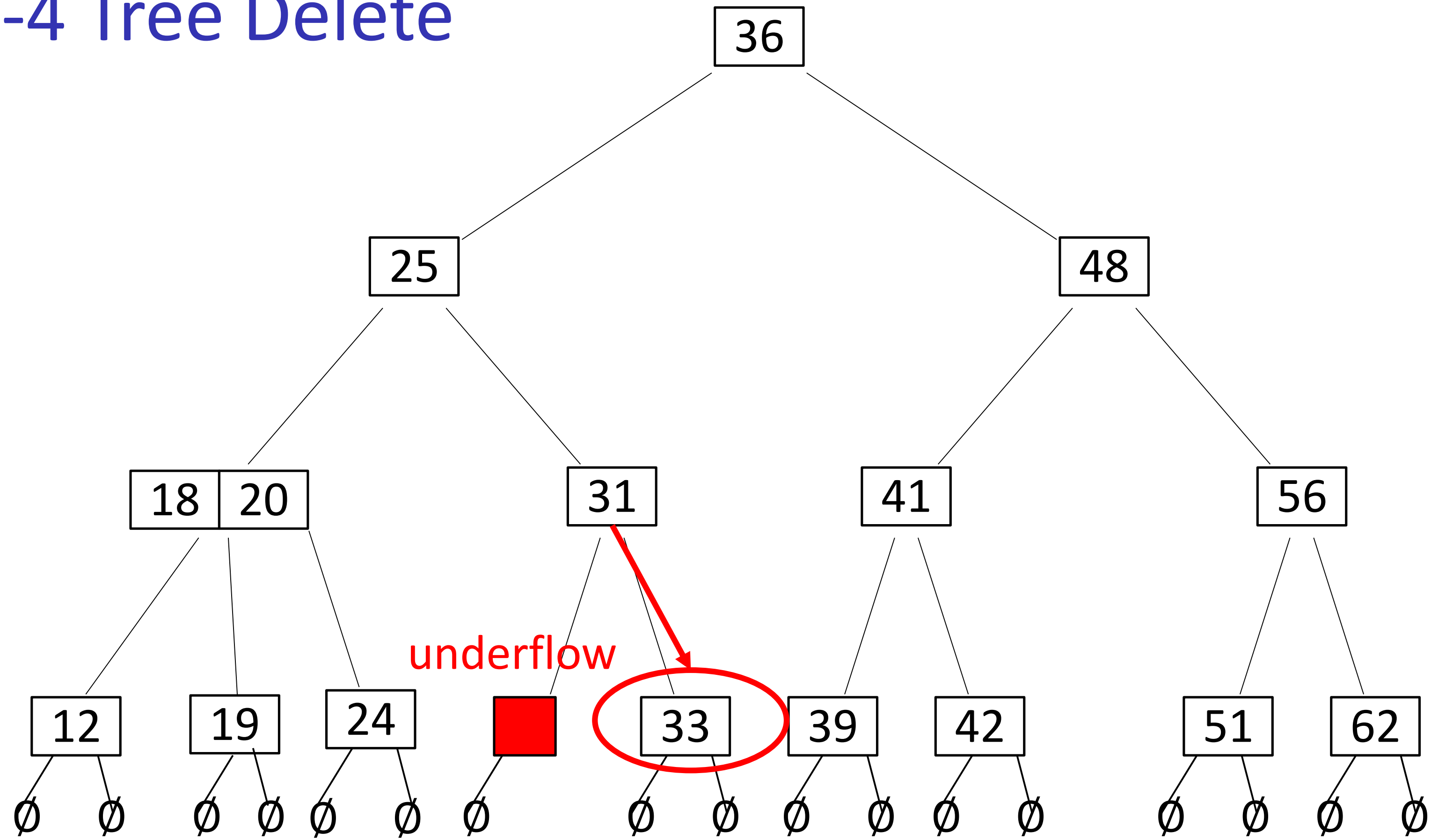
- Example: *delete*(28)
  - first search(28)
  - delete key 28 with one empty subtree

2-4 Tree Delete



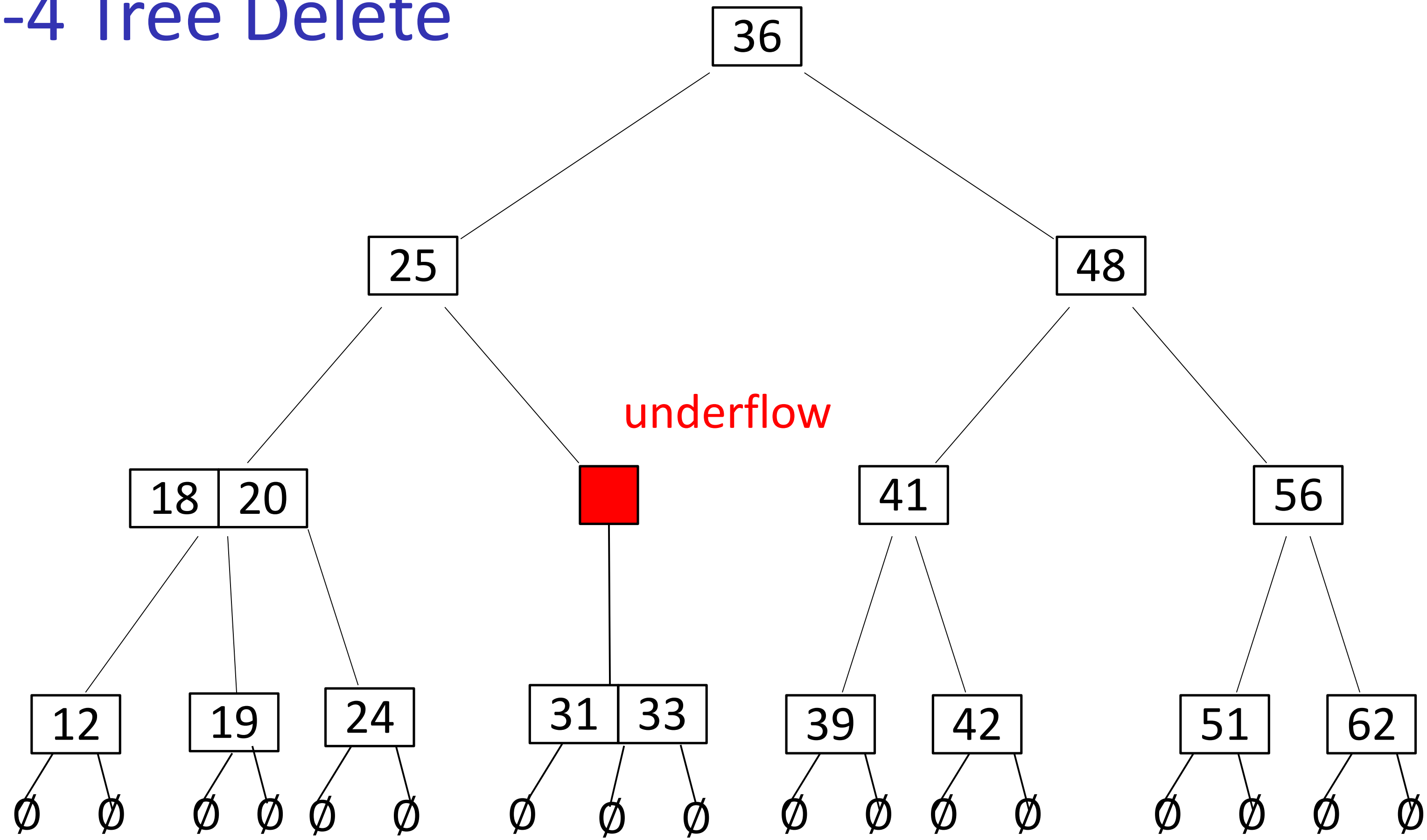
- Example: *delete*(28)
  - first search(28)
  - delete key 28 with one empty subtree

2-4 Tree Delete



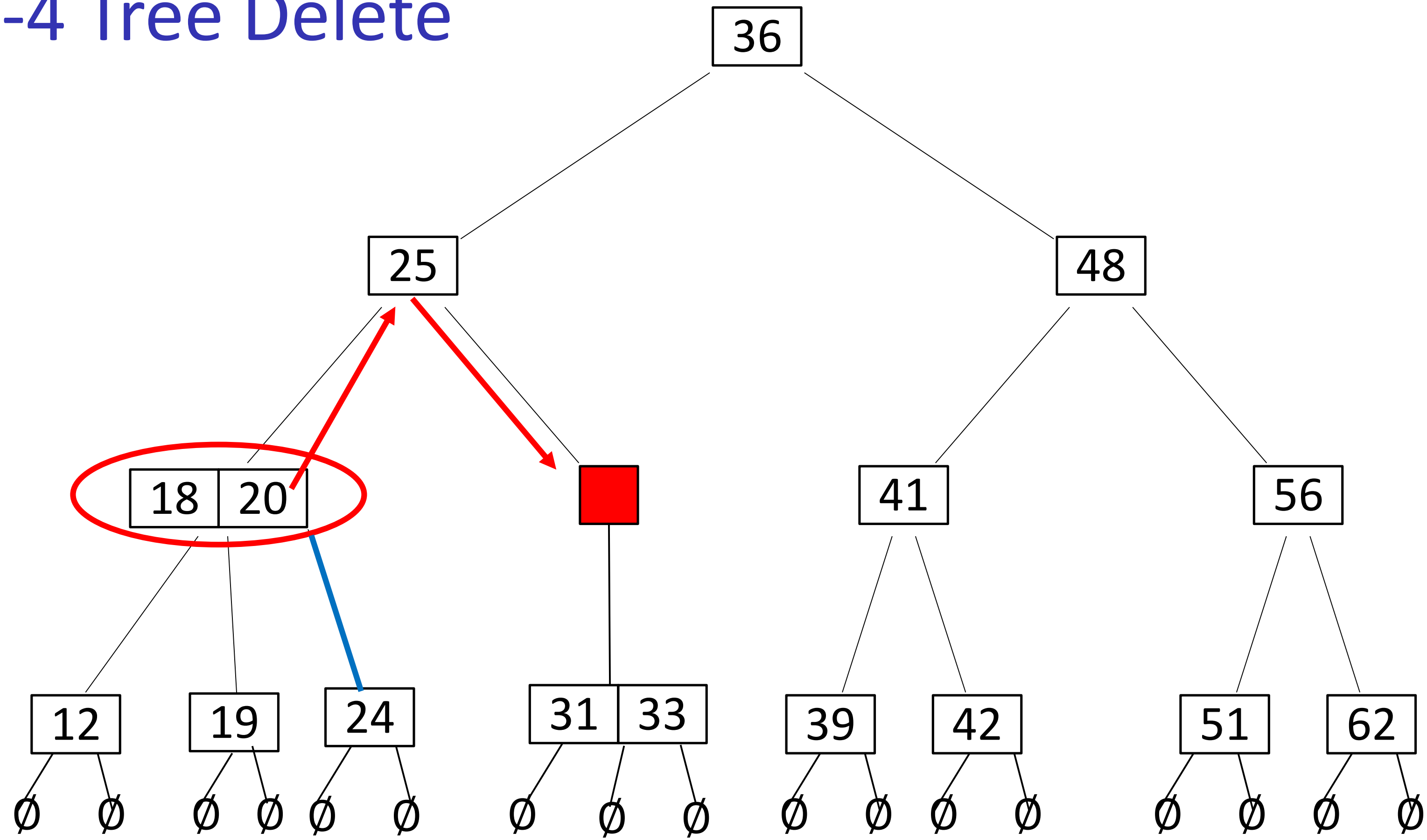
- Example: *delete*(28)
  - first search(28)
  - delete key 28 with one empty subtree
  - merge with the only immediate sibling, who is not rich

2-4 Tree Delete



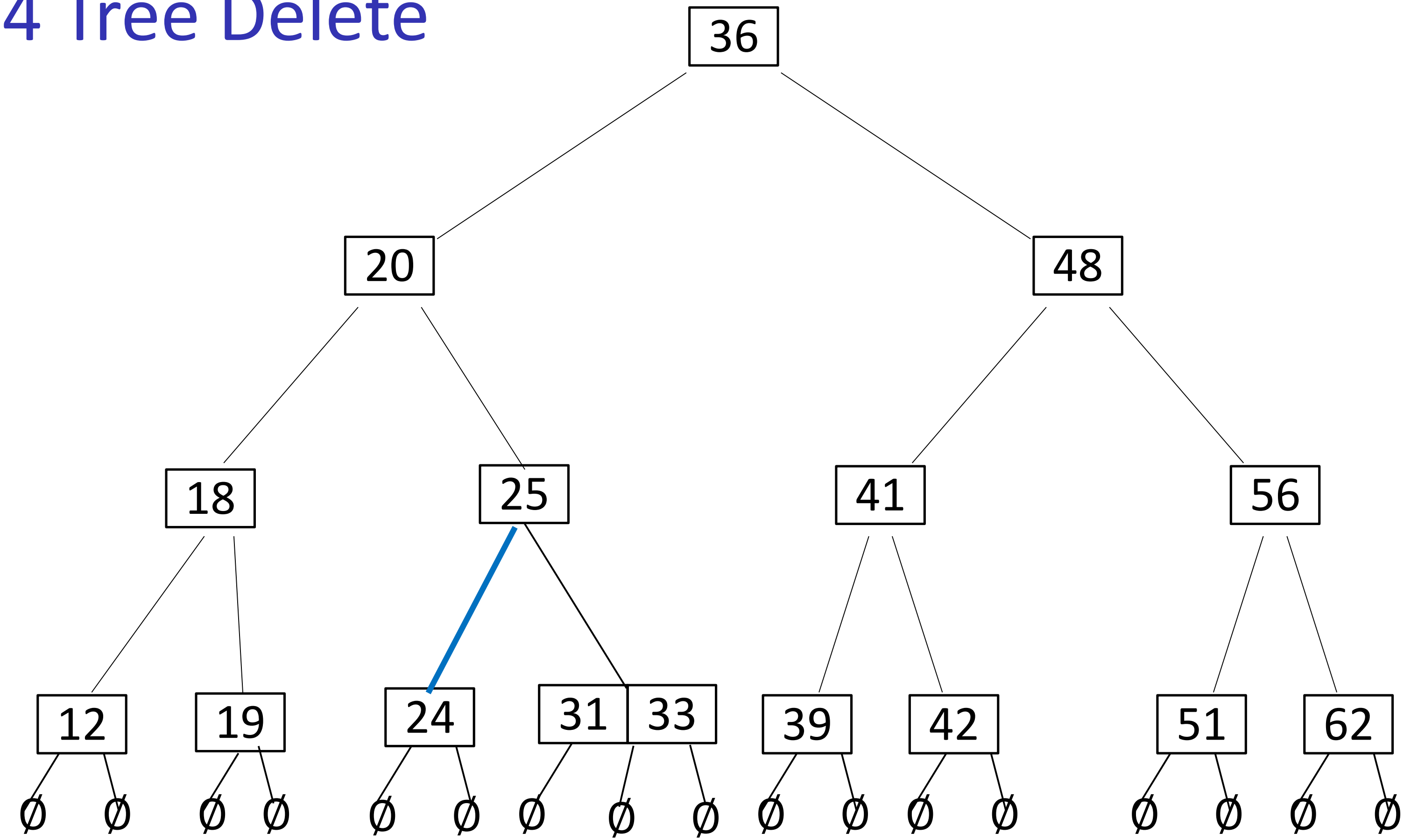
- Example: *delete*(28)
  - first search(28)
  - delete key 28 with one empty subtree
  - merge with the only immediate sibling, who is not rich

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
```

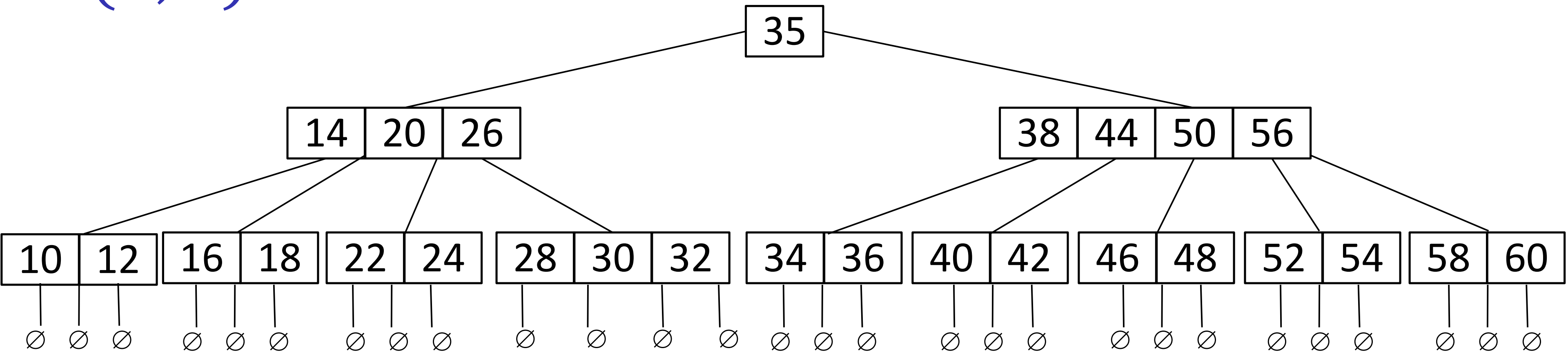
2-4 Tree Summary

- 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
    - no details
- 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
        - (*a*, *b*)-Trees
        - B-Trees

(a,b)-Trees

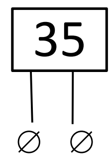


$(3,5)$-tree, also a valid $(3,6)$-tree

- 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: $a \le \left\lceil \frac{b}{2} \right\rceil = \lfloor (b+1)/2 \rfloor$

(a, b)-Trees: Root

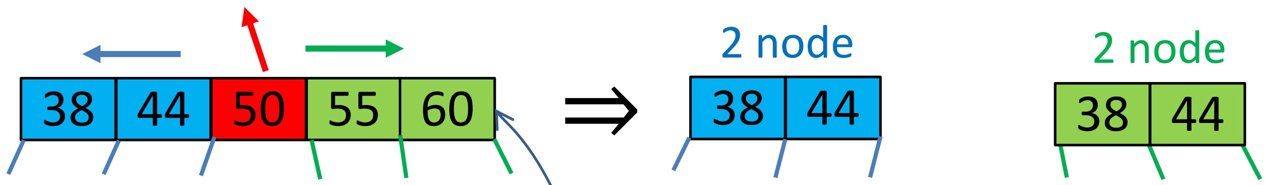
- Why special condition for the root?
- Needed for (a,b)-tree storing very few KVP
- (3,5) tree storing only 1 KVP



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

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

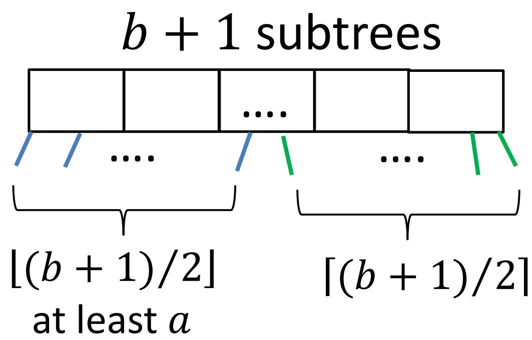
- Because $a \leq \lfloor (b+1)/2 \rfloor$ *search*, *insert*, *delete* work just like for 2-4 trees
  - straightforward redefinition of underflow and overflow
- For example, for $(3{,}5)$-tree
  - at least 3 children, at most 5
    - allowed: 2-node, 3-node, 4-node
  - during insert, overflow if get a 5-node



  - 2-node is smallest allowed node
- If $a > \lfloor (b+1)/2 \rfloor$, no valid split exists for overflowed node
  - this is similar to requiring you 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 $a \leq \lfloor (b+1)/2 \rfloor$
- In general, overflow means node has $b+1$ subtrees
  - split in the middle $\Rightarrow$ two new nodes have $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ subtrees
  - since $a \leq \lfloor (b+1)/2 \rfloor \leq \lceil (b+1)/2 \rceil$, each new node has at least a subtrees, as required

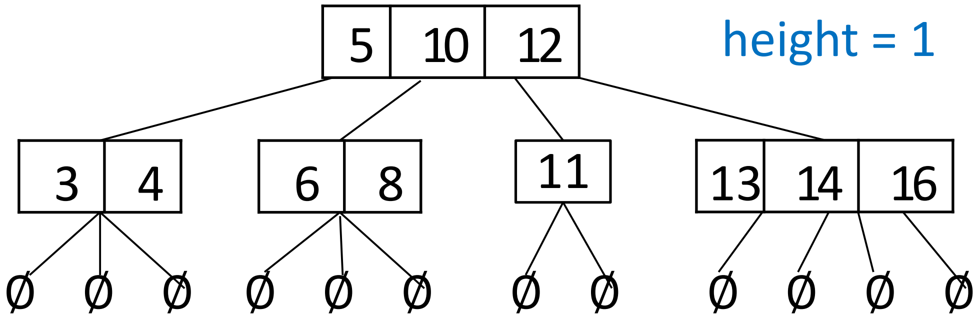


(a, b)-Trees Delete

- 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

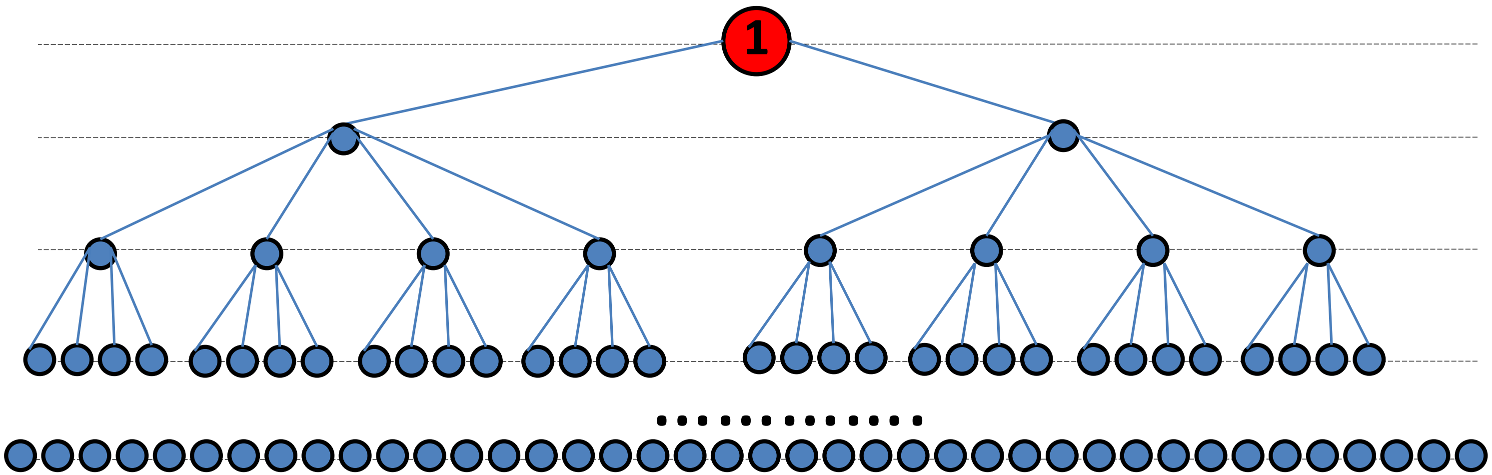
- Height = number of levels **not** counting empty subtrees



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	$2a^{h-1}$



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

- 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 $a \le \left\lceil \frac{b}{2} \right\rceil$ 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

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

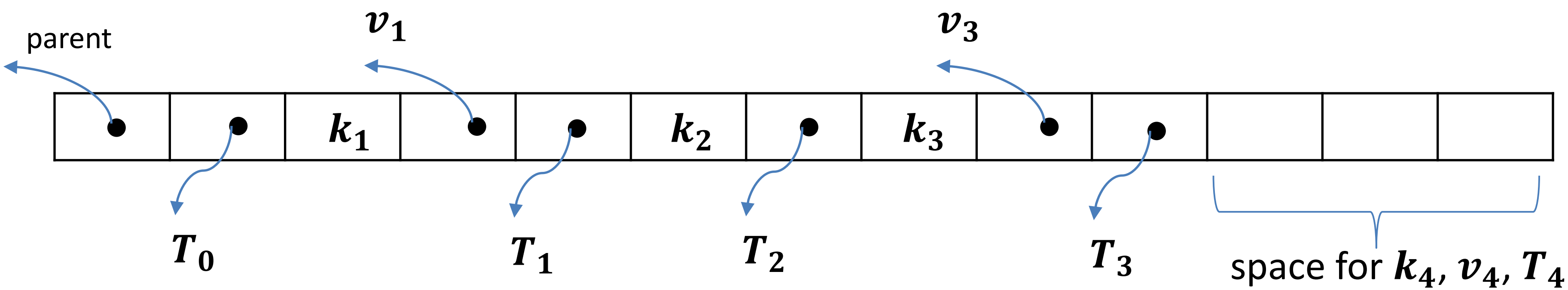
  - this is not better than AVL-trees in internal memory
- External memory
  - we count just block transfers
  - running time is $O(\log n/\log a)$, assuming each node fits into one block
  - makes sense to make a as large as possible so that a node still fits into one block

Outline

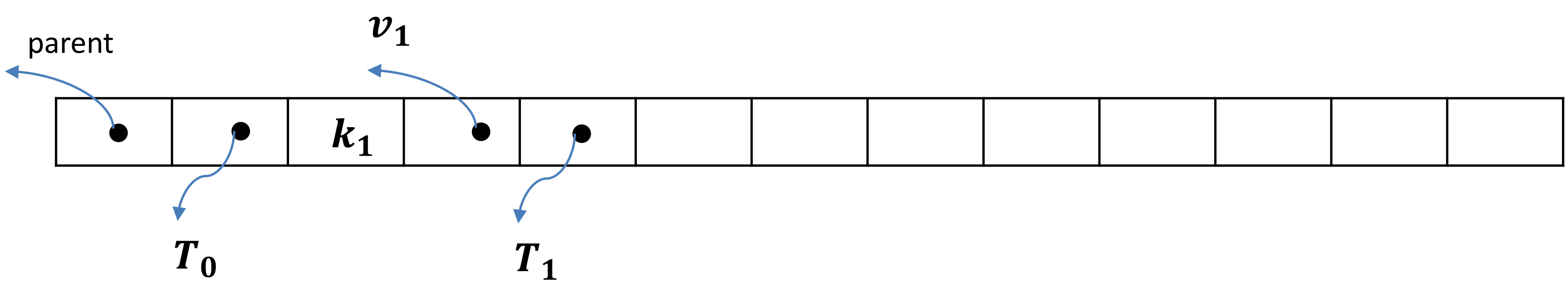
- External Memory
  - Motivation
  - Stream based algorithms
  - External sorting
  - External dictionaries
    - 2-4 Trees
    - (*a*, *b*)-Trees
    - B-Trees

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



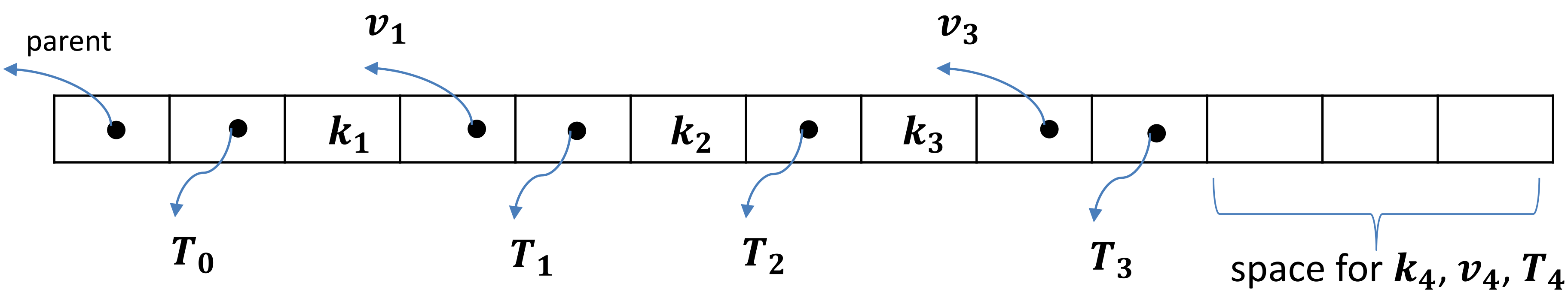
- If a is small, would allow wasting most block space



- Height is $O(\log n / \log a)$, so small a leads to large height and bad running time

B-trees: Definition

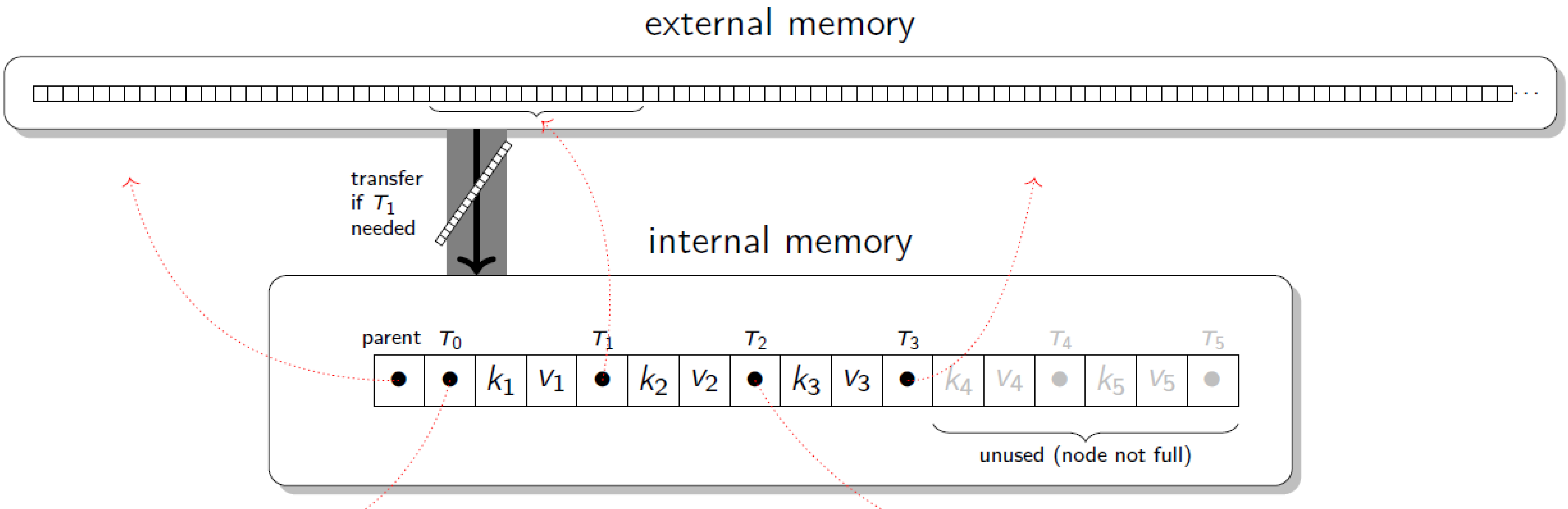
- For external memory use (a, b)-tree s.t.
  - largest possible node (i.e. b subtrees) still fits into a block
  - and a is as large as possible, recall that largest allowed $a = \lceil b/2 \rceil$
  - each block will be at least half full
- Thus use $(\lceil b/2 \rceil, b)$- tree for external memory
- This is defined as B-tree
- We 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
- Example: node for B-tree of order 5



- Typically $b \in \Theta(B)$
  - $B = b * const$

B-trees in External Memory

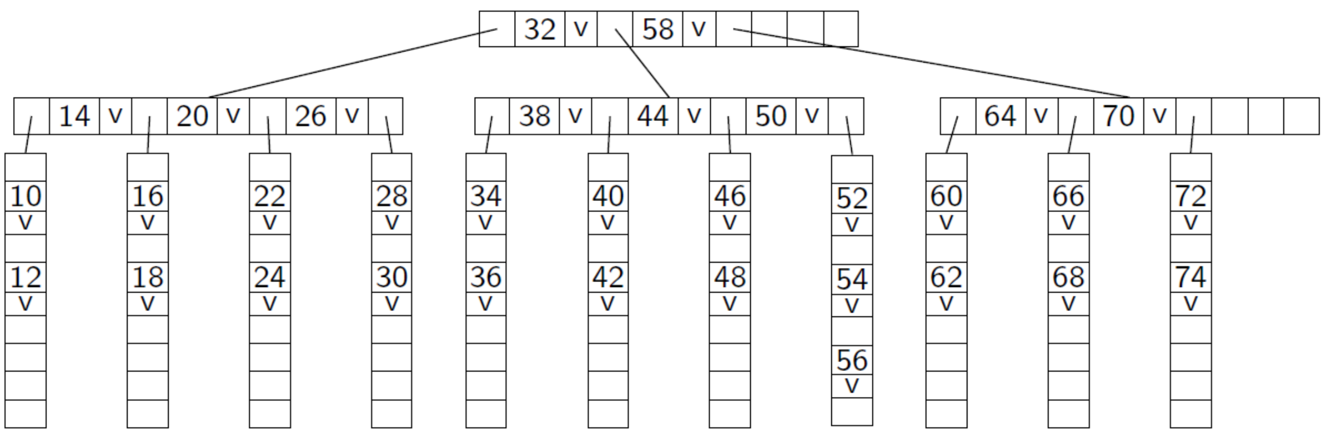
- Close-up on one node in one block



- In this example, 12 references and 5 keys fit into one block, so B-tree can have order 6
- Values can be stored in the block directly if they do not need much space, otherwise store them by reference
  - storing values by reference is ok as we do not need values during tree search

B-tree Analysis in External Memory

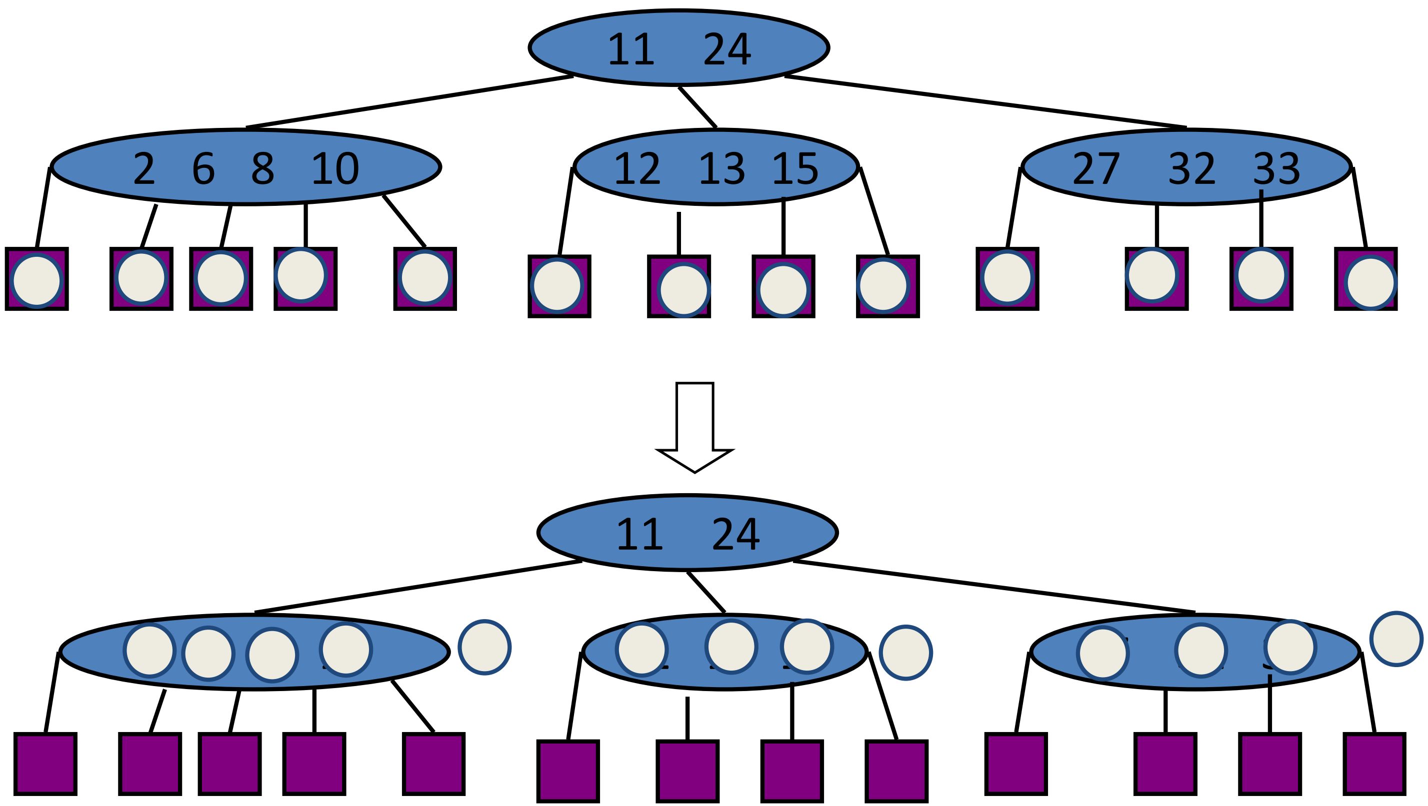
- Search, insert, and delete each requires visiting $\Theta(height)$ nodes
  - $\Theta(height)$ block transfers
- Work within a node is done in internal memory, no block transfers
- The height is $\Theta(\log_b n) = \Theta(\log_B n)$
  - since $b \in \Theta(B)$
- So all operations require $\Theta(\log_B n)$ block transfers
  - this is asymptotically optimal
- There are variants that are even better in practice
- B-trees are hugely important for storing databases (cs448)



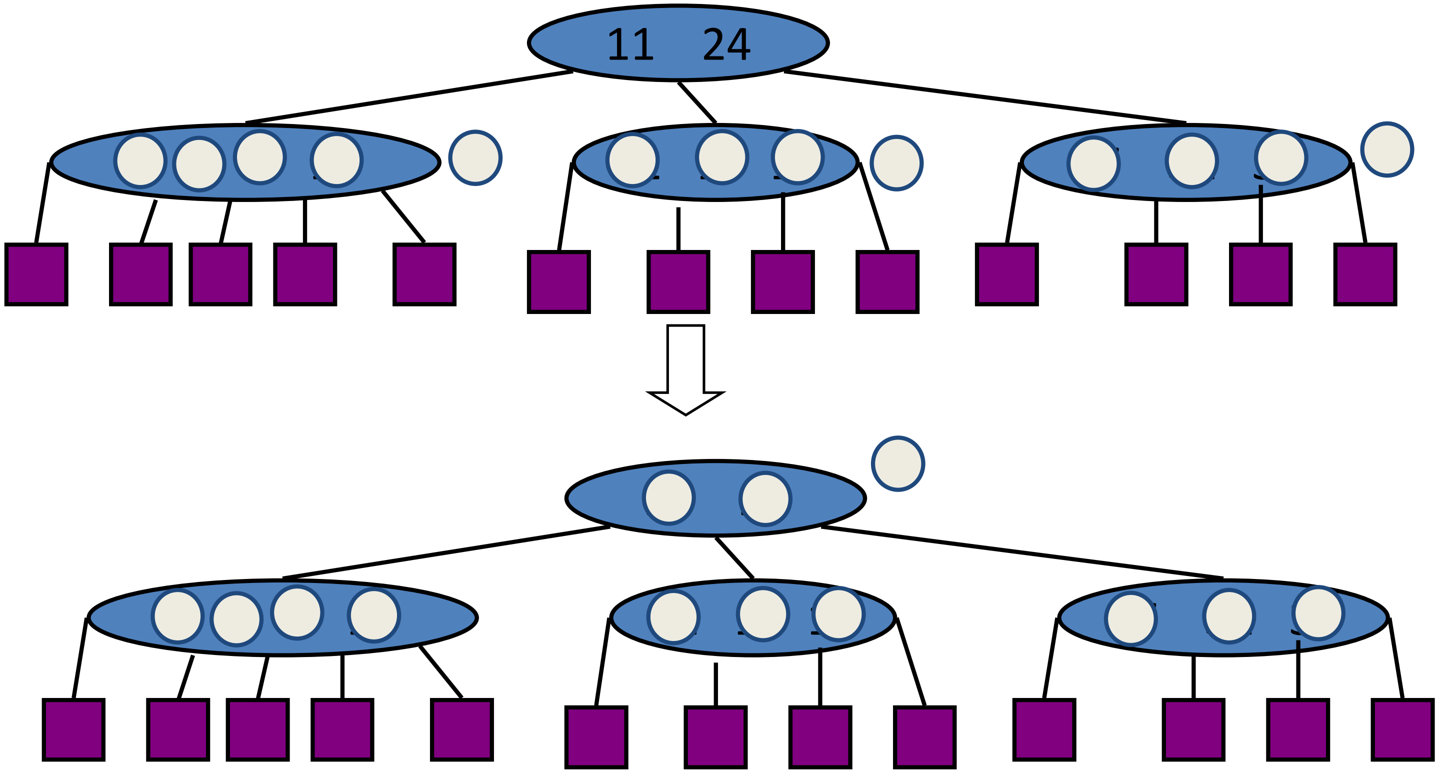
Useful Fact about (a, b)-trees

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

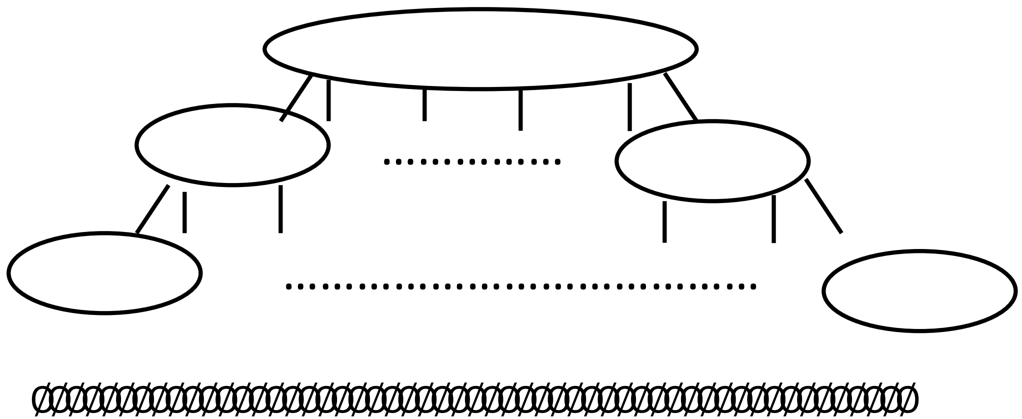
Proof: Put one stone on each empty subtree and pass the stones up the tree. Each node keeps 1 stone per KVP, and passes the rest to its parent. Since for each node, #KVP = # children – 1, each node will pass only 1 stone to its parent. This process stops at the root, and the root will pass 1 stone outside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



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