CS 240E – Data Structures and Data Management (Enriched)

Module 11: External Memory

Therese Biedl

Based on lecture notes by many previous cs240 instructors

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

Winter 2025

version *2025-03-25 15:10*

Outline

Outline

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

A typical current computer architecture includes

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- disk or cloud (slow, very large)

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

A typical current computer architecture includes

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- disk or cloud (slow, very large)

General question: how to adapt our algorithms to take the memory hierarchy into account, avoiding transfers as much as possible?

Different levels of memory

Recall the RAM model of a computer: Any access to a memory location takes the same (constant) time.

This is not at all realistic!

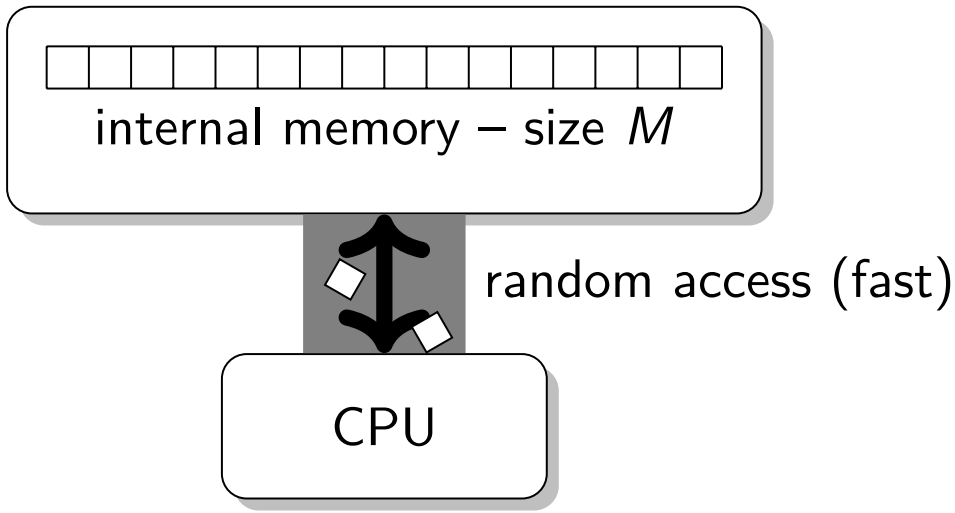
A typical current computer architecture includes

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- disk or cloud (slow, very large)

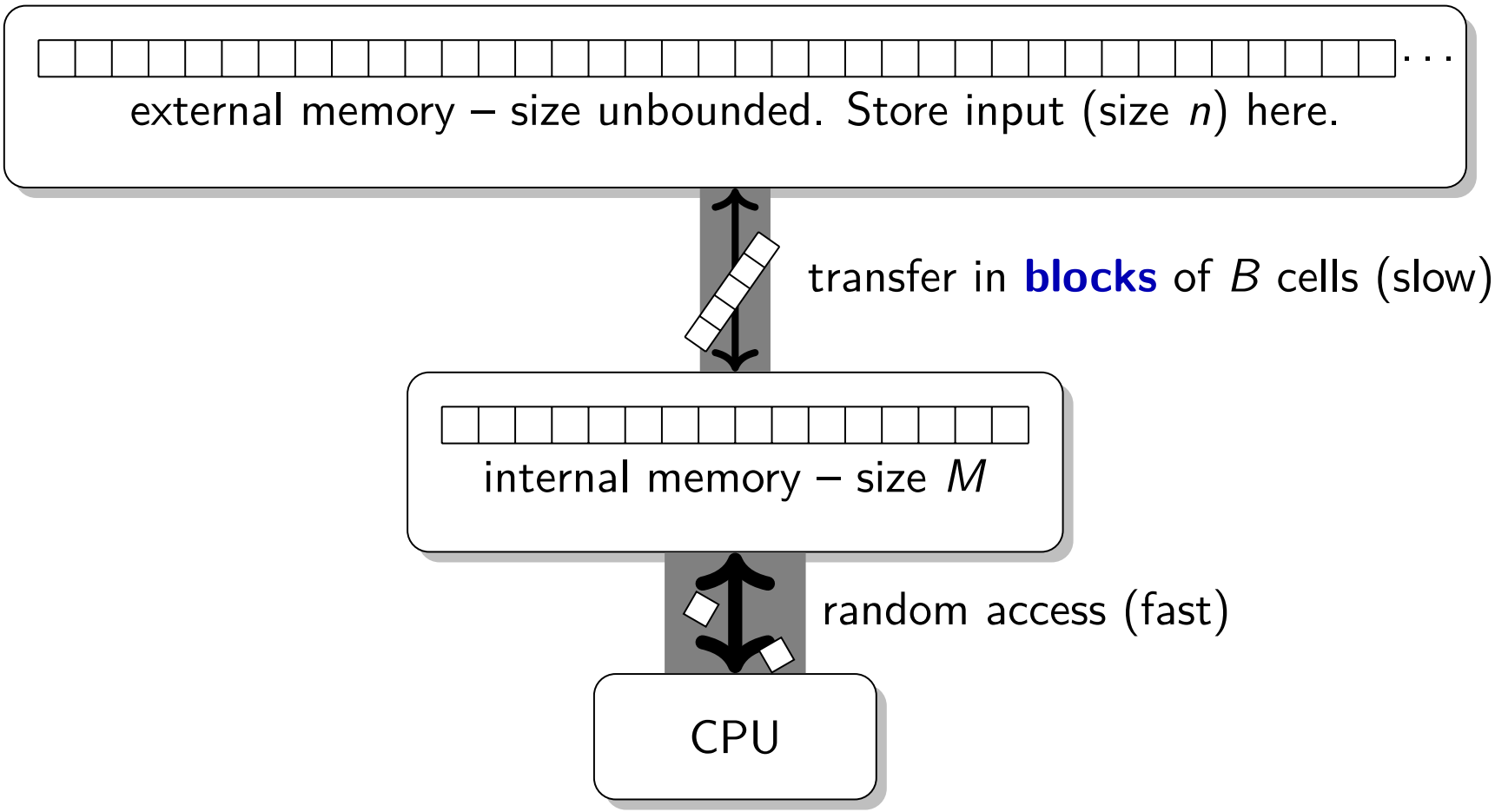
General question: how to adapt our algorithms to take the memory hierarchy into account, avoiding transfers as much as possible?

Define a new computer model that models one such 'gap' across which we must transfer.

The External-Memory Model (EMM)



The External-Memory Model (EMM)



Assumption: During a *transfer*, we automatically load a whole **block** (or "page"). This is quite realistic.

New objective: revisit all algorithms/data structures with the objective of minimizing **block transfers** ("probes", "disk transfers", "page loads")

The External-Memory model

Main objective: minimize the number of block transfers.

- We *completely* ignore all operations done in internal memory.
  (Since these are orders of magnitude faster, this is not unrealistic.)

The External-Memory model

Main objective: minimize the number of block transfers.

- We *completely* ignore all operations done in internal memory. (Since these are orders of magnitude faster, this is not unrealistic.)
- Our results now depend on three parameters:
  - n—the input size
  - M—the internal memory size
  - B—the block size

The External-Memory model

Main objective: minimize the number of block transfers.

- We *completely* ignore all operations done in internal memory. (Since these are orders of magnitude faster, this is not unrealistic.)
- Our results now depend on three parameters:
  - n—the input size ("typical": $n = 2^{50}$)
  - M—the internal memory size ("typical": $M = 2^{30}$)
  - B—the block size ("typical": $B = 2^{15}$)
- The actual values of n, M, B depend much on the application, but we sometimes use "typical" numbers to get a better feel for the bounds.

  For example, how much worse is $n \log n$ compared to $\frac{n}{B} \log_{M/B}(\frac{n}{M})$?

The External-Memory model

Main objective: minimize the number of block transfers.

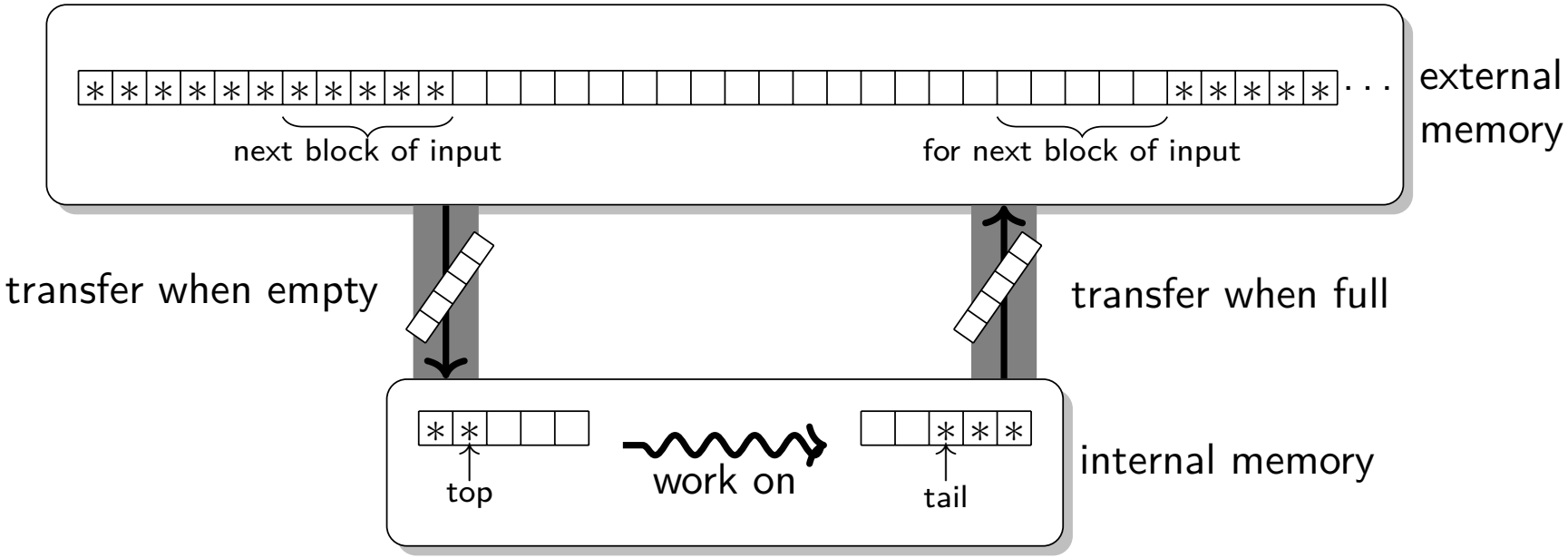
- We *completely* ignore all operations done in internal memory. (Since these are orders of magnitude faster, this is not unrealistic.)
- Our results now depend on three parameters:
  - n—the input size ("typical": $n = 2^{50}$)
  - M—the internal memory size ("typical": $M = 2^{30}$)
  - B—the block size ("typical": $B = 2^{15}$)
- The actual values of n, M, B depend much on the application, but we sometimes use "typical" numbers to get a better feel for the bounds.

  For example, how much worse is $n \log n$ compared to $\frac{n}{B} \log_{M/B}(\frac{n}{M})$?
- Some results will assume that we *know* M, B. This is unrealistic, and "cache-oblivious" results are preferred.
- Some results will also be interesting for the "standard" (RAM) computer model where we do count operations in internal memory.

Outline

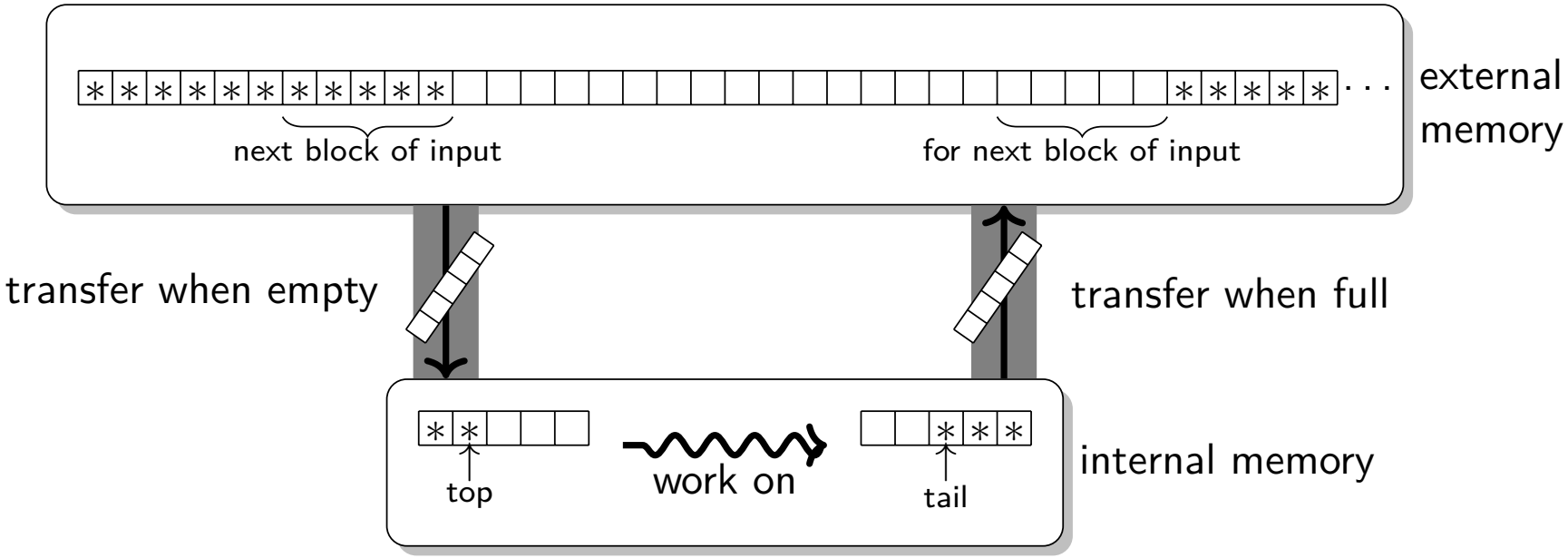
Streams and external memory

Stream-based algorithms (with $O(1)$ resets) use $\Theta(\frac{n}{B})$ block transfers.



Streams and external memory

Stream-based algorithms (with $O(1)$ resets) use $\Theta(\frac{n}{B})$ block transfers.



So can do the following with $\Theta(\frac{n}{B})$ block transfers:

- Text compression: Huffman, Lempel-Ziv-Welch (but not BWT)
- Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore (This assumes internal memory has $O(|P|)$ space.)
- We will revisit Sorting and Dictionaries below.

Outline

Sorting in external memory

As usual: Sort an array A of n elements.

Now assume n is huge and A is stored in blocks in external memory.

- *heap-sort* was optimal in time and space in RAM model
- But: *heap-sort* Heapsort accesses A at indices that are far apart
  $\rightsquigarrow$ typically one block transfer per array access
  $\rightsquigarrow$ typically $\Theta(n \log n)$ block transfers.
  Can we do better?

Sorting in external memory

As usual: Sort an array A of n elements.

Now assume n is huge and A is stored in blocks in external memory.

- *heap-sort* was optimal in time and space in RAM model
- But: *heap-sort* Heapsort accesses A at indices that are far apart
  $\rightsquigarrow$ typically one block transfer per array access
  $\rightsquigarrow$ typically $\Theta(n \log n)$ block transfers.
  Can we do better?

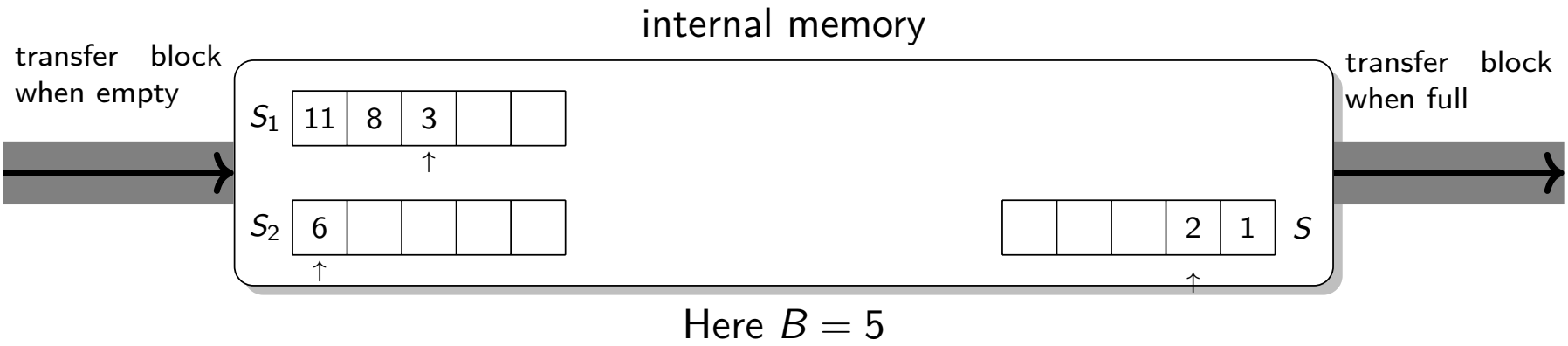
- *merge-sort* adapts well to external memory. Recall algorithm:
  - Split input in half
  - Sort each half recursively $\rightarrow$ two sorted parts
  - Merge sorted parts.

  Key idea: *merge* can be done $O(n/B)$ block-transfers.

Merging with streams

merge(S_1, S_2, S)
S_1, S_2: input streams have items in sorted order, S: output stream

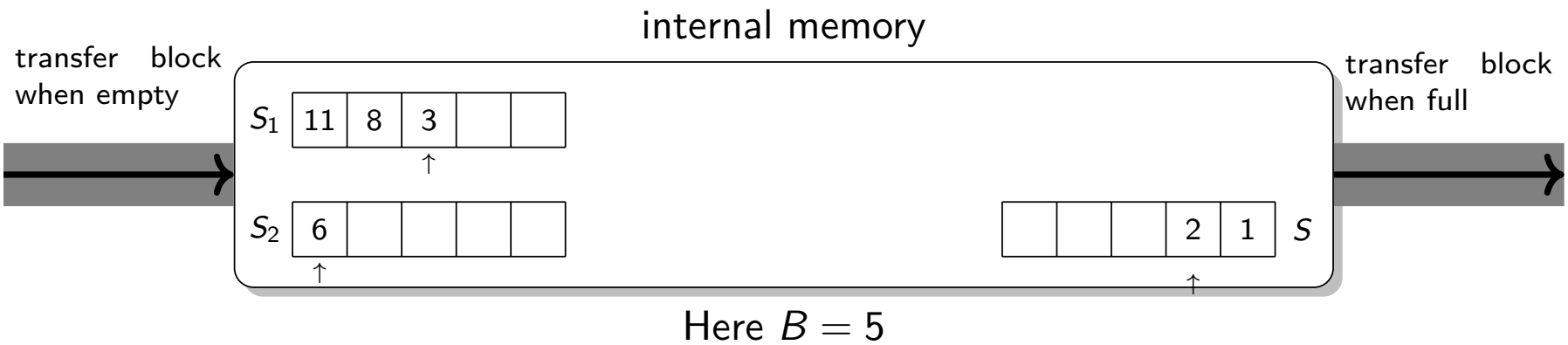
1. **while** S_1 or S_2 is not empty **do**
2. **if** (S_1 is empty) S.*append*(S_2.*pop*())
3. **else if** (S_2 is empty) S.*append*(S_1.*pop*())
4. **else if** (S_1.*top*() $<$ S_2.*top*()) S.*append*(S_1.*pop*())
5. **else** S.*append*(S_2.*pop*())



Merging with streams

merge(S_1, S_2, S)
S_1, S_2: input streams have items in sorted order, S: output stream

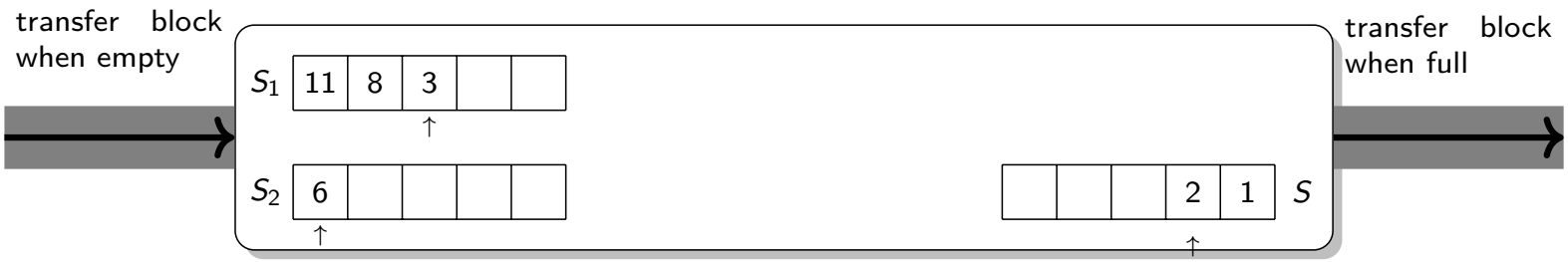
1. **while** S_1 or S_2 is not empty **do**
2. **if** (S_1 is empty) S.*append*(S_2.*pop*())
3. **else if** (S_2 is empty) S.*append*(S_1.*pop*())
4. **else if** (S_1.*top*() $<$ S_2.*top*()) S.*append*(S_1.*pop*())
5. **else** S.*append*(S_2.*pop*())



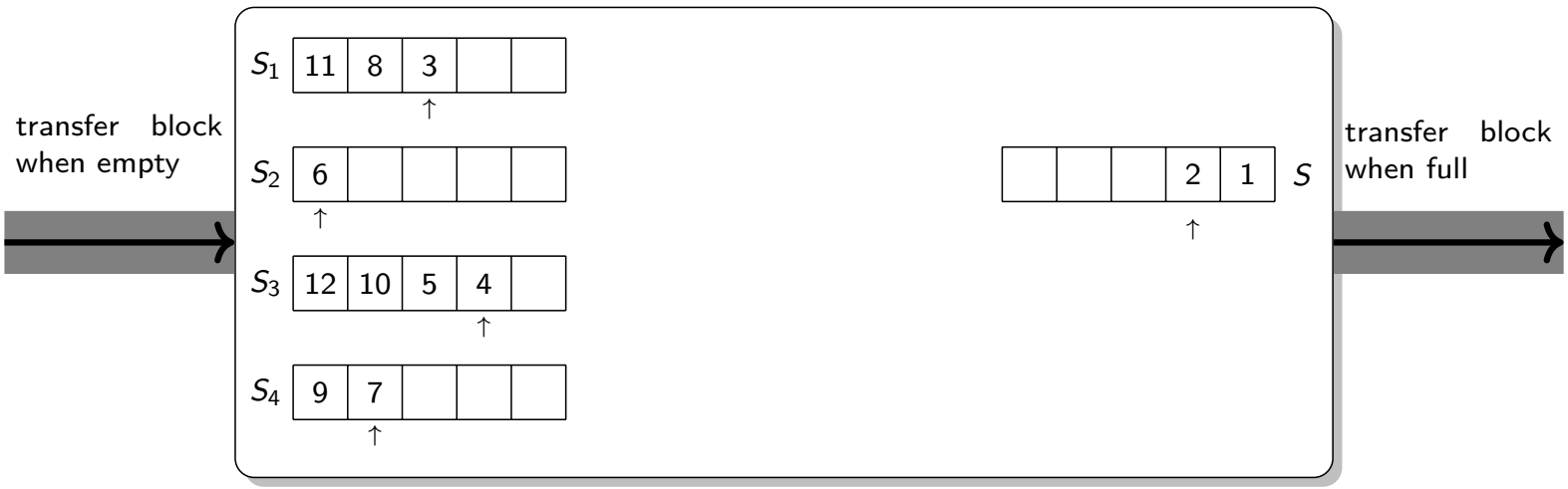
Using this gives $O(\frac{n}{B} \log_2 n)$ block transfers for *merge-sort*.
Not bad, but we can do better!

Towards d-way Mergesort

Observe: We had space left in internal memory during *merge*.



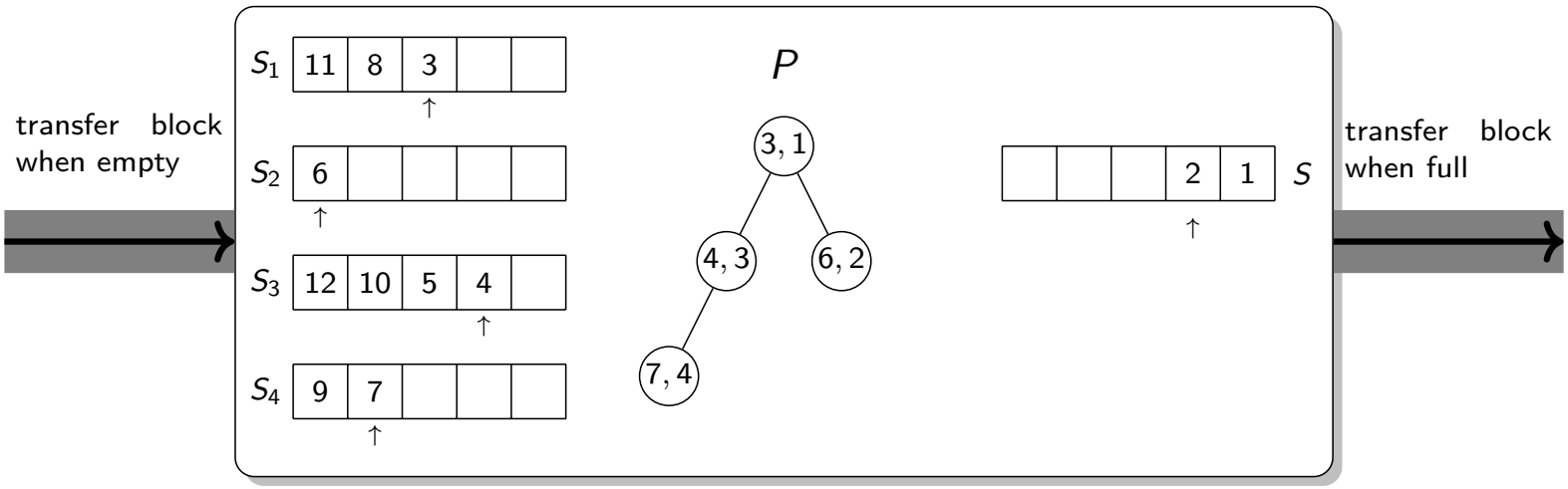
- We use only three blocks, but typically $M \gg 3B$.
- **Idea**: We could merge d parts at once, for some $d \in \Theta(M/B)$.



d-way merge

d-way-merge$(S_1, \ldots, S_d, S)$
$S_1, \ldots, S_d$: input streams have items in sorted order, S: output stream

1. $P \leftarrow$ empty *min-oriented* priority queue
2. **for** $i \leftarrow 1$ to d **do** P.*insert*((S_i.*top*(),i))
   `// each item in P keeps track of its input-steam`
3. **while** P is not empty **do**
4. $(x, i) \leftarrow P$.*deleteMin*()
5. S.*append*(S_i.*pop*())
6. **if** S_i is not empty **do** P.*insert*((S_i.*top*(),i))



d-way merge

- We use a *min-oriented* priority queue P to find the next item to add to the output.
  - This is irrelevant for the number of block transfers.
  - But there is no space-overhead needed for a priority queue. (Recall: heaps are typically implemented as arrays.)
  - And with this the run-time (in RAM-model) is $O(n \log d)$.

d-way merge

- We use a *min-oriented* priority queue P to find the next item to add to the output.
  - This is irrelevant for the number of block transfers.
  - But there is no space-overhead needed for a priority queue. (Recall: heaps are typically implemented as arrays.)
  - And with this the run-time (in RAM-model) is $O(n \log d)$.
- The items in P store not only the next key but also the index of the stream that contained the item.
  - With this, can efficiently find the stream to reload from.

d-way merge

- We use a *min-oriented* priority queue P to find the next item to add to the output.
  - This is irrelevant for the number of block transfers.
  - But there is no space-overhead needed for a priority queue. (Recall: heaps are typically implemented as arrays.)
  - And with this the run-time (in RAM-model) is $O(n \log d)$.
- The items in P store not only the next key but also the index of the stream that contained the item.
  - With this, can efficiently find the stream to reload from.
- We assume d is such that $d+1$ blocks and P fit into internal memory. (Since $|P| \in O(d)$, we have $d \in \Theta(M/B)$.)
- The number of *block transfers* then is $O(\frac{|S_1|+\cdots+|S_d|}{B})$.

  (In "normal" *merge-sort* $|S_1| + \cdots + |S_d| = n$, but this will change soon.)

d-way merge

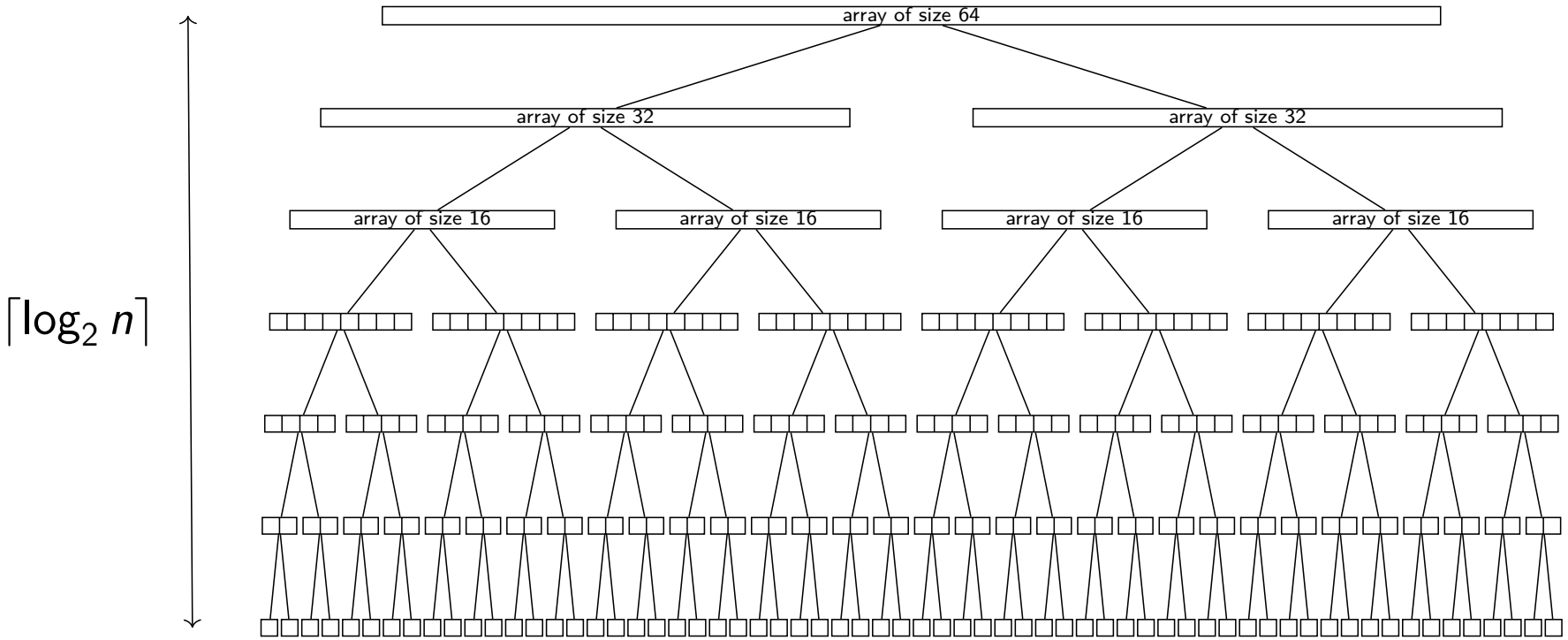
- We use a *min-oriented* priority queue P to find the next item to add to the output.
  - This is irrelevant for the number of block transfers.
  - But there is no space-overhead needed for a priority queue. (Recall: heaps are typically implemented as arrays.)
  - And with this the run-time (in RAM-model) is $O(n \log d)$.
- The items in P store not only the next key but also the index of the stream that contained the item.
  - With this, can efficiently find the stream to reload from.
- We assume d is such that $d+1$ blocks and P fit into internal memory. (Since $|P| \in O(d)$, we have $d \in \Theta(M/B)$.)
- The number of *block transfers* then is $O(\frac{|S_1|+\cdots+|S_d|}{B})$.

  (In "normal" *merge-sort* $|S_1| + \cdots + |S_d| = n$, but this will change soon.)

How does *d-way merge* help to improve external sorting?

Towards d-way Mergesort

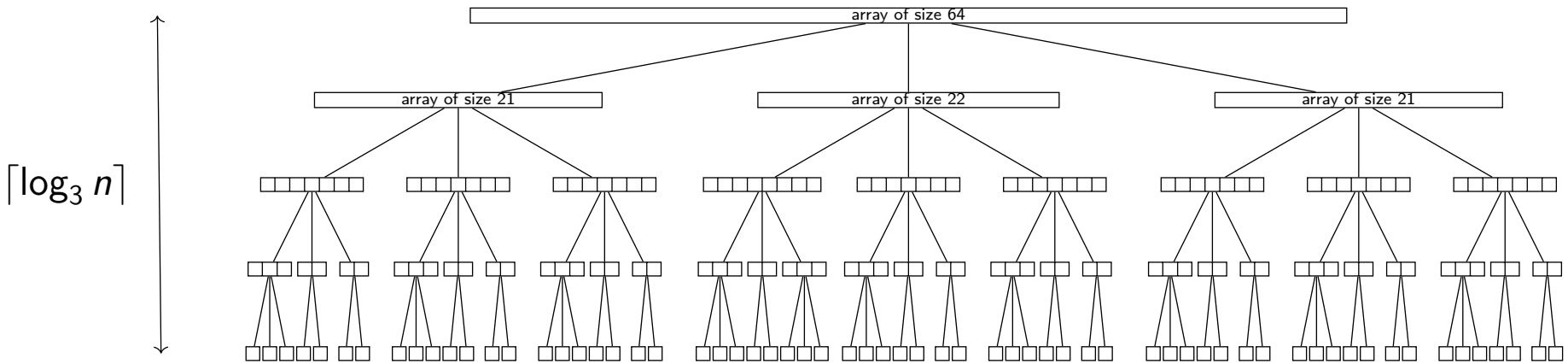
Recall: Mergesort uses $\lceil \log_2 n \rceil$ rounds of splitting-and-merging.



Round: Time spent until all items were part of *merge* once
$\approx$ one level of the recursion tree.

Towards d-way Mergesort

Observe: If we split and merge d-ways, there are fewer rounds.



- Number of rounds is now $\lceil \log_d n \rceil$
- We choose d such that each round uses $\Theta(\frac{n}{B})$ block transfers. (Then the number of block transfers is $\Theta(\log_d n \cdot \frac{n}{B})$.)
- Two further improvements:
  - Proceed bottom-up (while-loops) rather than top-down (recursions).
  - Save more rounds by starting with larger **runs** (sorted subsequences)

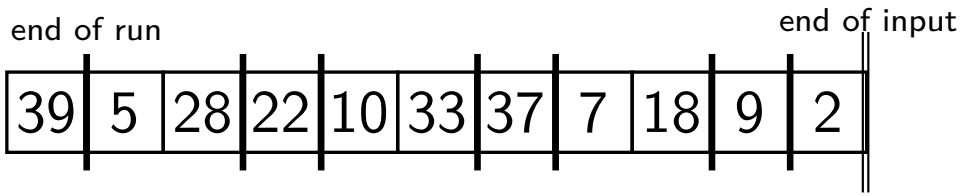
d-way mergesort

- Keep a list L of streams, each stream stores a run
- Initialize L with input split into runs ($\rightsquigarrow$ details later)
- Repeatedly merge the next d runs until only one left.

d-way mergesort

- Keep a list L of streams, each stream stores a run
- Initialize L with input split into runs ($\rightsquigarrow$ details later)
- Repeatedly merge the next d runs until only one left.

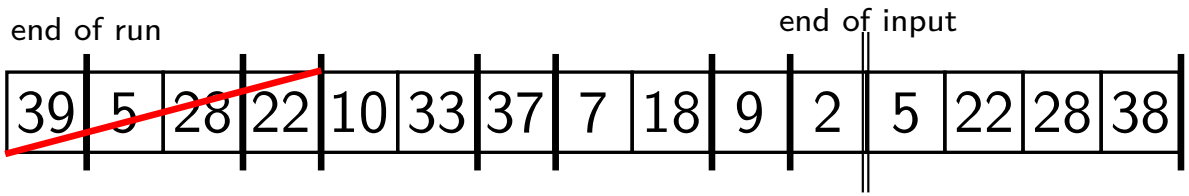
Example ($d = 3$):



d-way mergesort

- Keep a list L of streams, each stream stores a run
- Initialize L with input split into runs ($\rightsquigarrow$ details later)
- Repeatedly merge the next d runs until only one left.

Example ($d = 3$):

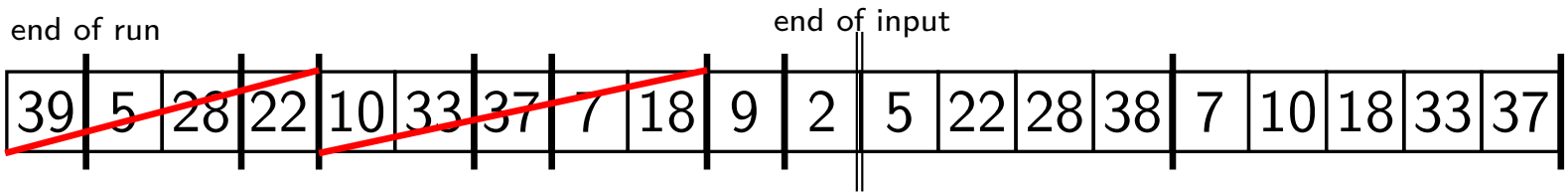


- Merge the first 3 runs with *3-way-merge*.

d-way mergesort

- Keep a list L of streams, each stream stores a run
- Initialize L with input split into runs ($\rightsquigarrow$ details later)
- Repeatedly merge the next d runs until only one left.

Example ($d = 3$):

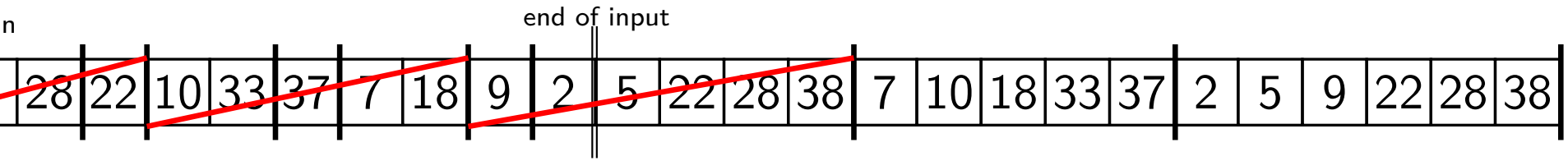


- Merge the first 3 runs with *3-way-merge*.
- Merge the next 3 runs.

d-way mergesort

- Keep a list L of streams, each stream stores a run
- Initialize L with input split into runs ($\rightsquigarrow$ details later)
- Repeatedly merge the next d runs until only one left.

Example ($d = 3$):

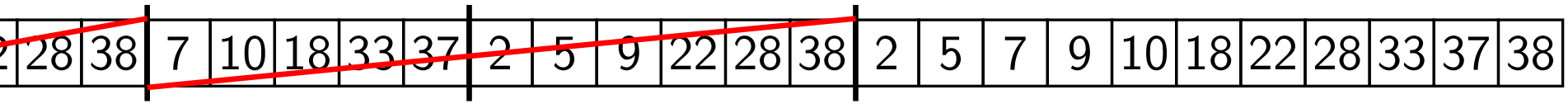


- Merge the first 3 runs with *3-way-merge*.
- Merge the next 3 runs.
- Merge the next 3 runs. End of round 1.
  We include runs from the next round if there is room.

d-way mergesort

- Keep a list L of streams, each stream stores a run
- Initialize L with input split into runs ($\rightsquigarrow$ details later)
- Repeatedly merge the next d runs until only one left.

Example ($d = 3$):



- Merge the first 3 runs with *3-way-merge*.
- Merge the next 3 runs.
- Merge the next 3 runs. End of round 1.
  We include runs from the next round if there is room.
- With one more merge we are done.

d-way mergesort code

```
d-way-mergesort(S)
1.  L ← initialize empty list of streams
2.  while S is not empty do L.append(extract-sorted-run(S))

3.  while L.size ≥ 2 do
4.      for i = 1, ..., d do S_i ← L.delete-front()
5.          // S_i should be an empty stream if L empty
6.      d-way-merge(S_1, ..., S_d, S)
7.      L ← append(S)
8.  // The single stream left on L is now sorted
```

d-way mergesort code

```
d-way-mergesort(S)
1.  L ← initialize empty list of streams
2.  while S is not empty do L.append(extract-sorted-run(S))

3.  while L.size ≥ 2 do
4.      for i = 1, ..., d do S_i ← L.delete-front()
5.          // S_i should be an empty stream if L empty
6.      d-way-merge(S_1, ..., S_d, S)
7.      L ← append(S)
8.  // The single stream left on L is now sorted
```

Subroutine *extract-sorted-run* extracts a run from stream S:

- In RAM model: Take existing maximum runs in input.

d-way mergesort code

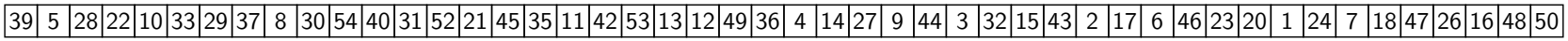
```
d-way-mergesort(S)
1.  L ← initialize empty list of streams
2.  while S is not empty do L.append(extract-sorted-run(S))

3.  while L.size ≥ 2 do
4.      for i = 1, ..., d do S_i ← L.delete-front()
5.          // S_i should be an empty stream if L empty
6.      d-way-merge(S_1, ..., S_d, S)
7.      L ← append(S)
8.  // The single stream left on L is now sorted
```

Subroutine *extract-sorted-run* extracts a run from stream S:

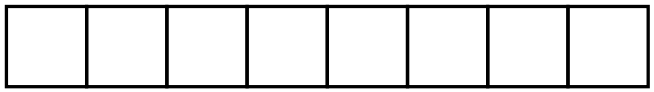
- In RAM model: Take existing maximum runs in input.
- In EMM model: We can *guarantee* that the run has length $\geq M$.
  - Take M numbers from S and sort them in internal memory.
  - These can all be in one run. (And we can perhaps extend it even more.)

d-way mergesort in external memory model

External ($B = 2$):



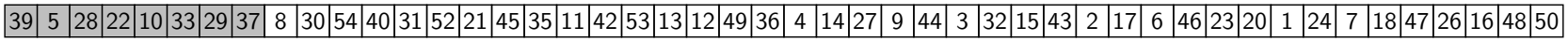
Internal ($M = 8$):



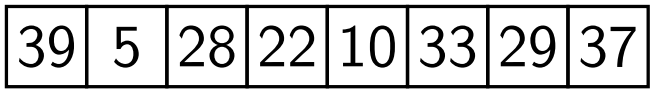
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$.

d-way mergesort in external memory model

External ($B = 2$):



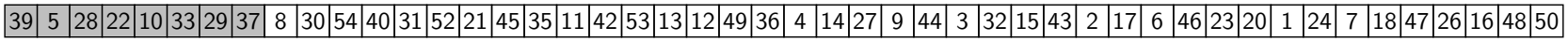
Internal ($M = 8$):



1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$.

d-way mergesort in external memory model

External ($B = 2$):



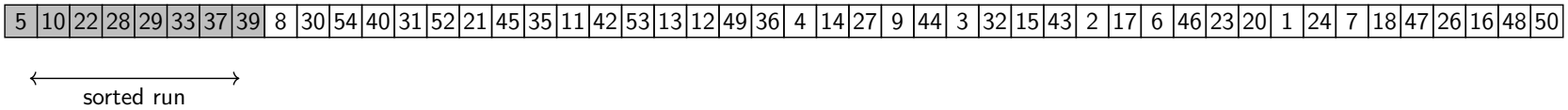
Internal ($M = 8$):

5	10	22	28	29	33	37	39

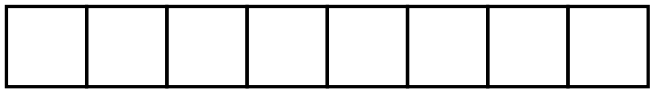
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$.

d-way mergesort in external memory model

External ($B = 2$):



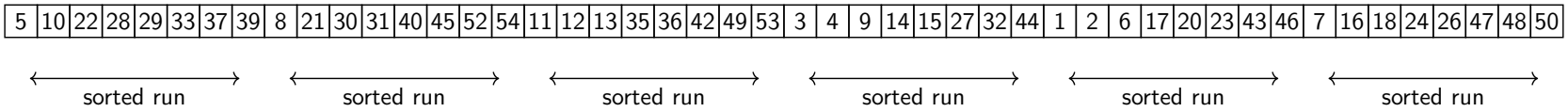
Internal ($M = 8$):



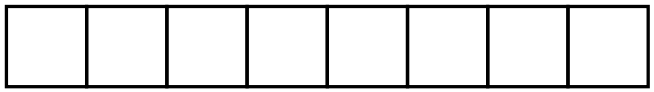
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$.

d-way mergesort in external memory model

External ($B = 2$):



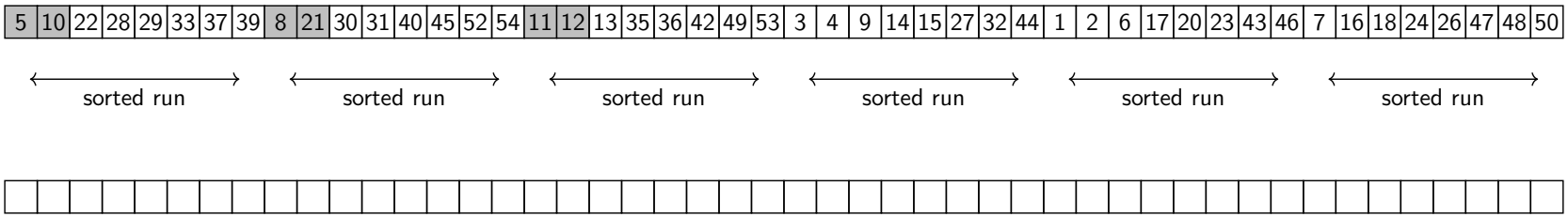
Internal ($M = 8$):



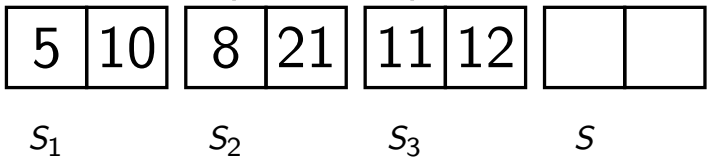
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

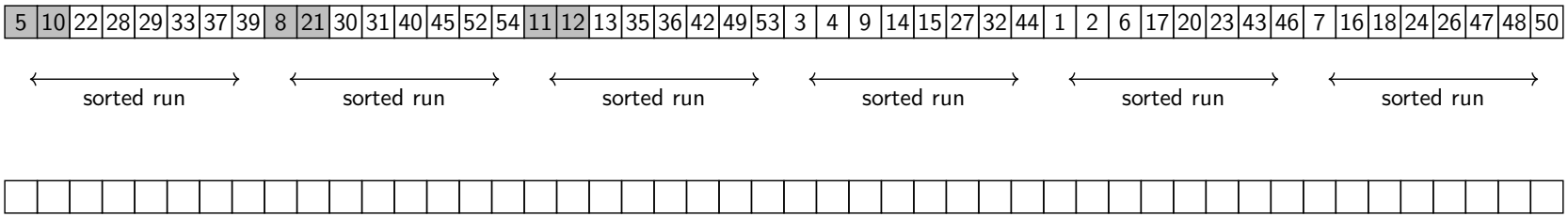


(priority queue not shown)

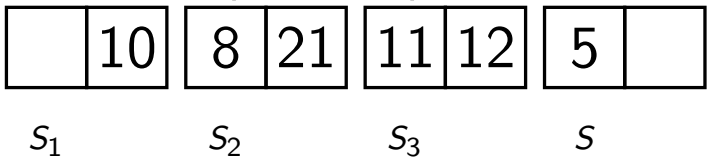
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

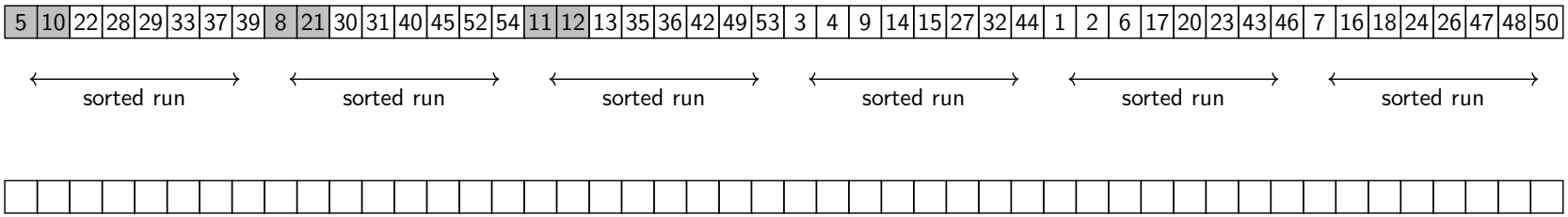


(priority queue not shown)

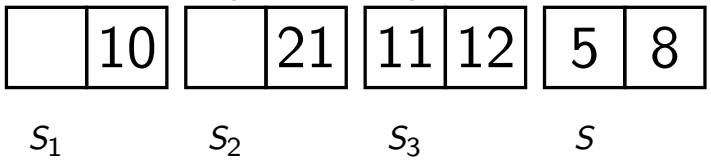
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

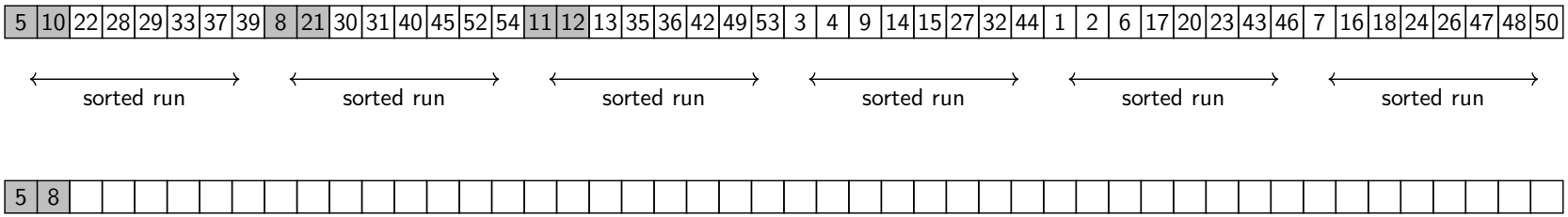


(priority queue not shown)

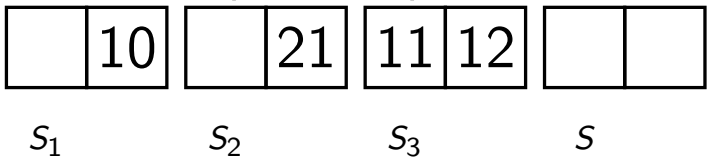
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

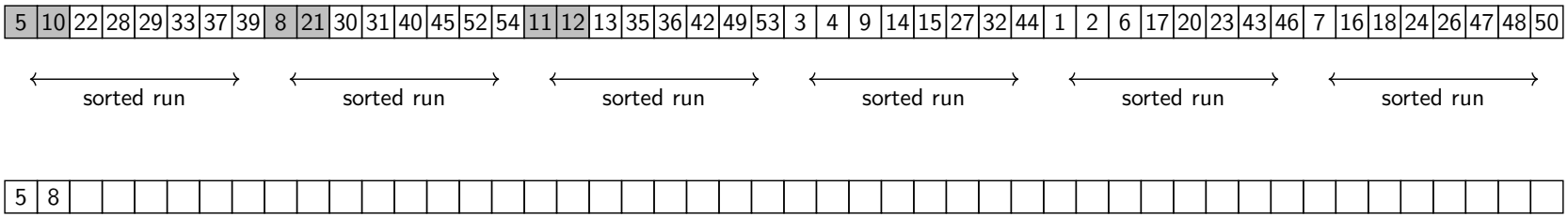


(priority queue not shown)

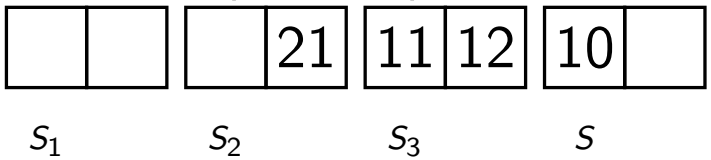
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

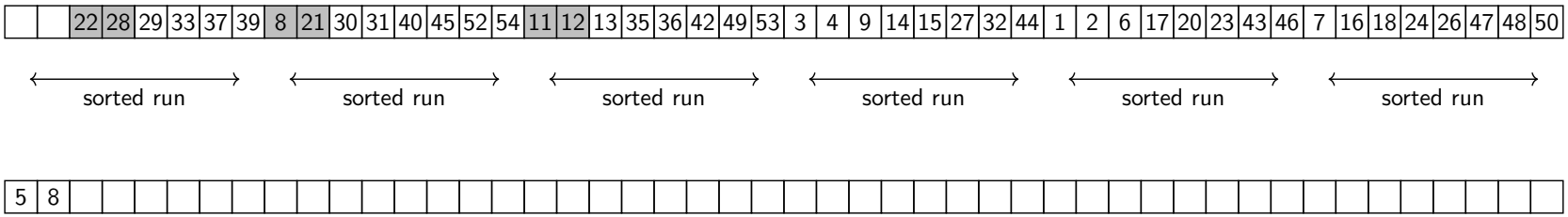


(priority queue not shown)

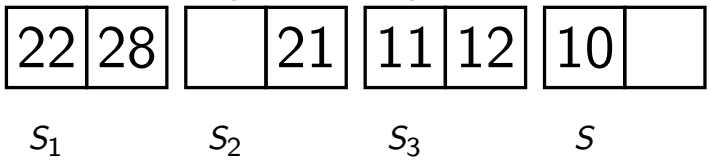
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

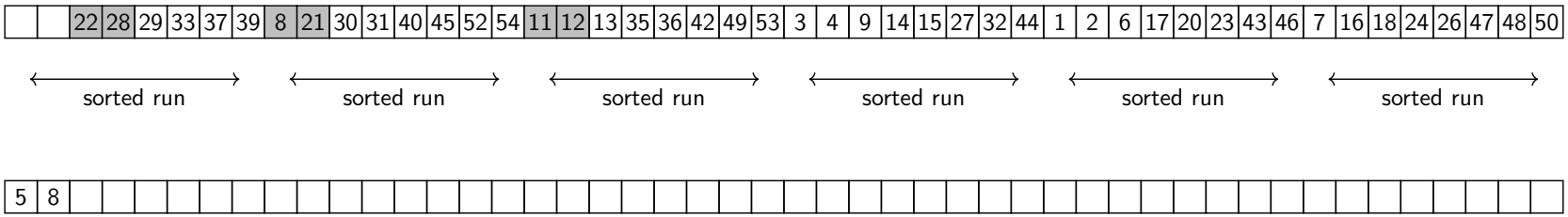


(priority queue not shown)

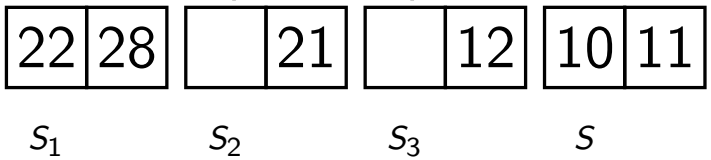
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

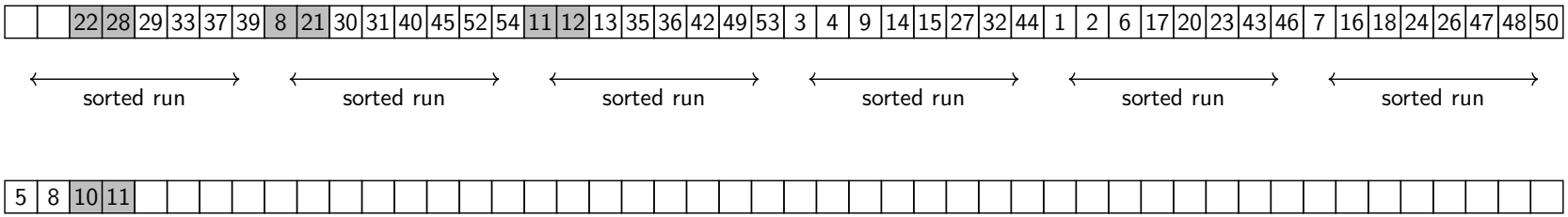


(priority queue not shown)

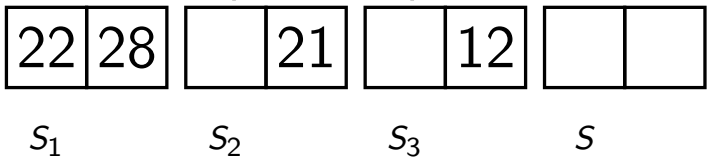
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

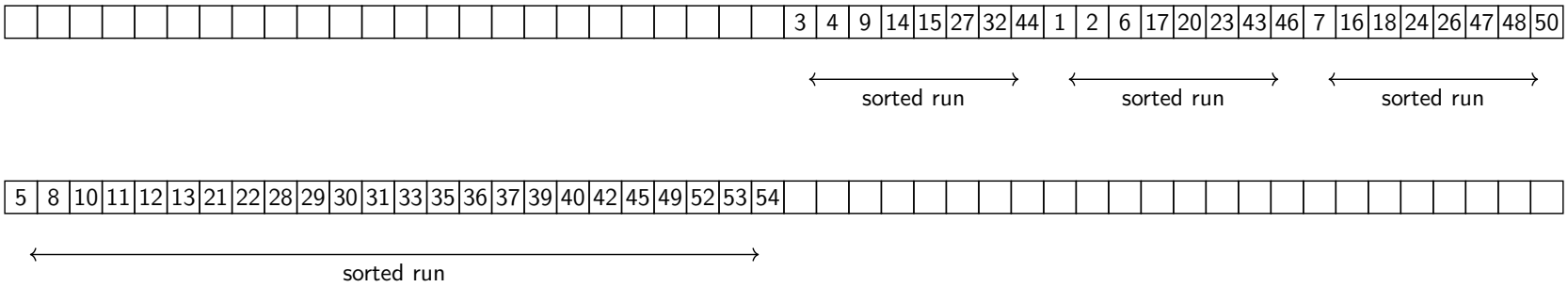


(priority queue not shown)

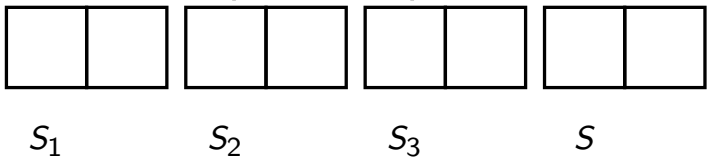
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

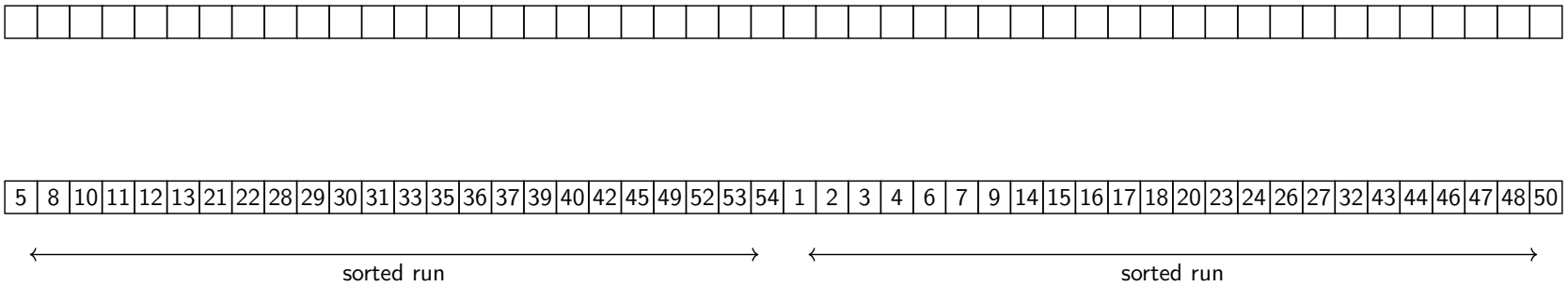


(priority queue not shown)

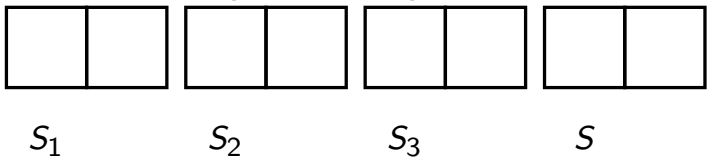
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):

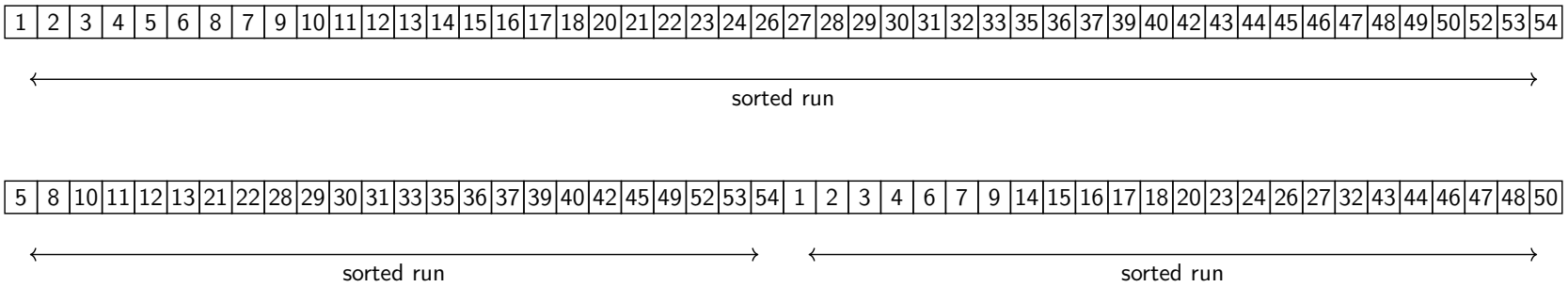


(priority queue not shown)

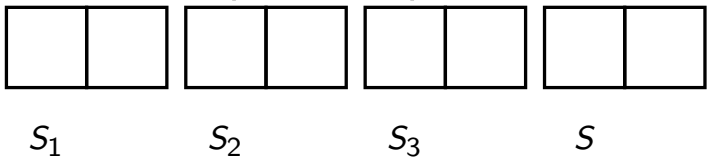
1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.
3. One round of merging uses $\Theta(\frac{n}{B})$ **block transfers**
   and reduces number of runs by a factor of d

d-way mergesort in external memory model

External ($B = 2$):



Internal ($M = 8$):



(priority queue not shown)

1. Create $\leq \frac{n}{M}$ sorted runs of length $\geq M$. $\Theta(\frac{n}{B})$ **block transfers**
2. Merge the first d runs using *d-way merge*.
   Use the largest d that will fit with internal memory; $d \in \Theta(\frac{M}{B})$.
3. One round of merging uses $\Theta(\frac{n}{B})$ **block transfers**
   and reduces number of runs by a factor of d
4. Keep doing rounds until only one run is left

d-way merge-sort

- We have $\leq \log_d(\frac{n}{M})$ **rounds** of merging:
  - $\leq \frac{n}{M}$ runs after initialization
  - $\leq \frac{n}{M}/d$ runs after one round.
  - $\leq \frac{n}{M}/d^k$ runs after k rounds $\Rightarrow k \leq \log_d(\frac{n}{M})$.

d-way merge-sort

- We have $\leq \log_d(\frac{n}{M})$ **rounds** of merging:
  - $\leq \frac{n}{M}$ runs after initialization
  - $\leq \frac{n}{M}/d$ runs after one round.
  - $\leq \frac{n}{M}/d^k$ runs after k rounds $\Rightarrow k \leq \log_d(\frac{n}{M})$.
- We have $O(\frac{n}{B})$ block-transfers per round and $d \in \Theta(\frac{M}{B})$.

$\Rightarrow$ Total # block transfers is proportional to

$$\log_d(\tfrac{n}{M}) \cdot \tfrac{n}{B} \in O\big(\log_{M/B}(\tfrac{n}{M}) \cdot \tfrac{n}{B}\big)$$

d-way merge-sort

- We have $\leq \log_d(\frac{n}{M})$ **rounds** of merging:
  - $\leq \frac{n}{M}$ runs after initialization
  - $\leq \frac{n}{M}/d$ runs after one round.
  - $\leq \frac{n}{M}/d^k$ runs after k rounds $\Rightarrow k \leq \log_d(\frac{n}{M})$.
- We have $O(\frac{n}{B})$ block-transfers per round and $d \in \Theta(\frac{M}{B})$.

$\Rightarrow$ Total # block transfers is proportional to

$$\log_d(\tfrac{n}{M}) \cdot \tfrac{n}{B} \in O\big(\log_{M/B}(\tfrac{n}{M}) \cdot \tfrac{n}{B}\big)$$

One can prove lower bounds in the external memory model:

*We **require** $\Omega\big(\log_{M/B}(\frac{n}{M}) \cdot \frac{n}{B}\big)$ block transfers in any comparison-based sorting algorithm.*

(The proof is beyond the scope of the course.)

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

Outline

Dictionaries in external memory

Recall: Dictionaries store n KVPs and support *search*, *insert* and *delete*.

- **Recall:** AVL-trees were optimal in time and space in RAM model
- $\Theta(\log n)$ run-time $\Rightarrow$ $O(\log n)$ block transfers per operation
- But: Inserts happen at varying locations of the tree.
  $\rightsquigarrow$ nearby nodes are unlikely to be on the same block
  $\rightsquigarrow$ typically $\Theta(\log n)$ block transfers per operation

Dictionaries in external memory

Recall: Dictionaries store n KVPs and support *search*, *insert* and *delete*.

- **Recall:** AVL-trees were optimal in time and space in RAM model
- $\Theta(\log n)$ run-time $\Rightarrow$ $O(\log n)$ block transfers per operation
- But: Inserts happen at varying locations of the tree.
  $\rightsquigarrow$ nearby nodes are unlikely to be on the same block
  $\rightsquigarrow$ typically $\Theta(\log n)$ block transfers per operation
- We would like to have *fewer* block transfers.
  - Goal: $O(\log_B n)$ block transfers.
  - Does this really make a difference?
  - Consider 'typical' values: $n \approx 2^{50}, B \approx 2^{15}$.
    What is $\log n$ vs. $\log_B n$?

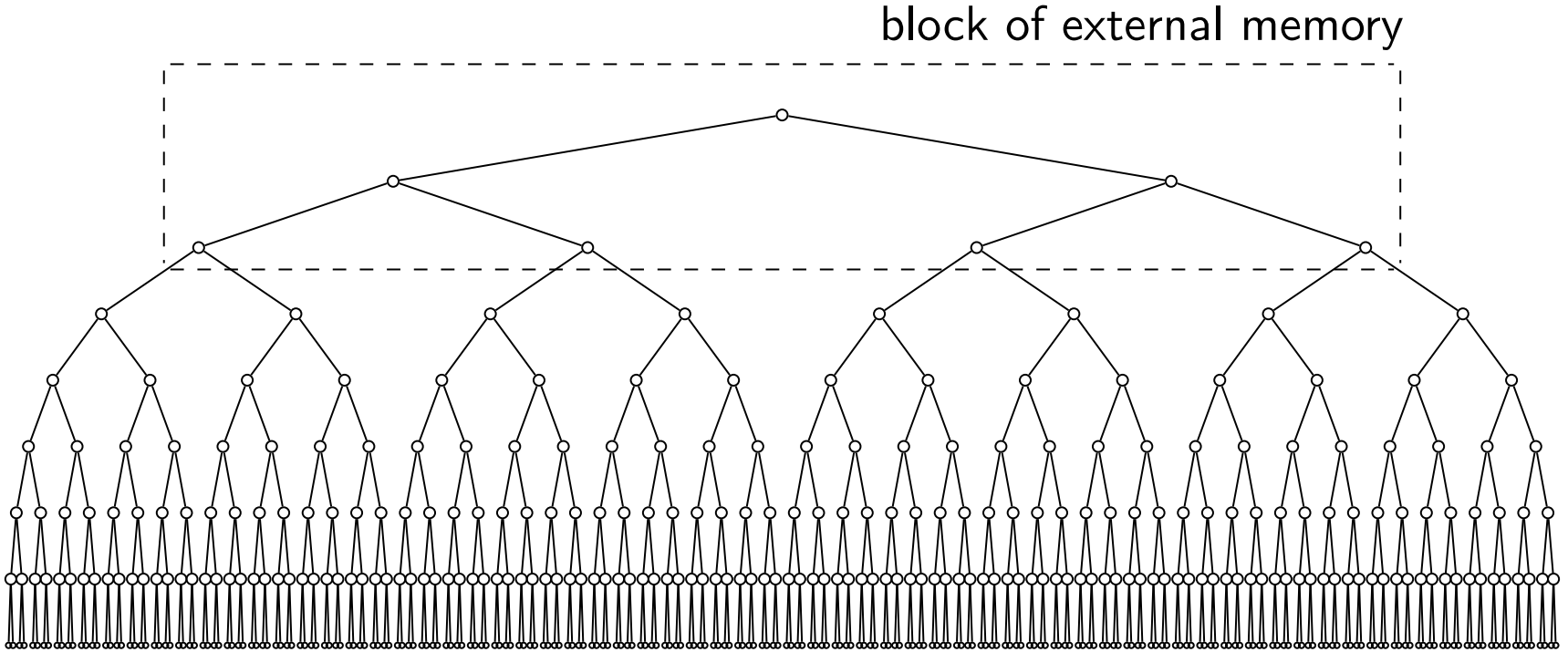
Dictionaries in external memory

Recall: Dictionaries store n KVPs and support *search*, *insert* and *delete*.

- **Recall:** AVL-trees were optimal in time and space in RAM model
- $\Theta(\log n)$ run-time $\Rightarrow$ $O(\log n)$ block transfers per operation
- But: Inserts happen at varying locations of the tree.
  $\rightsquigarrow$ nearby nodes are unlikely to be on the same block
  $\rightsquigarrow$ typically $\Theta(\log n)$ block transfers per operation
- We would like to have *fewer* block transfers.
  - Goal: $O(\log_B n)$ block transfers.
  - Does this really make a difference?
  - Consider 'typical' values: $n \approx 2^{50}, B \approx 2^{15}$.
    What is $\log n$ vs. $\log_B n$?

Better solution: design a tree-structure that *guarantees* that many nodes on search-paths are within one block.

Idealized structure



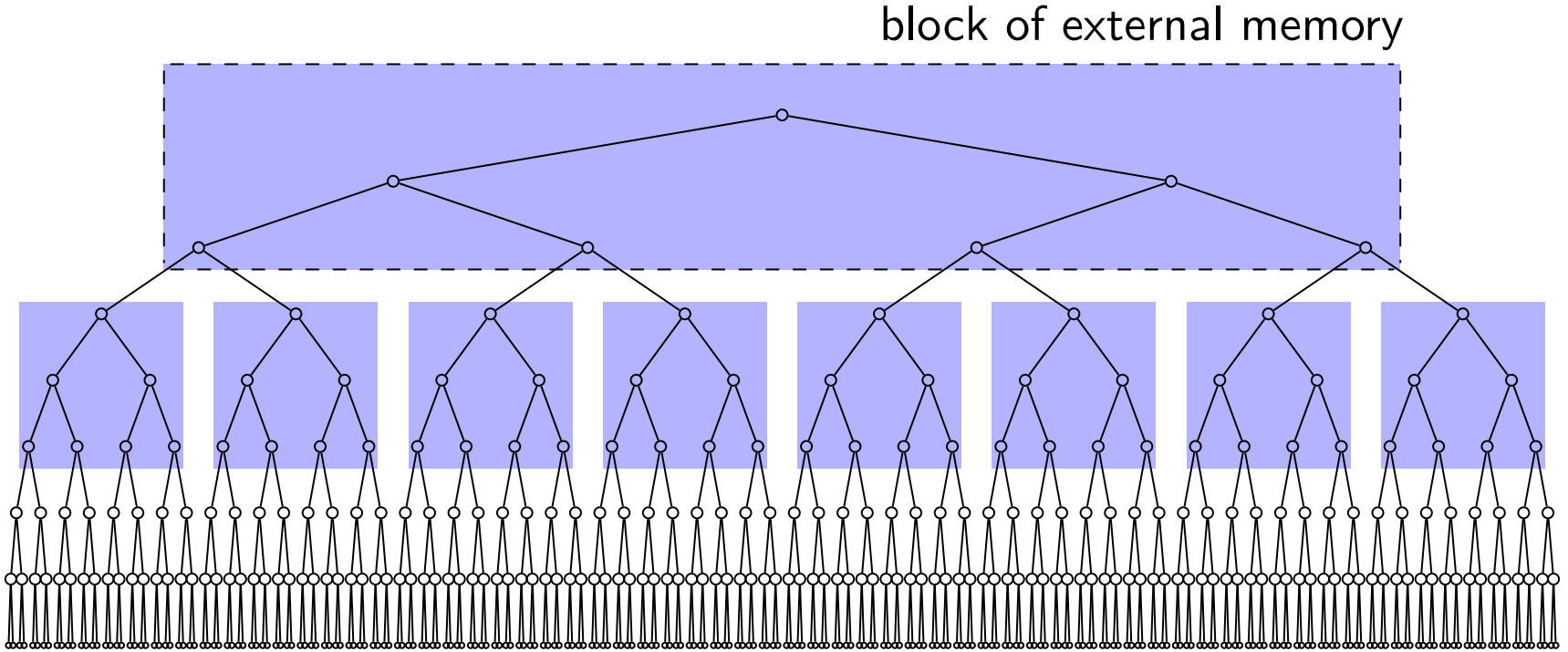
Idea: Store complete subtrees with $\log b$ levels in one block of memory.
($b \in \Theta(B)$ is maximal so that these fit into one block.)

- Each block/subtree then covers height $\log b$

$\Rightarrow$ Search-path hits $\frac{\log n}{\log b}$ blocks $\Rightarrow \log_b n$ block-transfers

- Since $b \in \Theta(B)$, we have $\log_b n \in \Theta(\log_B n)$ (why?)

Idealized structure



Idea: Store complete subtrees with $\log b$ levels in one block of memory.
($b \in \Theta(B)$ is maximal so that these fit into one block.)

- Each block/subtree then covers height $\log b$

$\Rightarrow$ Search-path hits $\frac{\log n}{\log b}$ blocks $\Rightarrow \log_b n$ block-transfers

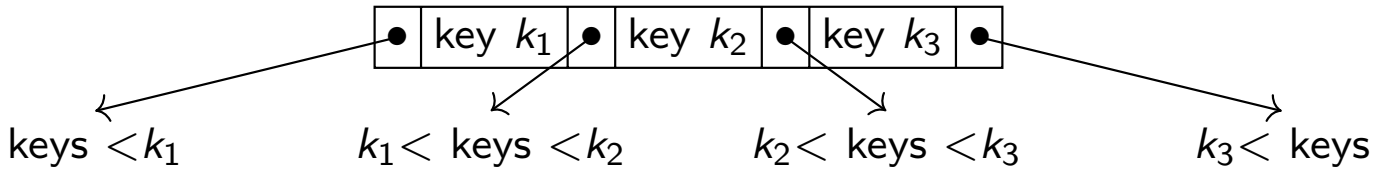
- Since $b \in \Theta(B)$, we have $\log_b n \in \Theta(\log_B n)$ (why?)

Idea: View the entire content of a block as one node.

Towards a-b-trees

Define *multiway-tree*: A node can store multiple keys.

Definition: A d-**node** stores d keys, has $d+1$ subtrees, and stored keys are between the keys in the subtrees.

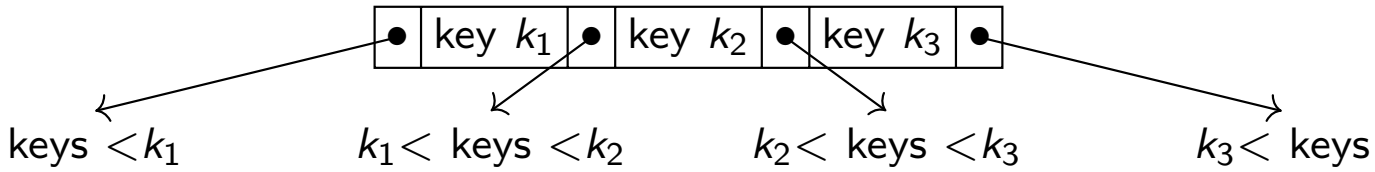


We *always* have one more subtree than keys (but subtrees may be empty).

Towards *a*-*b*-trees

Define *multiway-tree*: A node can store multiple keys.

Definition: A ***d*-node** stores d keys, has $d+1$ subtrees, and stored keys are between the keys in the subtrees.



We *always* have one more subtree than keys (but subtrees may be empty).

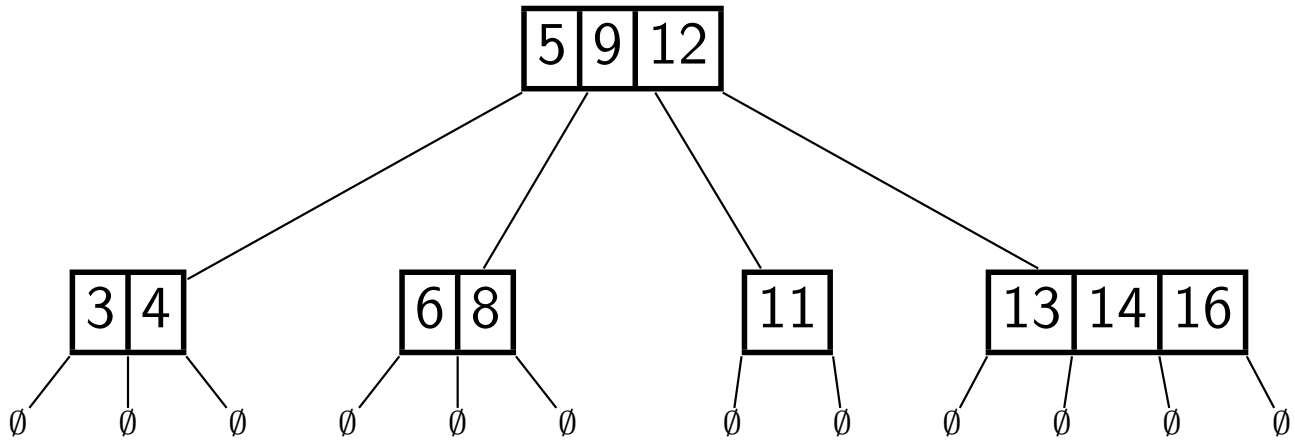
- To allow *insert*/*delete*, we permit a varying numbers of keys in nodes (within limits)
- We also rigidly restrict where empty subtrees may be.
- This gives much smaller height than for AVL-trees
  $\Rightarrow$ fewer block transfers

a-b-trees

Definition: An **a-b-tree** (for some $b \geq 3$ and $2 \leq a \leq \lceil \frac{b}{2} \rceil$) satisfies

1. Every non-root is a d-node for some $a-1 \leq d \leq b-1$.
   - Between a and b subtrees, between $a-1$ and $b-1$ keys.
2. The root is a d-node for $1 \leq d \leq b-1$.
   - Between 2 and b subtrees, between 1 and $b-1$ keys.
3. All empty subtrees are at the same level.

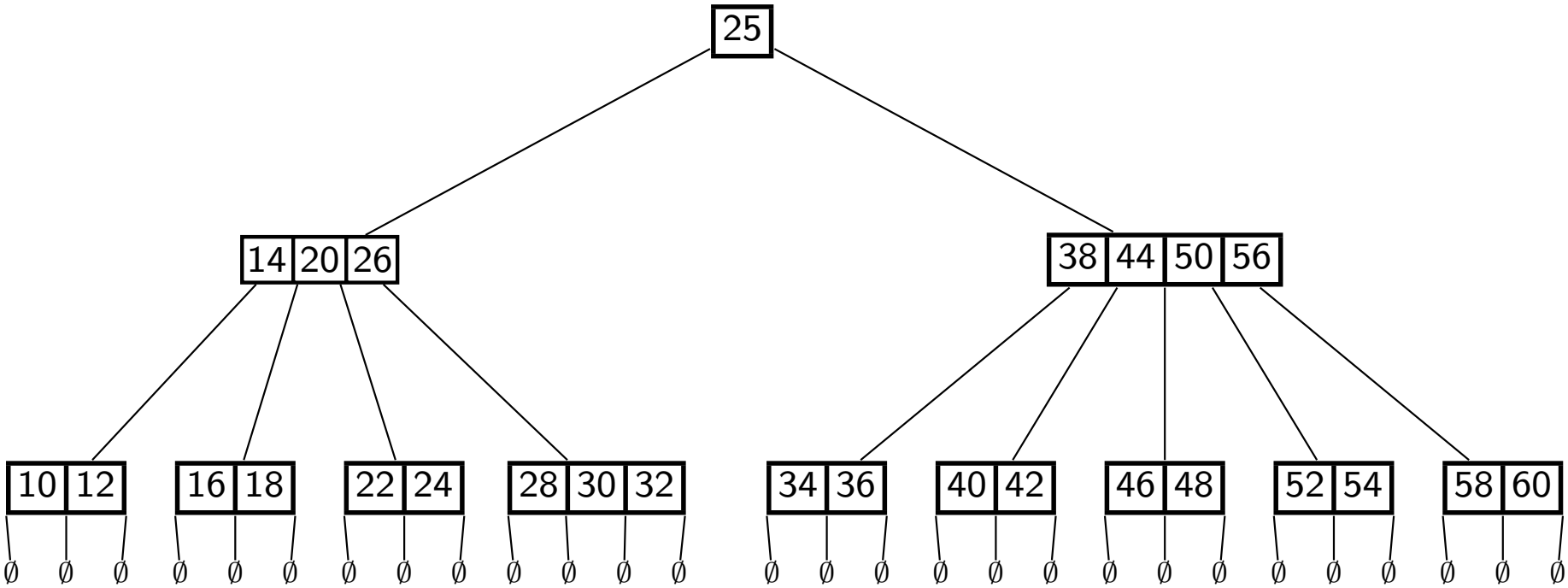
Example: A 2-4-tree of height 1.



For 2-4-trees, every node has between 1 and 3 keys.

a-b-tree Example

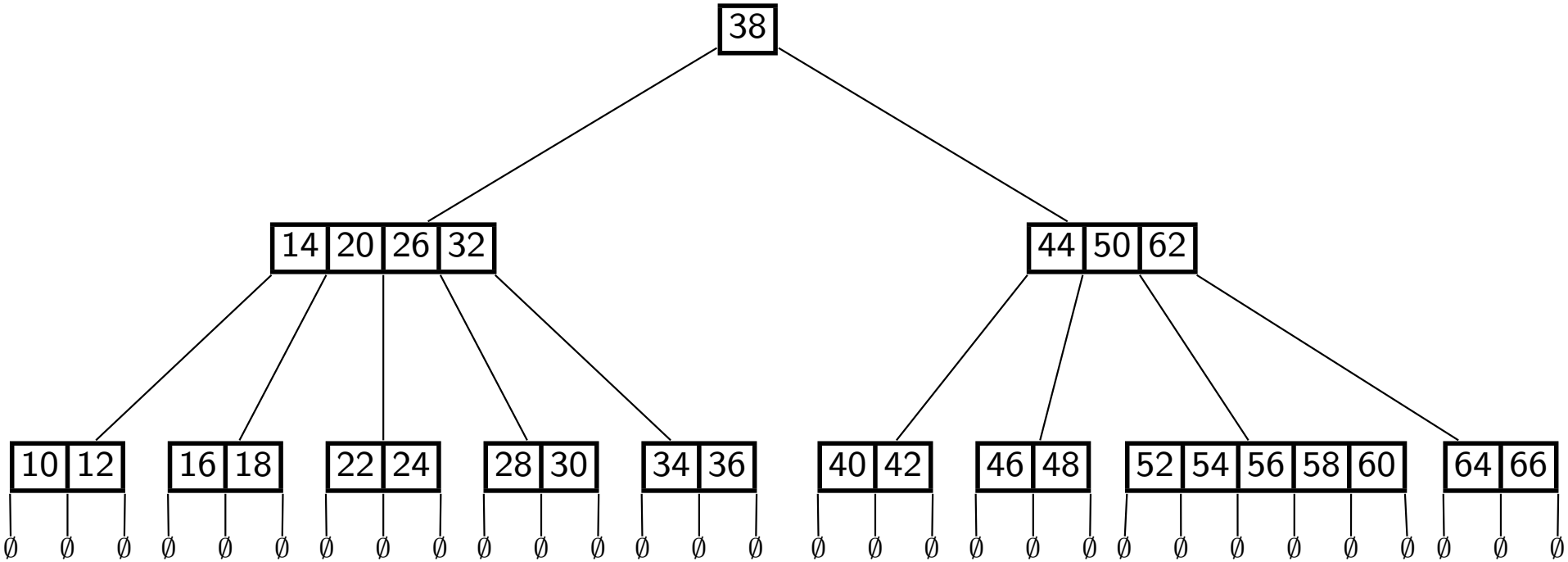
Example: A 3-5-tree of height 2.



Typically we will specify the **order** b and then set $a = \lceil \frac{b}{2} \rceil$.

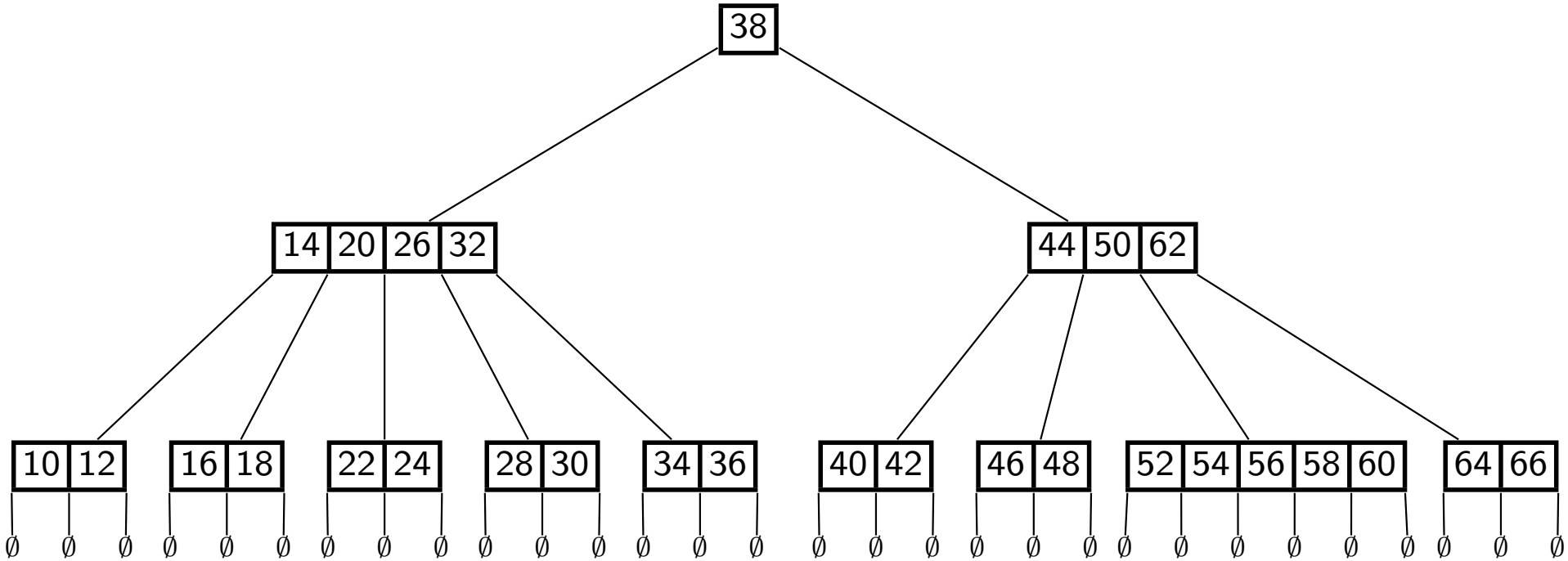
a-b-tree Example

Example: A 3-6-tree of height 2.



a-b-tree Example

Example: A 3-6-tree of height 2.



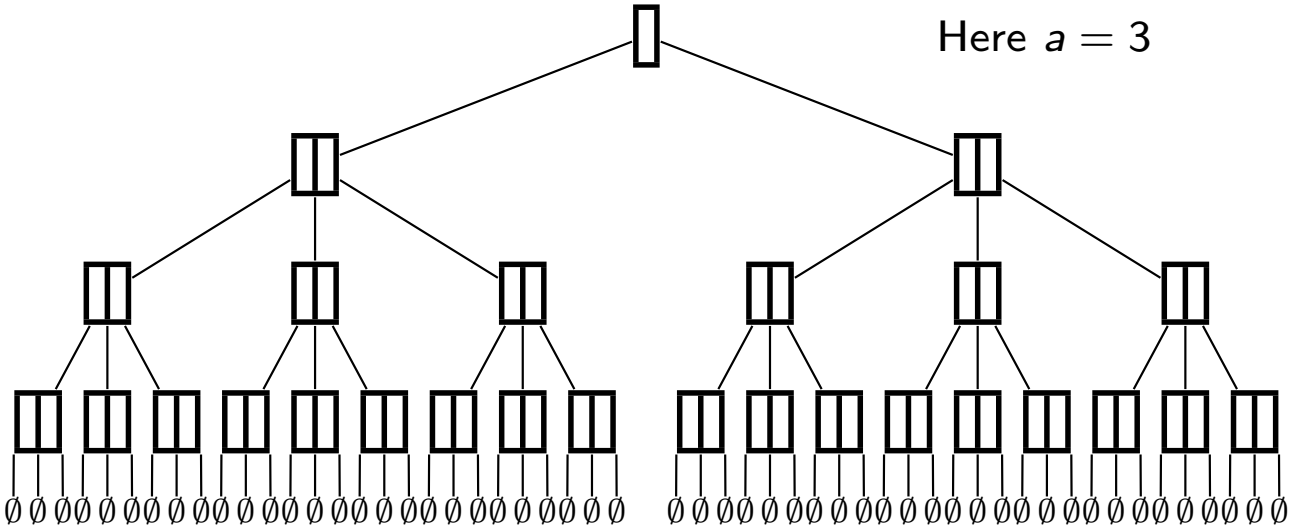
Note: With small height we can store *many* keys.
A 3-6-tree of height 2 can store up to $(1 + 6 + 36) \cdot 5 = 215$ keys.

a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$

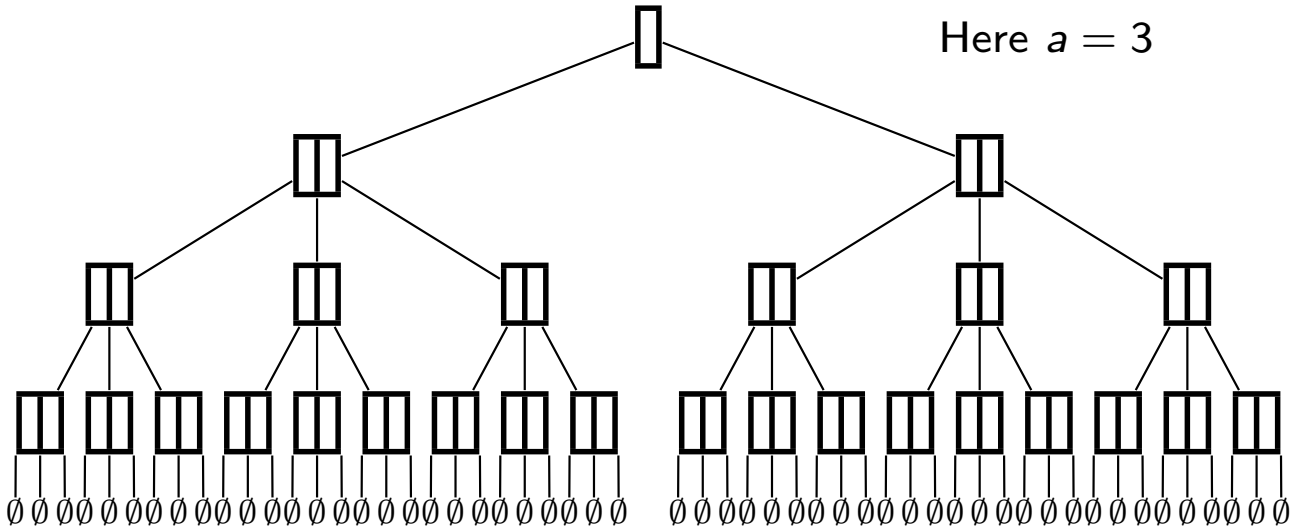


a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$



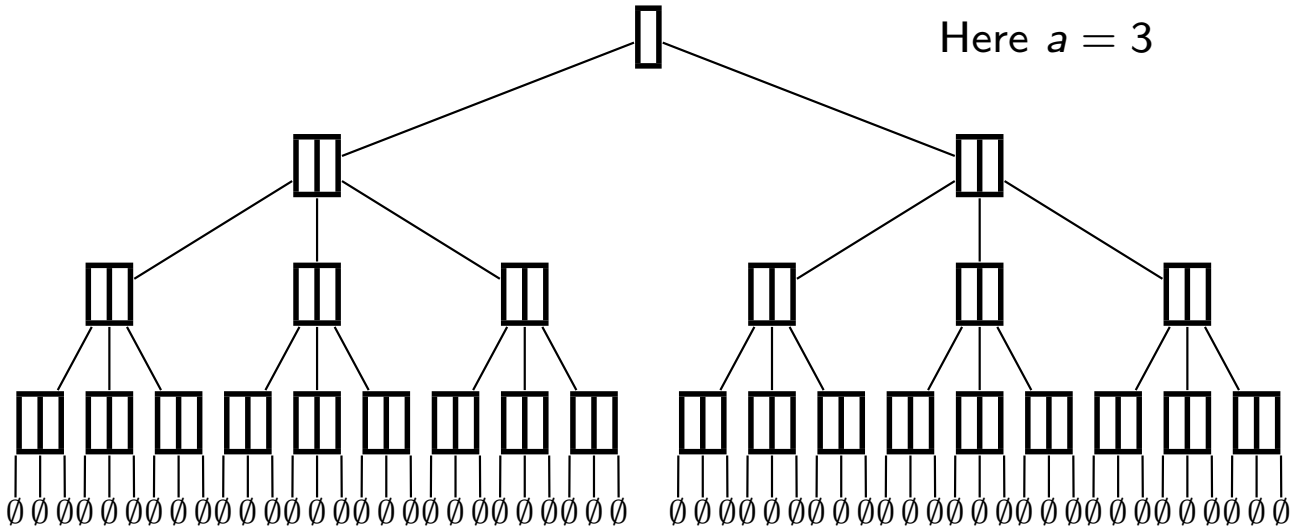
$$\#\text{ non-root nodes} \geq \sum_{i=1}^{h} 2a^{i-1} = 2\sum_{j=0}^{h-1} a^j = 2\frac{a^h - 1}{a - 1}$$

a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$



$$\#\text{ non-root nodes} \geq \sum_{i=1}^{h} 2a^{i-1} = 2\sum_{j=0}^{h-1} a^j = 2\frac{a^h - 1}{a - 1}$$

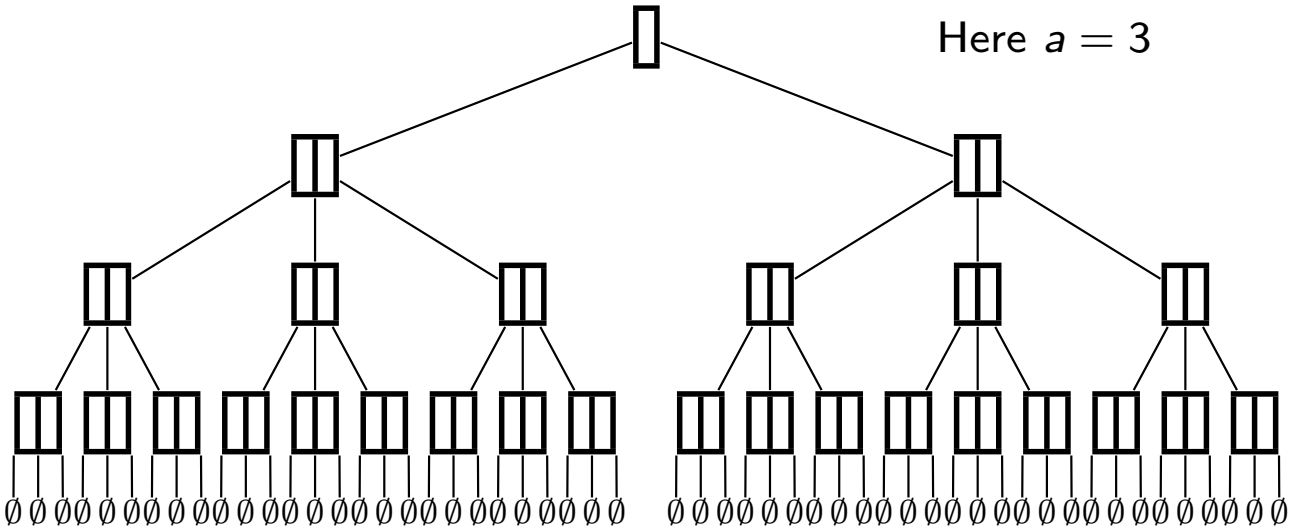
$$n = \#\text{ KVPs} \geq \underbrace{1}_{\text{root}} + \underbrace{(a-1)}_{\geq a-1 \text{ KVPs at non-root}} \quad 2\frac{a^h - 1}{a - 1} = 2a^h - 1$$

a-b-tree Height

Theorem: An a-b-tree with n keys has $O(\log_a(n))$ height.

Proof: How many keys *must* an a-b-tree of height h have?

Level	Nodes
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
$\vdots$	$\vdots$
h	$\geq 2a^{h-1}$



$$\#\text{ non-root nodes} \geq \sum_{i=1}^{h} 2a^{i-1} = 2\sum_{j=0}^{h-1} a^j = 2\frac{a^h - 1}{a - 1}$$

$$n = \#\text{ KVPs} \geq \underbrace{1}_{\text{root}} + \underbrace{(a-1)}_{\geq a-1 \text{ KVPs at non-root}} 2\frac{a^h - 1}{a - 1} = 2a^h - 1$$

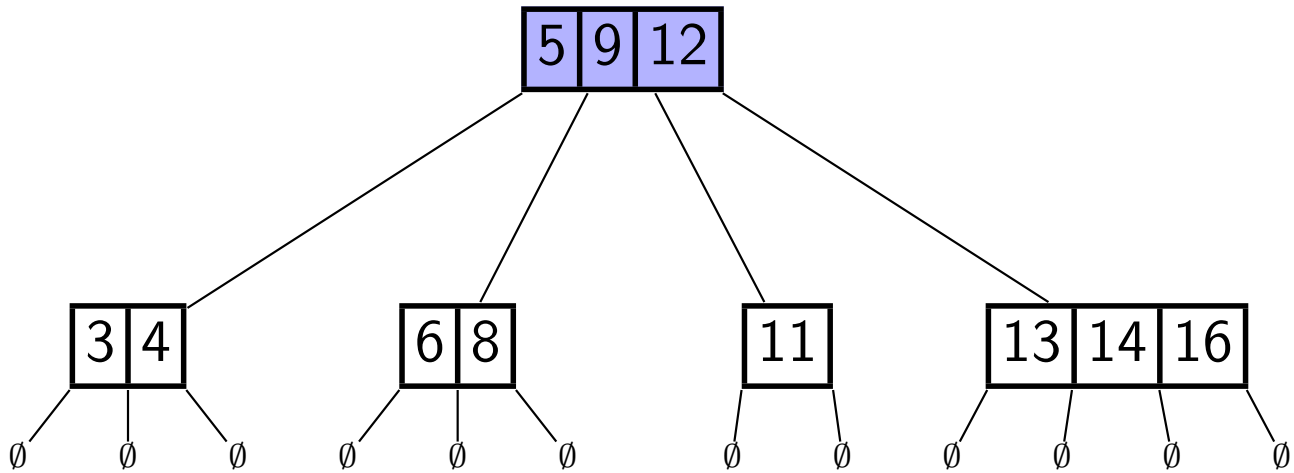
Therefore $h \leq \log_a\left(\frac{n+1}{2}\right)$.

a-b-tree Operations

Search is similar to BST:

- Compare search-key to keys at node
- If not found, continue in appropriate subtree until empty

Example: *search*(15)

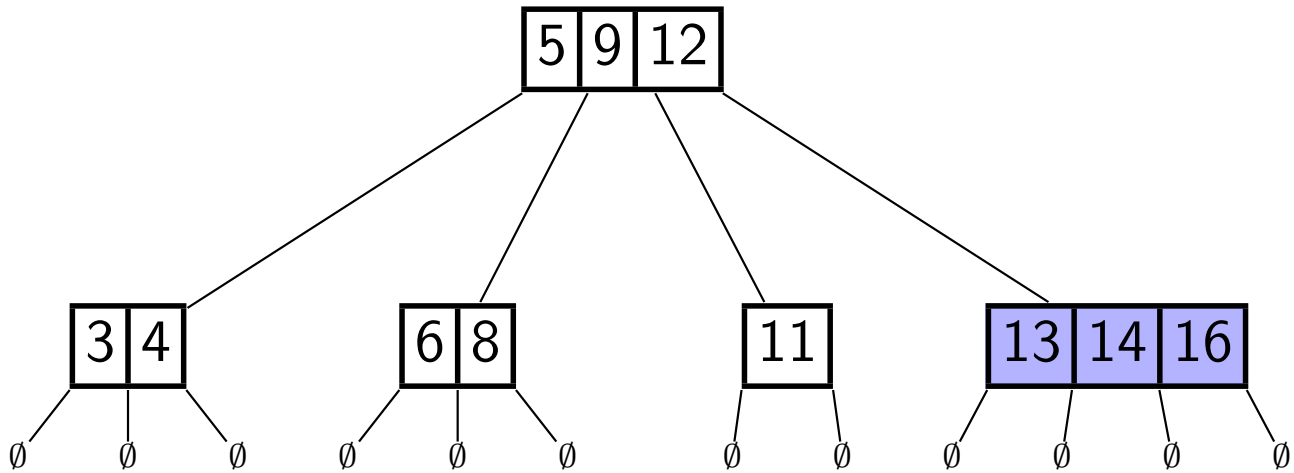


a-*b*-tree Operations

Search is similar to BST:

- Compare search-key to keys at node
- If not found, continue in appropriate subtree until empty

Example: *search*(15)

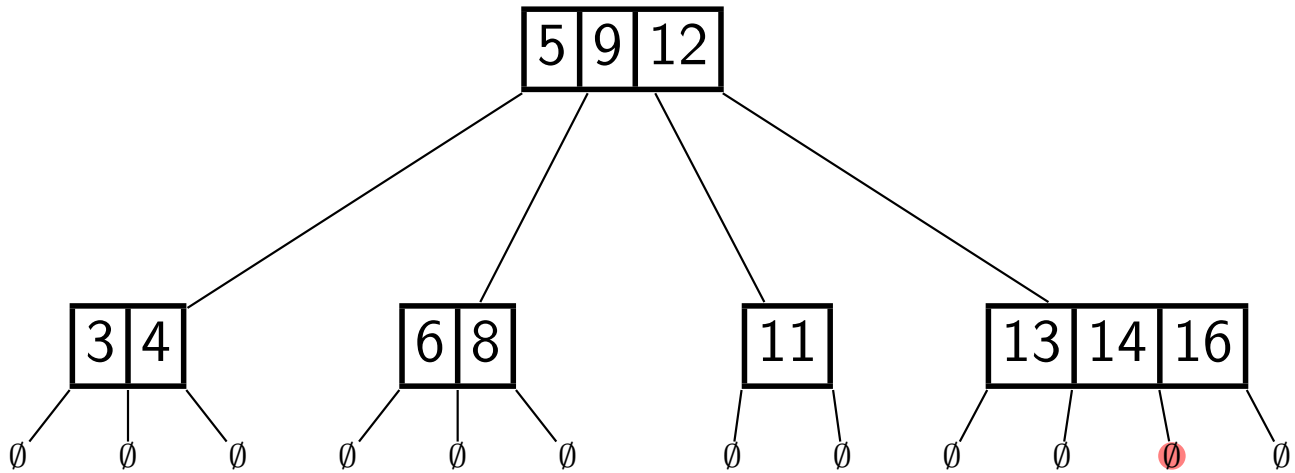


a-*b*-tree Operations

Search is similar to BST:

- Compare search-key to keys at node
- If not found, continue in appropriate subtree until empty

Example: *search*(15) *not found*



a-*b*-tree *search*

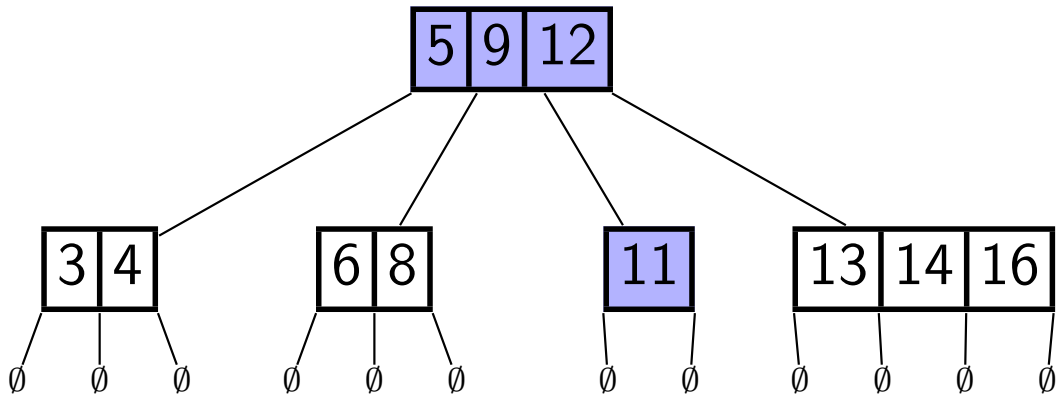
```
abTree::search(k)
1.  z ← root, p ← NULL                    // p: parent of z
2.  while z is not NULL
3.      let ⟨T_0, k_1, ..., k_d, T_d⟩ be key-subtree list at z
4.      if k ≥ k_1
5.          i ← maximal index such that k_i ≤ k
6.          if k_i = k then return KVP at k_i
7.          else p ← z, z ← root of T_i
8.      else p ← z, z ← root of T_0
9.  return "not found, would be in p"
```

- # visited nodes: $O(\log_a n)$ (one per level)
- Note: Finding i is not constant time (depending on b)

a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.

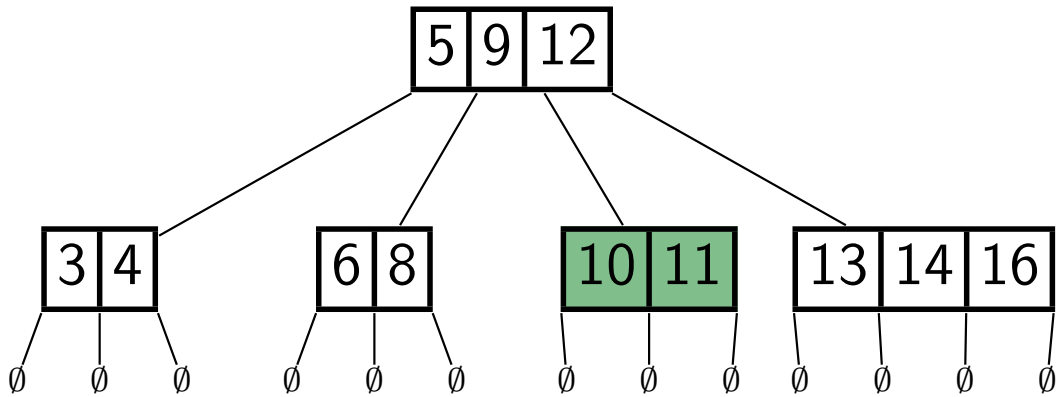
Example (2-4-tree): *insert*(10)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.

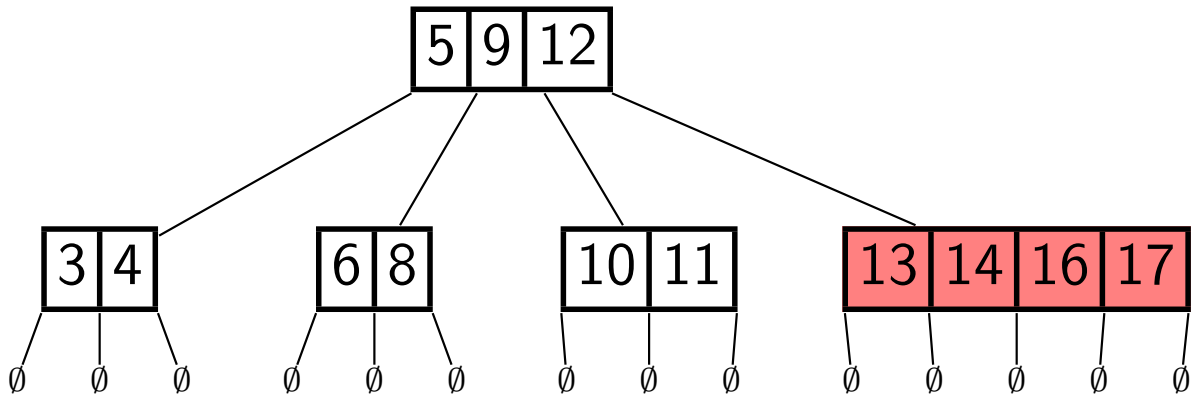
Example (2-4-tree): *insert*(10)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

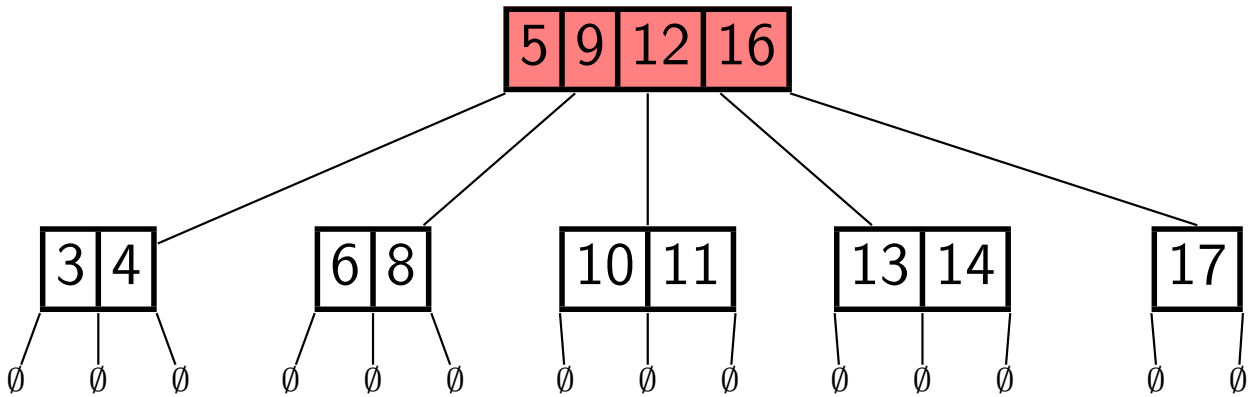
Example (2-4-tree): *insert*(17)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

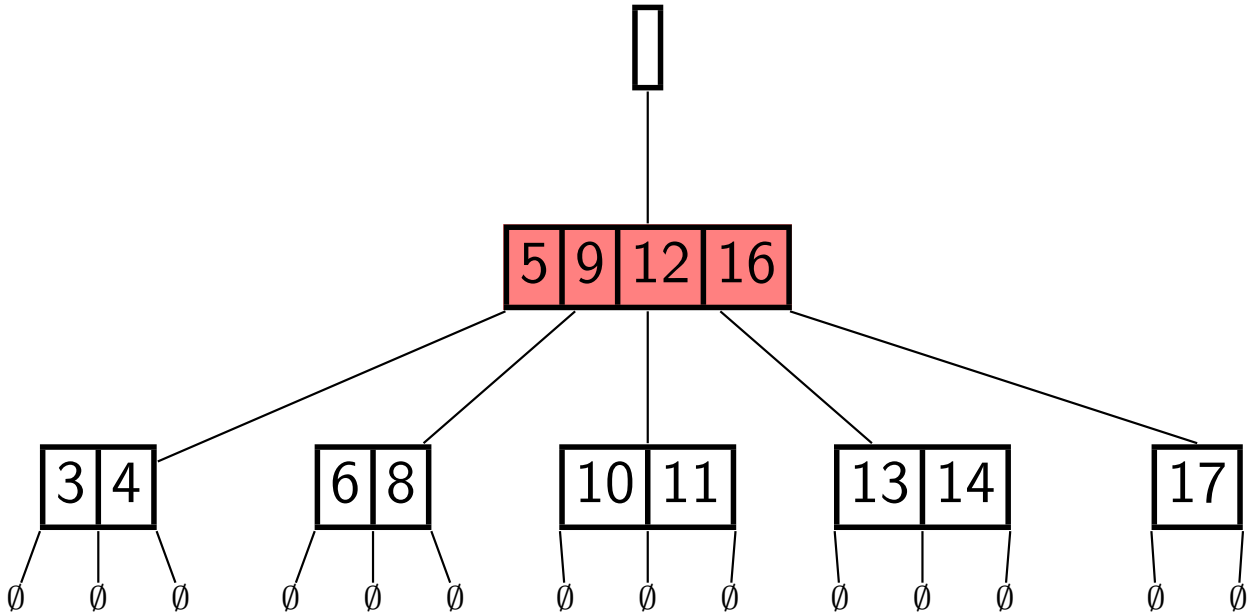
Example (2-4-tree): *insert*(17)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

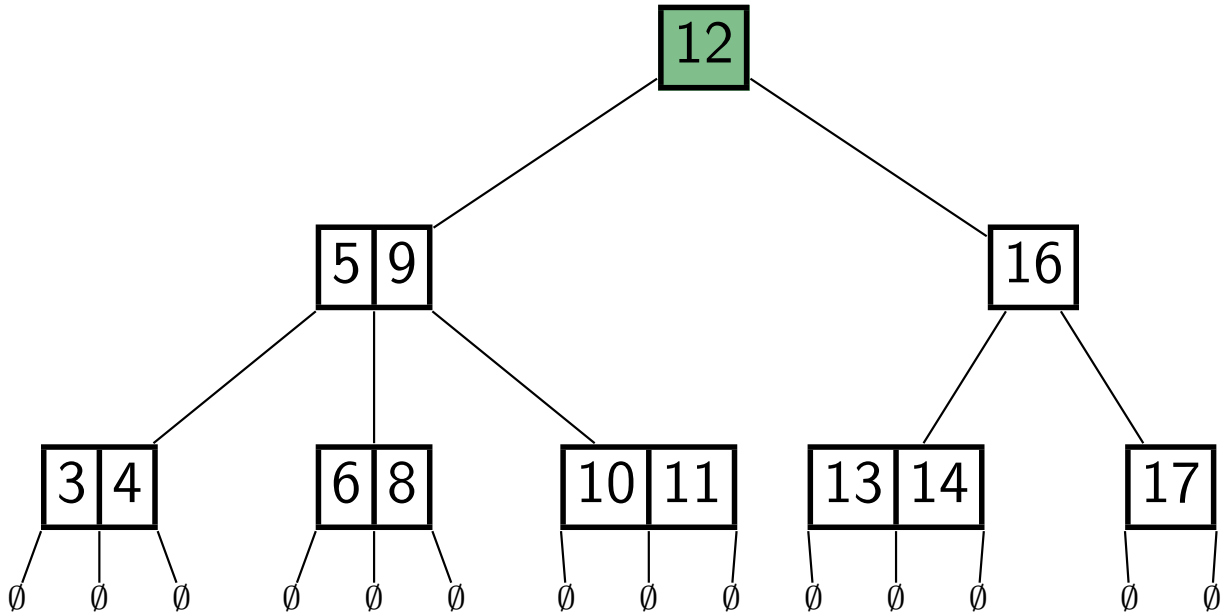
Example (2-4-tree): *insert*(17)



a-b tree *insert*

- Do *abTree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.

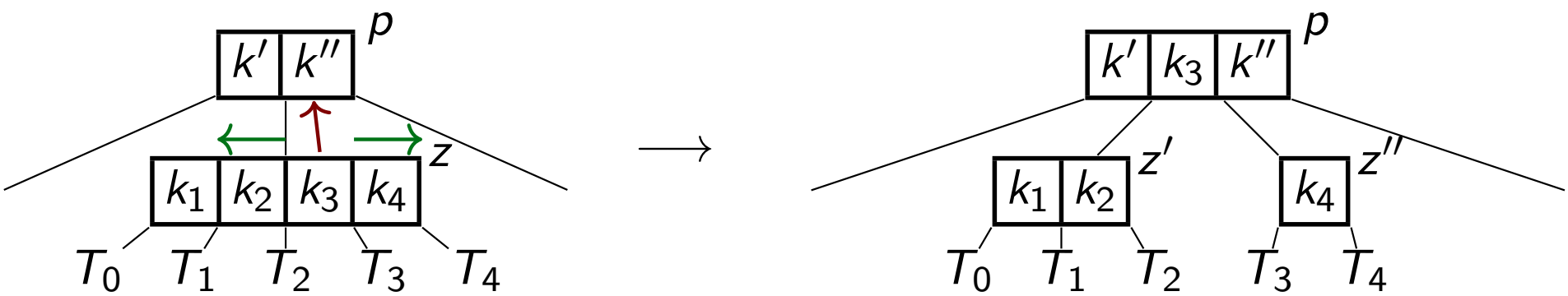
Example (2-4-tree): *insert*(17)



a-*b*-tree *insert*

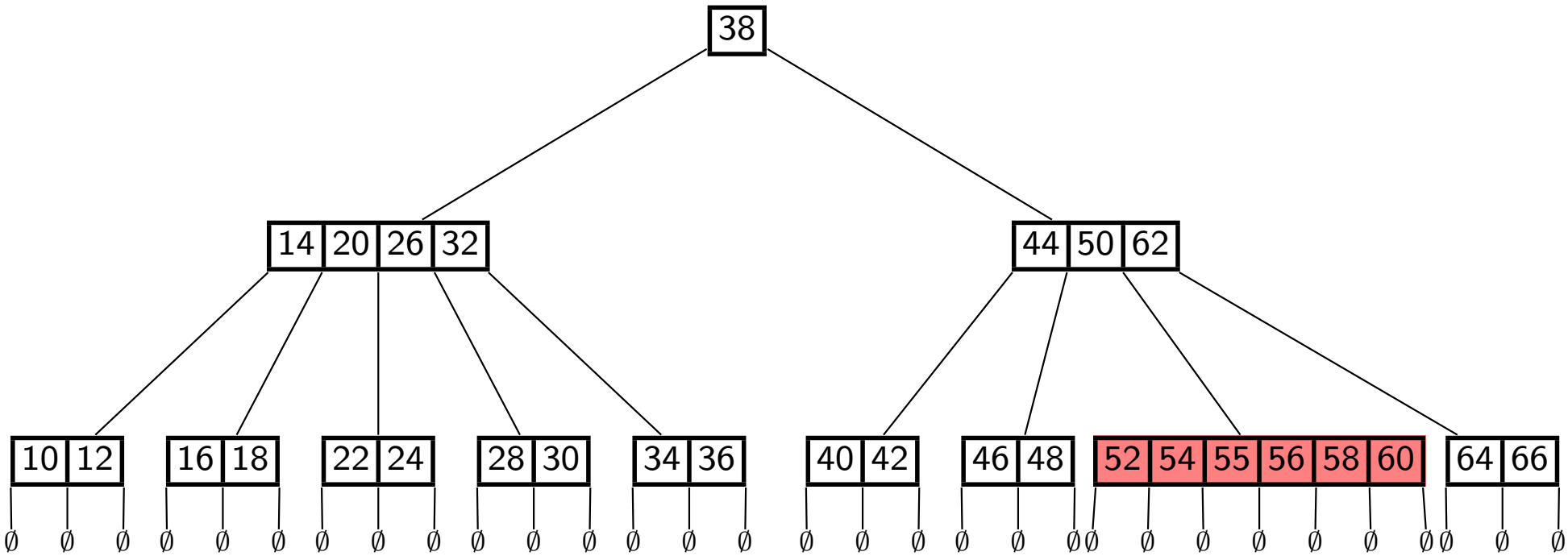
abTree::insert(k)

1. $z \leftarrow$ *abTree::search*(k) `// z: leaf where k should be`
2. Add k and an empty subtree in key-subtree-list of z
3. **while** z has b keys (**overflow** $\rightsquigarrow$ **node split**)
4. Let $\langle T_0, k_1, \ldots, k_b, T_b \rangle$ be key-subtree list at v
5. **if** (z has no parent) create a parent of z without KVPs
6. move upper median k_m of keys to parent p of z
7. $z' \leftarrow$ new node with $\langle T_0, k_1, \ldots, k_{m-1}, T_{m-1} \rangle$
8. $z'' \leftarrow$ new node with $\langle T_m, k_{m+1}, \ldots, k_b, T_b \rangle$
9. Replace $\langle z \rangle$ by $\langle z', k_m, z'' \rangle$ in key-subtree-list of p
10. $z \leftarrow p$



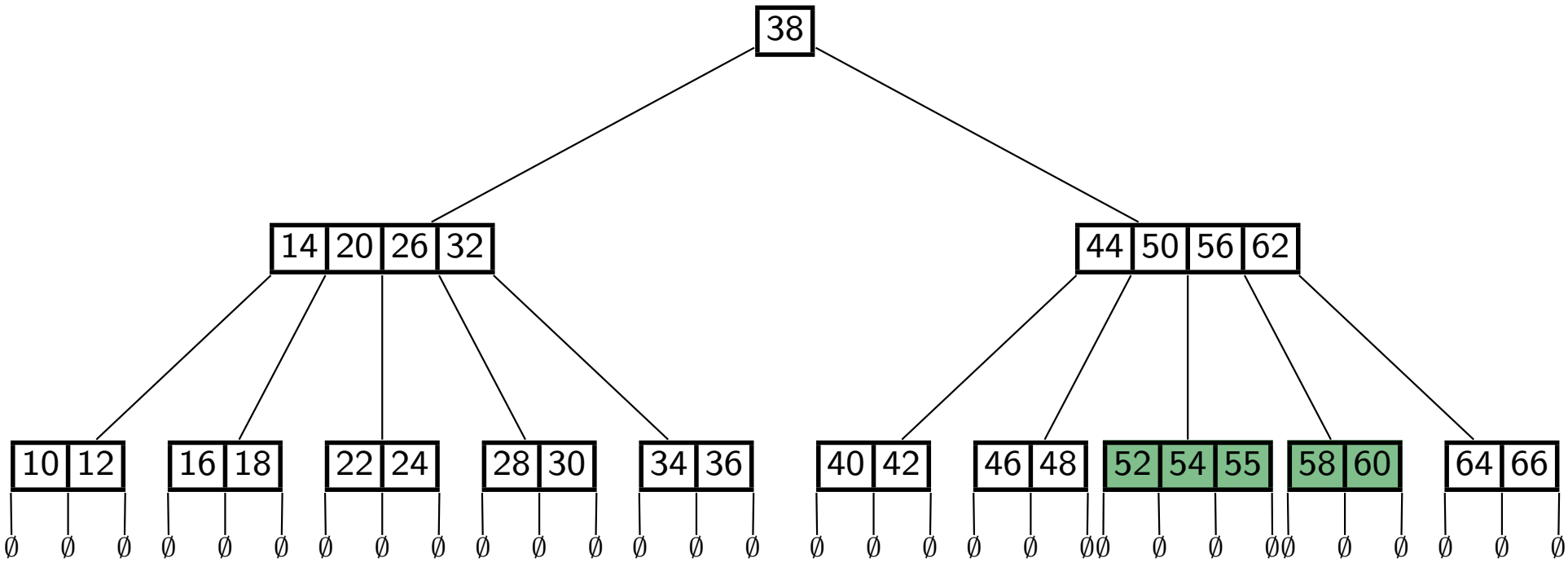
a-*b*-tree *insert*

Example: *insert*(55) in a 3-6-tree:



a-*b*-tree *insert*

Example: *insert*(55) in a 3-6-tree:



- Node split $\Rightarrow$ new nodes have $\geq \lfloor (b-1)/2 \rfloor = \lceil b/2 \rceil - 1$ keys
- Since we know $a \leq \lceil b/2 \rceil$, this is $\geq a-1$ keys as required.

a-b-tree Summary

- An a-b tree has height $O(\log_a n)$
- If $a \approx b/2$, then this height-bound is tight.
  - Level i contains at most b^i nodes
  - Each node contains at most $b-1$ KVPs
  - So $n \leq b^{h+1} - 1$ and $h \in \Omega(\log_b n)$.

a-b-tree Summary

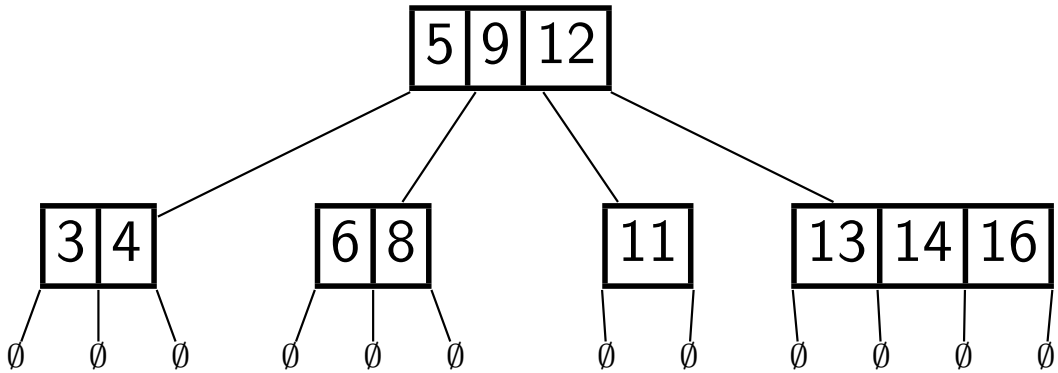
- An a-b tree has height $O(\log_a n)$
- If $a \approx b/2$, then this height-bound is tight.
  - Level i contains at most b^i nodes
  - Each node contains at most $b-1$ KVPs
  - So $n \leq b^{h+1} - 1$ and $h \in \Omega(\log_b n)$.
- *search* and *insert* visit $O(\log_a n)$ nodes.
- *delete* can also be implemented with $O(\log_a n)$ node-visits.
  But usually use *lazy deletion*—space is cheap in external memory.

a-b-tree Summary

- An a-b tree has height $O(\log_a n)$
- If $a \approx b/2$, then this height-bound is tight.
  - Level i contains at most b^i nodes
  - Each node contains at most $b - 1$ KVPs
  - So $n \leq b^{h+1} - 1$ and $h \in \Omega(\log_b n)$.
- *search* and *insert* visit $O(\log_a n)$ nodes.
- *delete* can also be implemented with $O(\log_a n)$ node-visits.
  But usually use *lazy deletion*—space is cheap in external memory.
- How do we choose the order b? (Recall: a is usually $\lceil \frac{b}{2} \rceil$.)
  - Option 1: b small, e.g. $b = 4$
    $\rightsquigarrow$ a new balanced BST, competetive with AVL-trees.
  - Option 2: b big (but one node still fits into one block of memory)
    $\rightsquigarrow$ a realization of ADT Dictionary for external memory

2-4-trees

Consider the special case of $b = 4$ (hence $a = 2$):

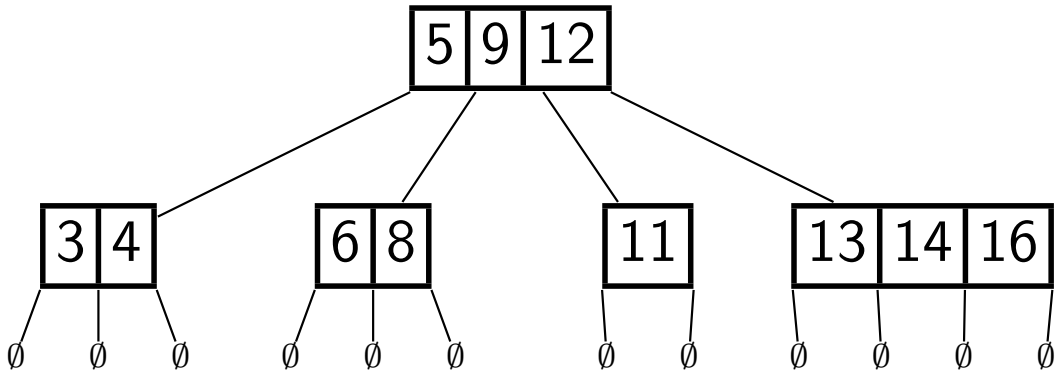


- We analyze here the runtime in the RAM-model (include cost of operations in internal memory)
- Height is $O(\log n)$, operations visit $O(\log n)$ nodes.
- Each node stores $O(1)$ keys and subtrees, so $O(1)$ time spent at node.

$\Rightarrow$ All operations take $O(\log n)$ **worst-case time**.

2-4-trees

Consider the special case of $b = 4$ (hence $a = 2$):



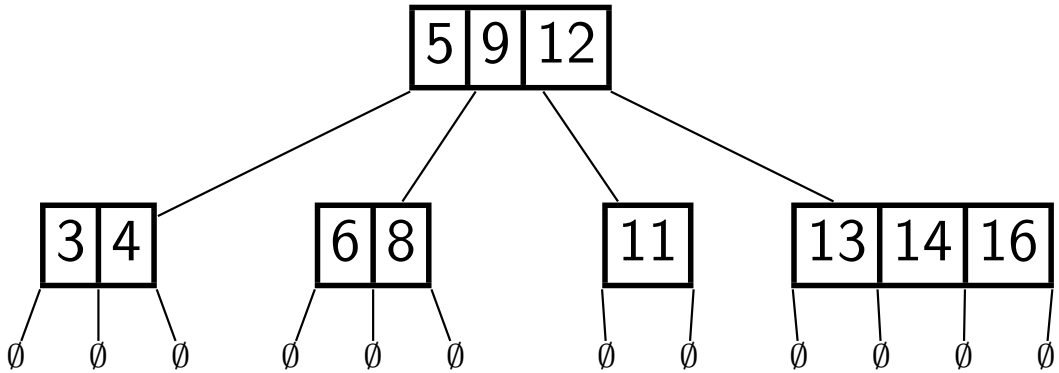
- We analyze here the runtime in the RAM-model (include cost of operations in internal memory)
- Height is $O(\log n)$, operations visit $O(\log n)$ nodes.
- Each node stores $O(1)$ keys and subtrees, so $O(1)$ time spent at node.

$\Rightarrow$ All operations take $O(\log n)$ **worst-case time**.

This is the same as AVL-trees in theory.
But we can make them even better in practice.

Towards red-black-trees

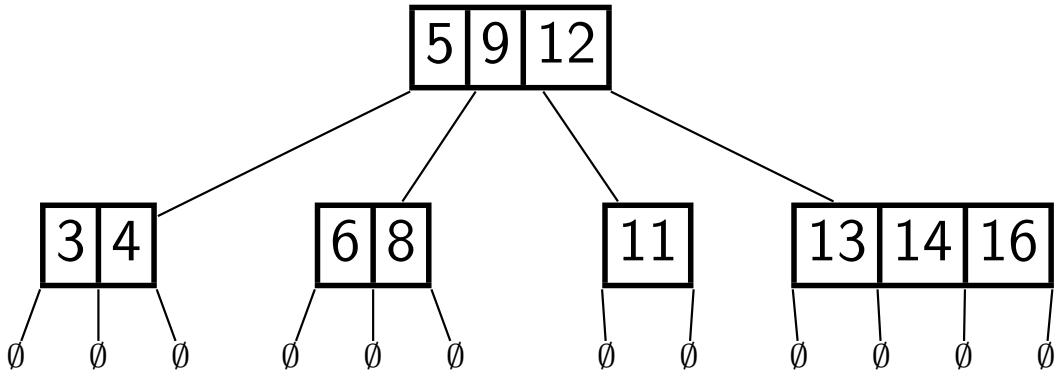
Problems with 2-4-trees:



- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?

Towards red-black-trees

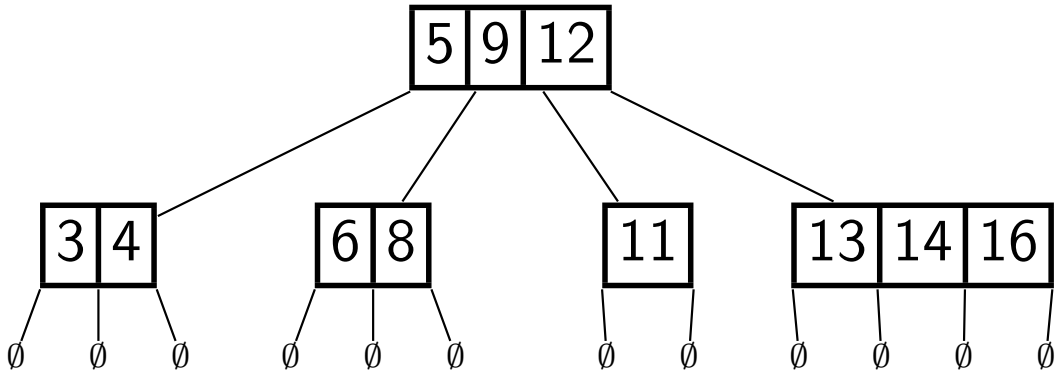
Problems with 2-4-trees:



- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?
  - Array? Then we must use length 7. *This wastes space.*

Towards red-black-trees

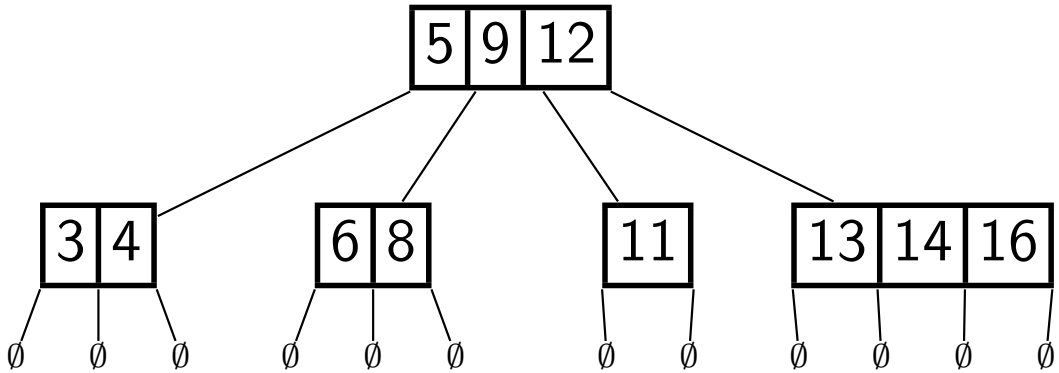
Problems with 2-4-trees:



- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?
  - Array? Then we must use length 7. *This wastes space.*
  - Linked list? We have overhead for list-nodes. *This wastes space.*

Towards red-black-trees

Problems with 2-4-trees:

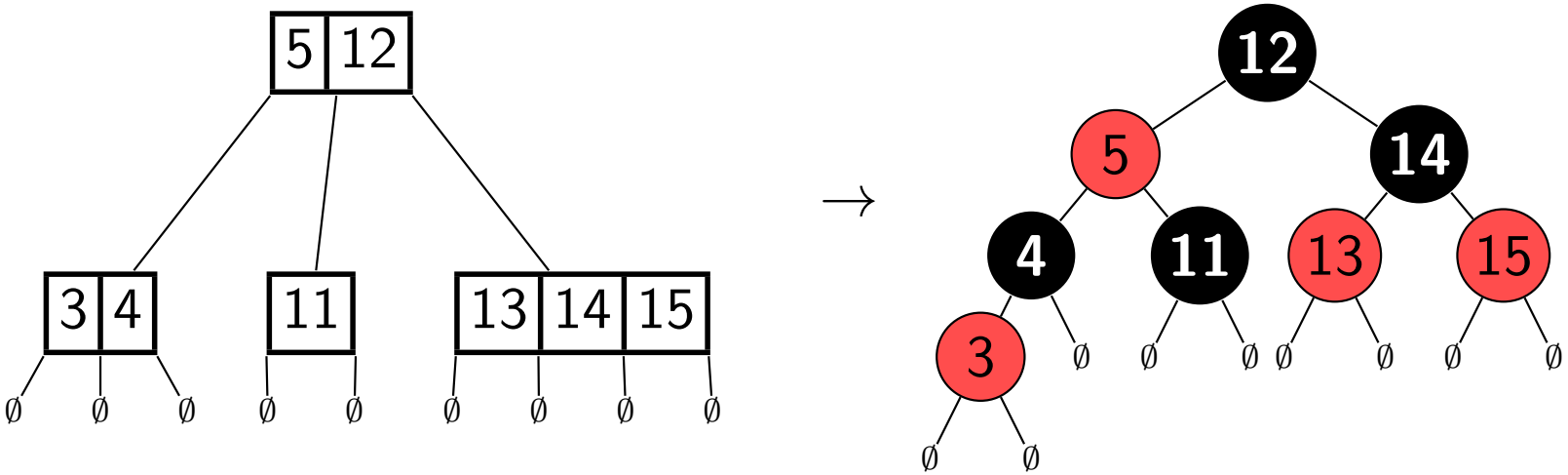


- Recall: We have three kinds of nodes (1-node, 2-node, 3-node) so up to 7 items (keys and subtree-references) at a node.
- *insert* can change the number of keys and subtrees at a node.
- How should we store key-subtree list?
  - Array? Then we must use length 7. *This wastes space.*
  - Linked list? We have overhead for list-nodes. *This wastes space.*

  It does not matter for the theoretical bound, but matters in practice.

Better idea: Design a class of binary search trees that mirrors 2-4-trees!

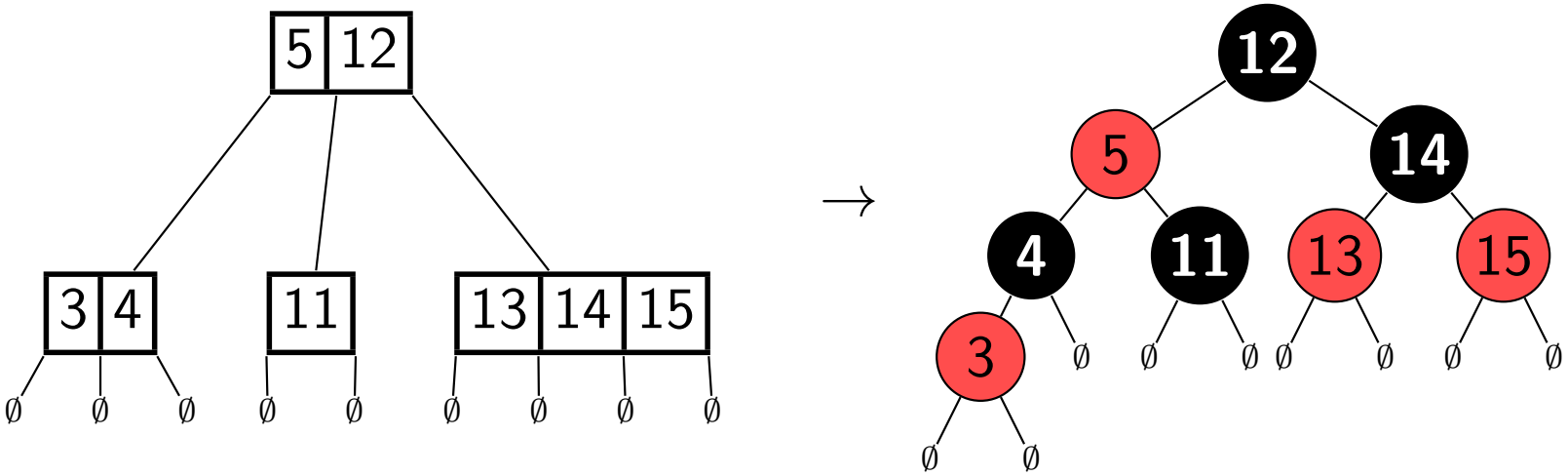
2-4-tree to red-black-tree



Converting a 2-4-tree:

- A d-node becomes a black node with $d-1$ red children (Assembled so that they form a BST of height at most 1.)

2-4-tree to red-black-tree



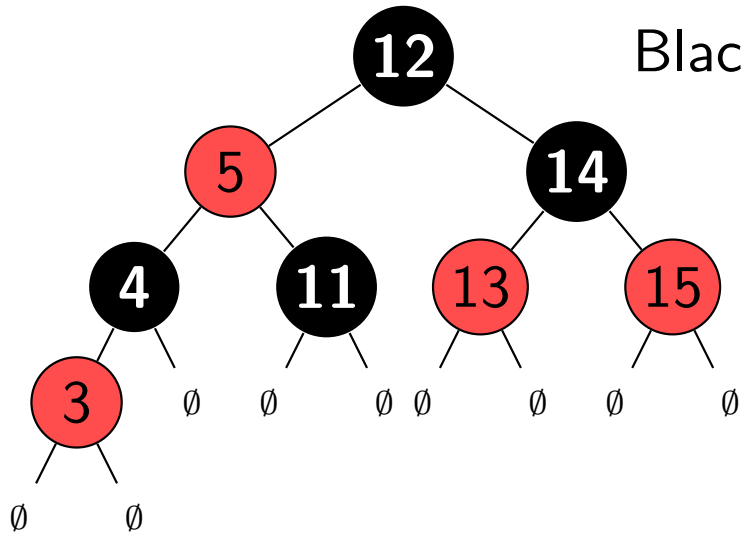
Converting a 2-4-tree:

- A d-node becomes a black node with $d-1$ red children
  (Assembled so that they form a BST of height at most 1.)

Resulting properties:

- Any red node has a black parent.
- Any empty subtree T has the same **black-depth**
  (number of black nodes on path from root to T)

Red-black-trees



Definition: A **red-black tree** is a binary search tree such that

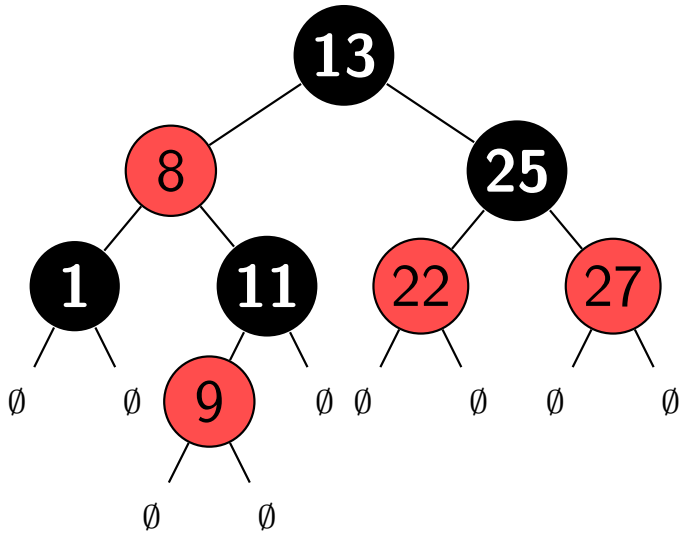
- every node has a color (red or black),
- every red node has a black parent
  (in particular the root is black),
- any empty subtree T has the same black-depth
  (number of black nodes on path from root to T)

Note: Can store this with only *one bit* overhead per node.

Red-black tree to 2-4-tree

Rather than proving properties or describing operations directly, we convert back to 2-4-trees.

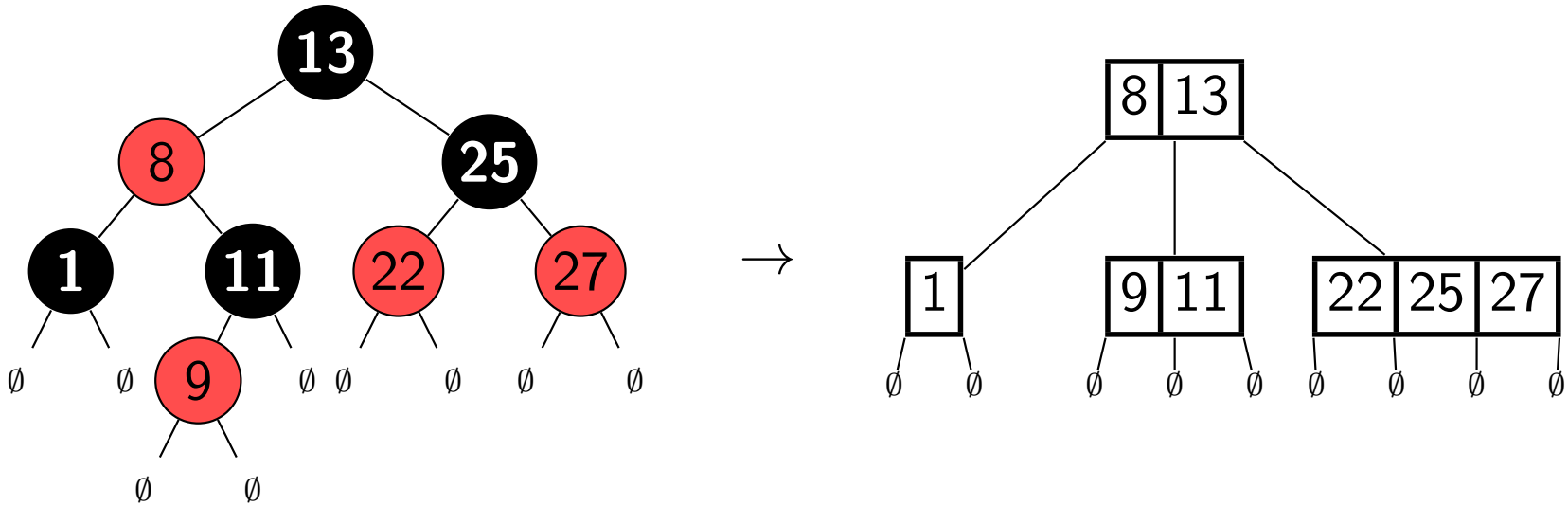
Lemma: Any red-black tree T can be converted into a 2-4-tree T'.



Red-black tree to 2-4-tree

Rather than proving properties or describing operations directly, we convert back to 2-4-trees.

Lemma: Any red-black tree T can be converted into a 2-4-tree T'.



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 same level due to the same blackdepth

Red-black tree summary

- Red-black trees have height $O(\log n)$.
  - Each level of the 2-4-tree creates at most 2 levels in the red-black tree.

Red-black tree summary

- Red-black trees have height $O(\log n)$.
  - Each level of the 2-4-tree creates at most 2 levels in the red-black tree.

- *insert* can be done in $O(\log n)$ worst-case time.
  - Convert relevant part to 2-4-tree.
  - Do insertion in the 2-4-tree.
  - Convert relevant parts back to red-black tree.

  It can actually be done in the red-black tree directly, using only rotations and recoloring (no details).
- *delete* can also be done in $O(\log n)$ worst-case time (no details)

Red-black tree summary

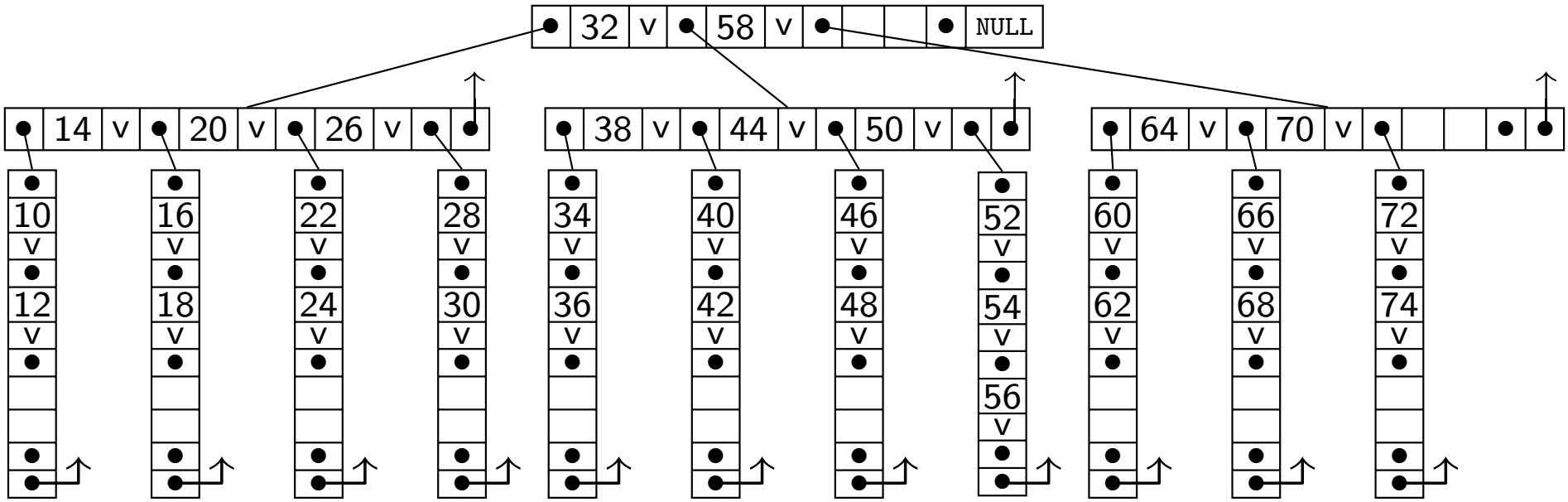
- Red-black trees have height $O(\log n)$.
  - Each level of the 2-4-tree creates at most 2 levels in the red-black tree.
- *insert* can be done in $O(\log n)$ worst-case time.
  - Convert relevant part to 2-4-tree.
  - Do insertion in the 2-4-tree.
  - Convert relevant parts back to red-black tree.

  It can actually be done in the red-black tree directly, using only rotations and recoloring (no details).
- *delete* can also be done in $O(\log n)$ worst-case time (no details)
- Experiments show that red-black tree use fewer rotations than AVL-trees.
- This is a very popular balanced binary search tree (`std::map`)

B-trees

A **B-tree** is an a-b-tree tailored to the external memory model.

- Every node is one block of memory (of size B).
- The order b is chosen maximally such that $(b-1)$-node fits into a block of memory. Typically $b \in \Theta(B)$.
- a is set to be $\lceil b/2 \rceil$ as before.



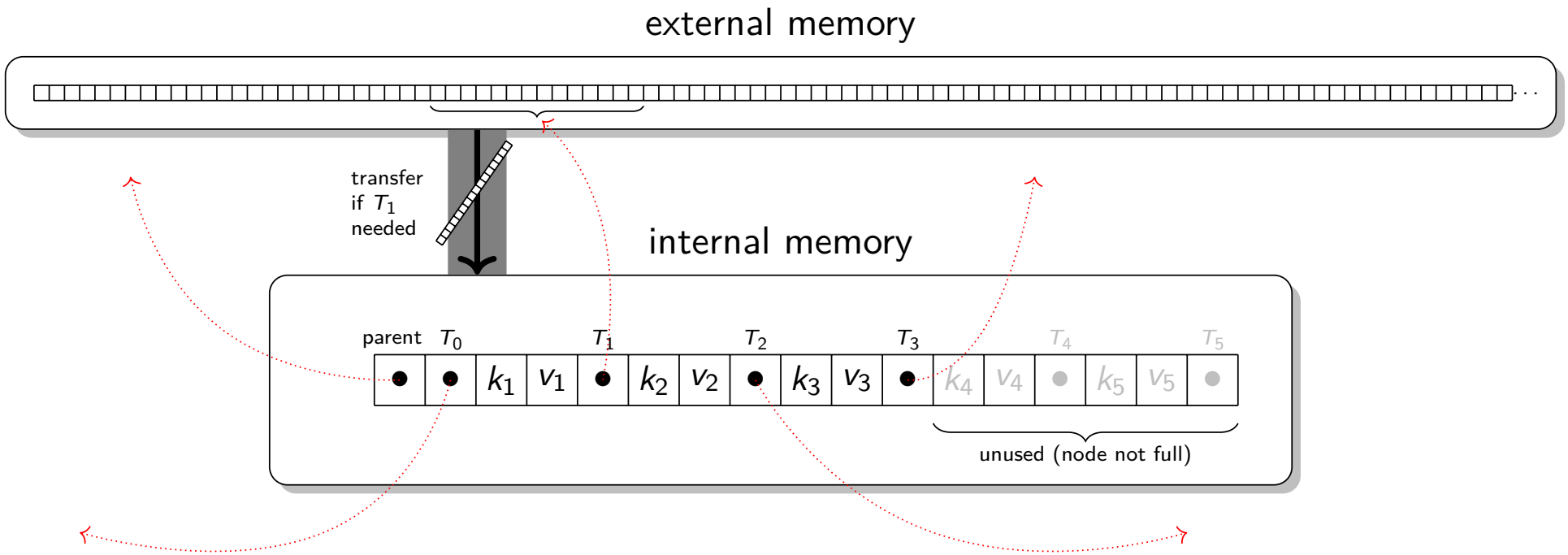
('v' indicates the value or value-reference associated with the key next to it)

(arrows indicate references to the parent)

B-tree Close-up

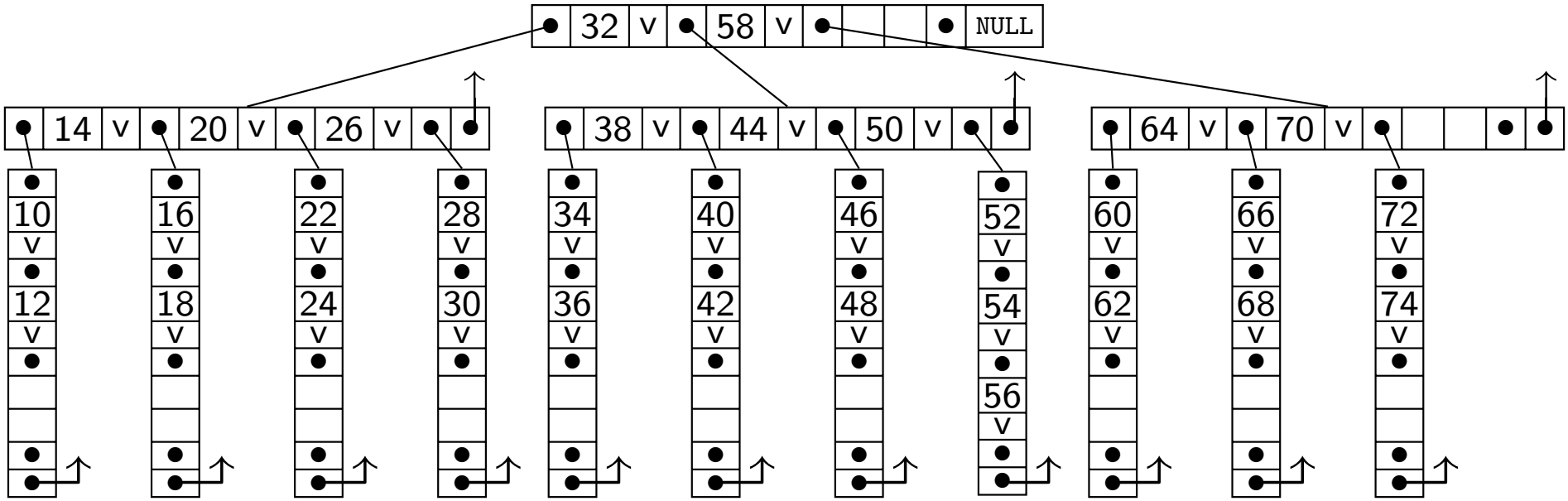
To see how to choose the order b, inspect a $(b-1)$-node:

- Stoe $b-1$ keys and $b-1$ values
- Store b references to subtrees
- Store parent-reference



In this example: $B = 17$ memory cells fit into one block, so we would choose order $b = 6$.

B-tree analysis



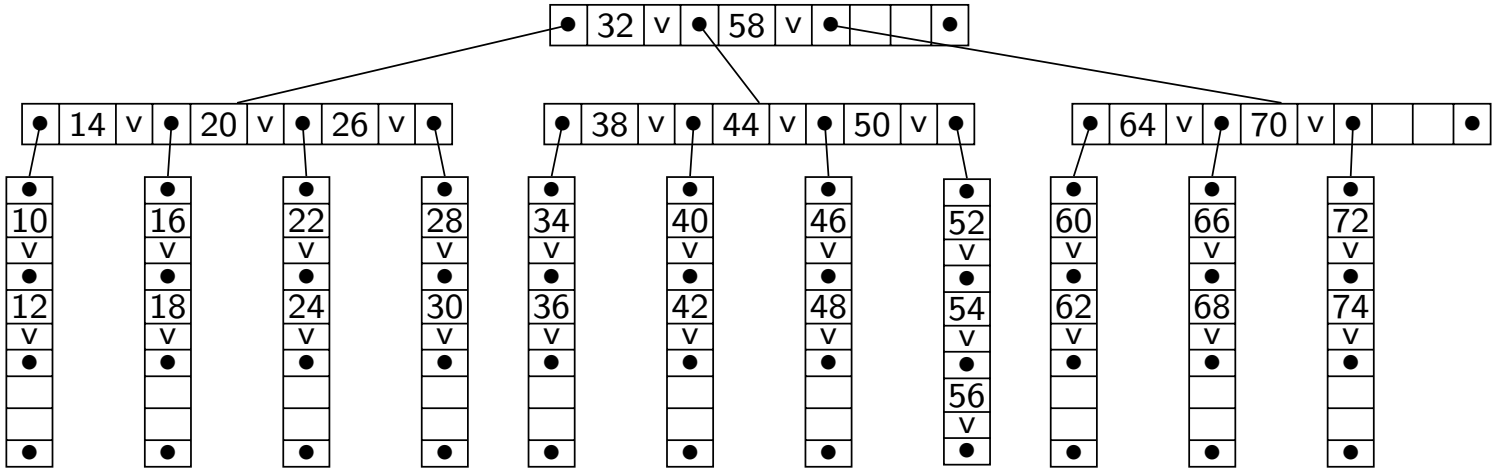
- *search*, *insert*, and *delete* each requires visiting $\Theta(\mathit{height})$ nodes
- Work within a node is done in internal memory $\Rightarrow$ no block-transfer.
- The height is $\Theta(\log_a n) = \Theta(\log_B n)$ (since $a = \lceil b/2 \rceil \in \Theta(B)$)

So all operations require $\Theta(\log_B n)$ **block transfers**.

B-tree summary

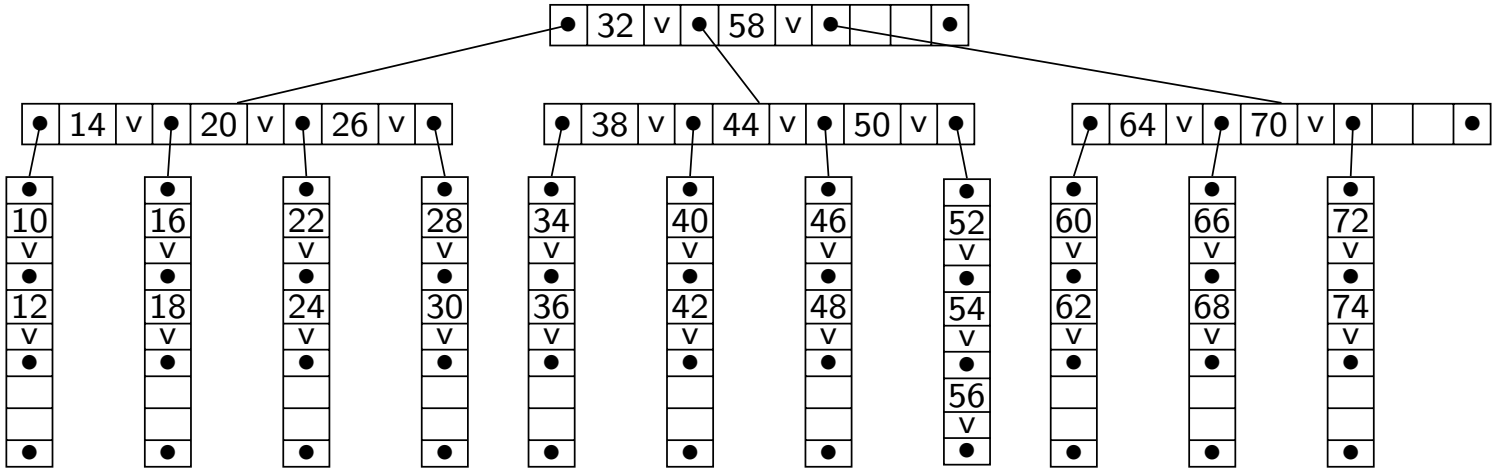
- All operations require $\Theta(\log_B n)$ **block transfers**.
  - This is asymptotically optimal.
  - **Can show:** Searching among n items requires $\Omega(\log_B n)$ block transfers.
- In practice, height is a small constant.
  - Say $n = 2^{50}$, and $B = 2^{15}$. So roughly $b = \frac{1}{3}2^{15}$, $a = \frac{1}{3}2^{14}$.
  - B-tree of height 4 would have $\geq 2a^4 - 1 > 2^{50}$ KVPs.
  - So height is 3.
- B-trees are hugely important for storing data bases ($\rightsquigarrow$ cs448)
- Study now: variations that are even better in practice

Pre-emptive splitting/merging



- Observe: *BTree::insert*(k, v) traverses tree twice:
  - Search down on a path to the leaf where we add (k, v).
  - Go back up on the path to fix overflow, if needed.
- So the number of block-transfers could be twice the height.
- How can we avoid this?

Pre-emptive splitting/merging

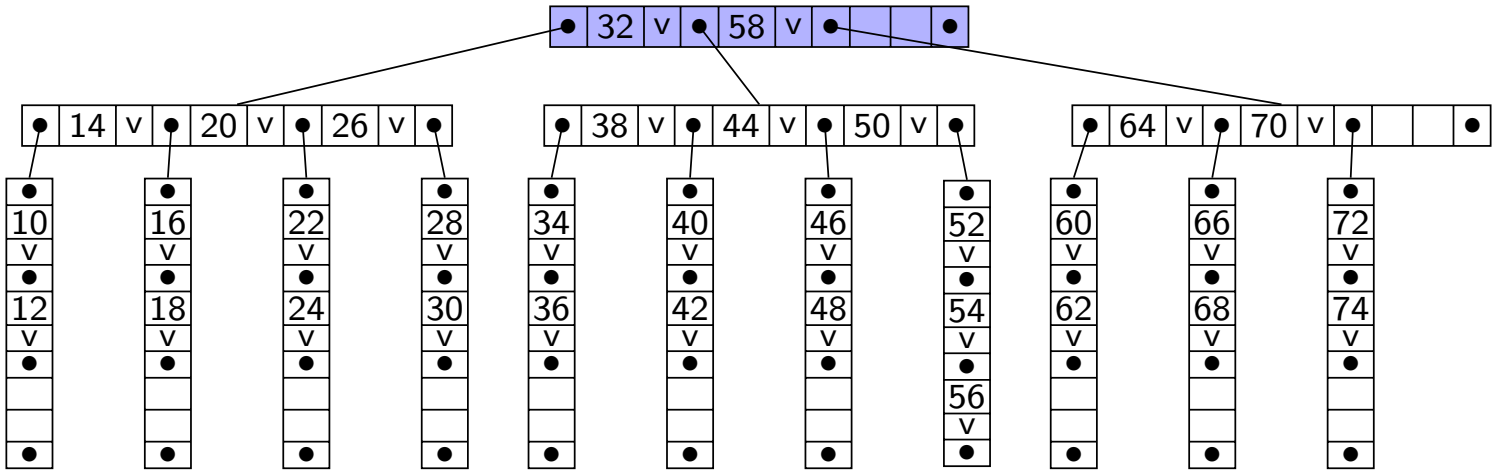


- Observe: *BTree::insert*(k, v) traverses tree twice:
  - Search down on a path to the leaf where we add (k, v).
  - Go back up on the path to fix overflow, if needed.
- So the number of block-transfers could be twice the height.
- How can we avoid this?

- **Idea:** During the search, *always* split if the node is full.
- Then a node split at the leaf does not create an overfull parent.

Pre-emptive splitting/merging example

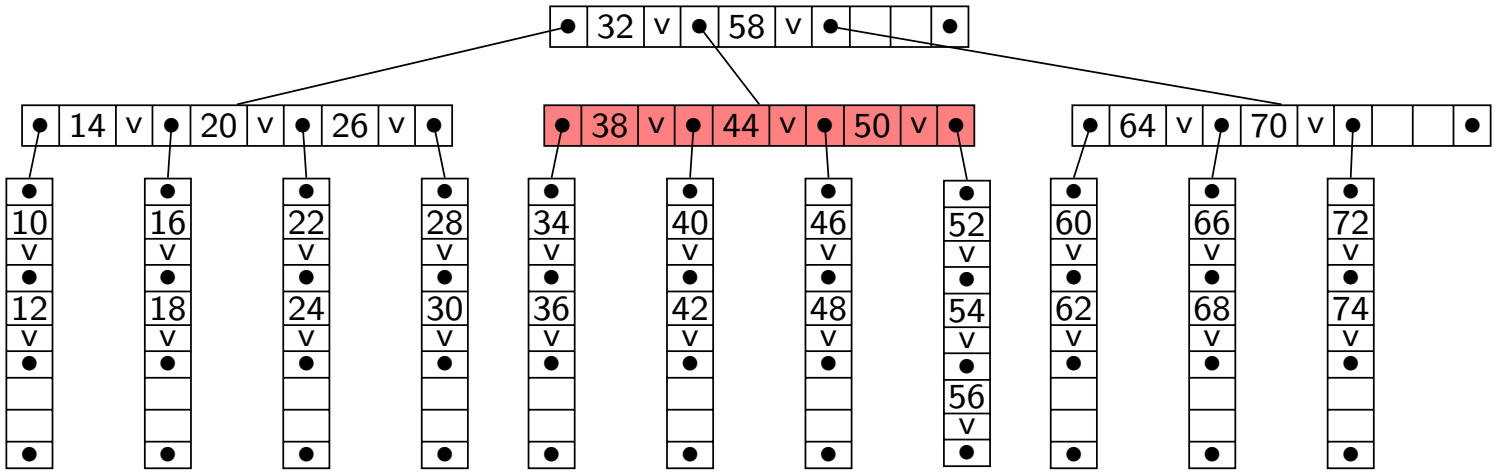
PreemptiveBTree::insert(49):



- If node is not full, keep searching.

Pre-emptive splitting/merging example

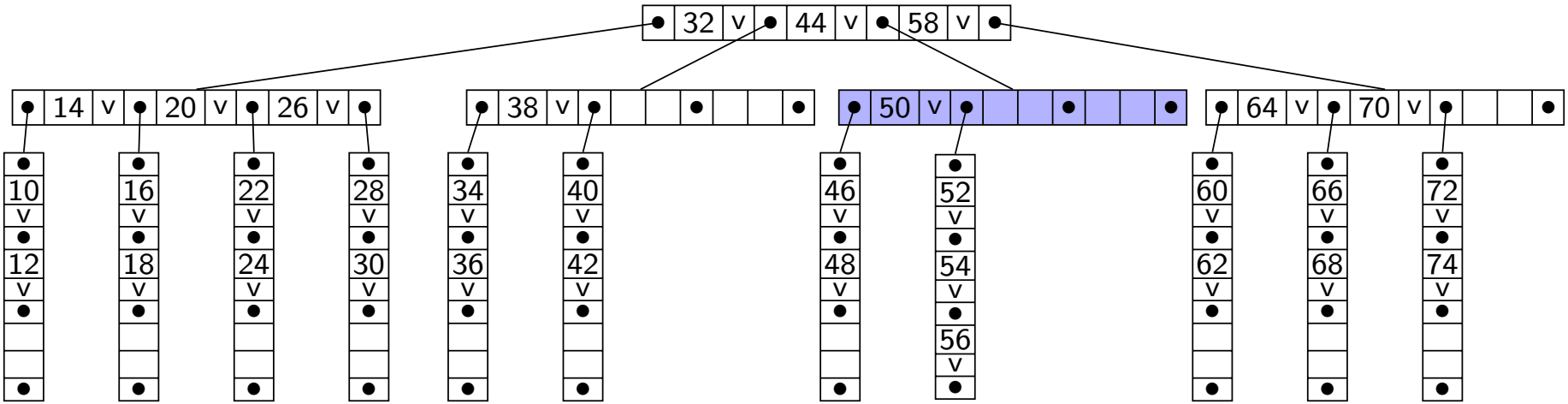
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.

Pre-emptive splitting/merging example

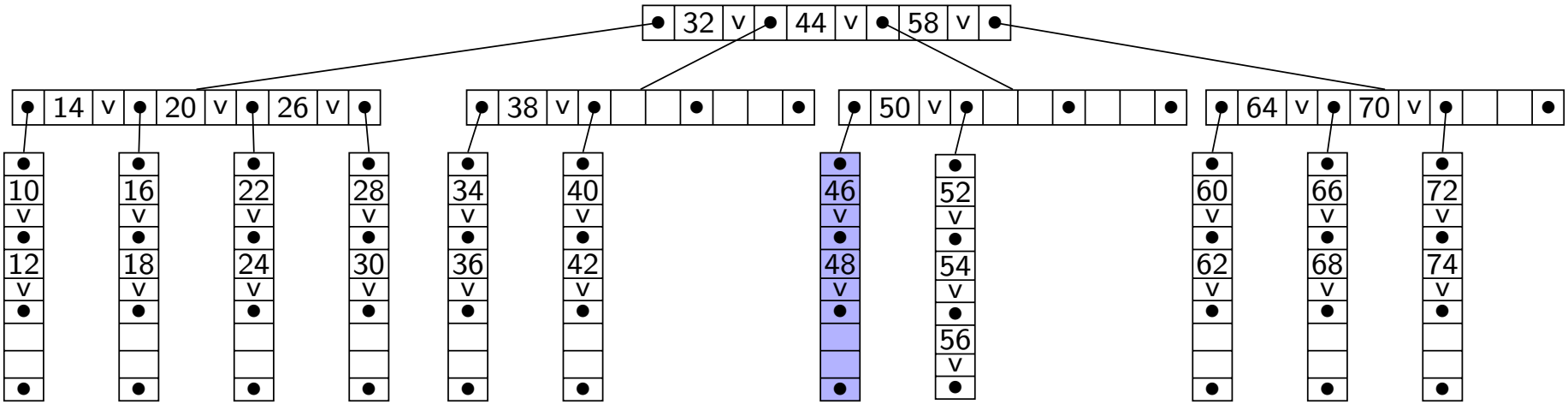
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.

Pre-emptive splitting/merging example

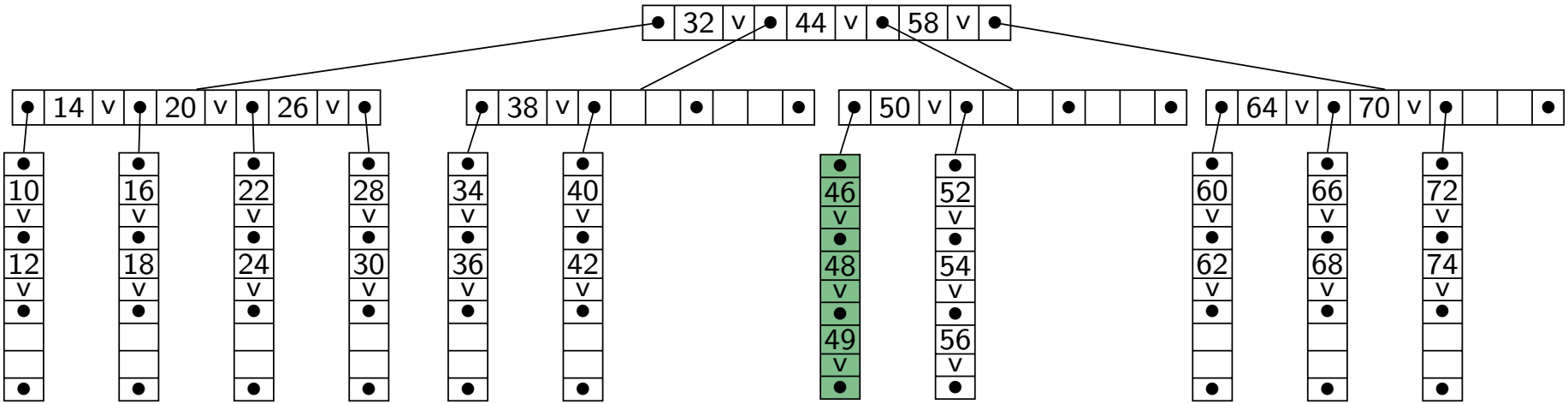
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.
- We may have split unnecessarily. (But space is cheap.)

Pre-emptive splitting/merging example

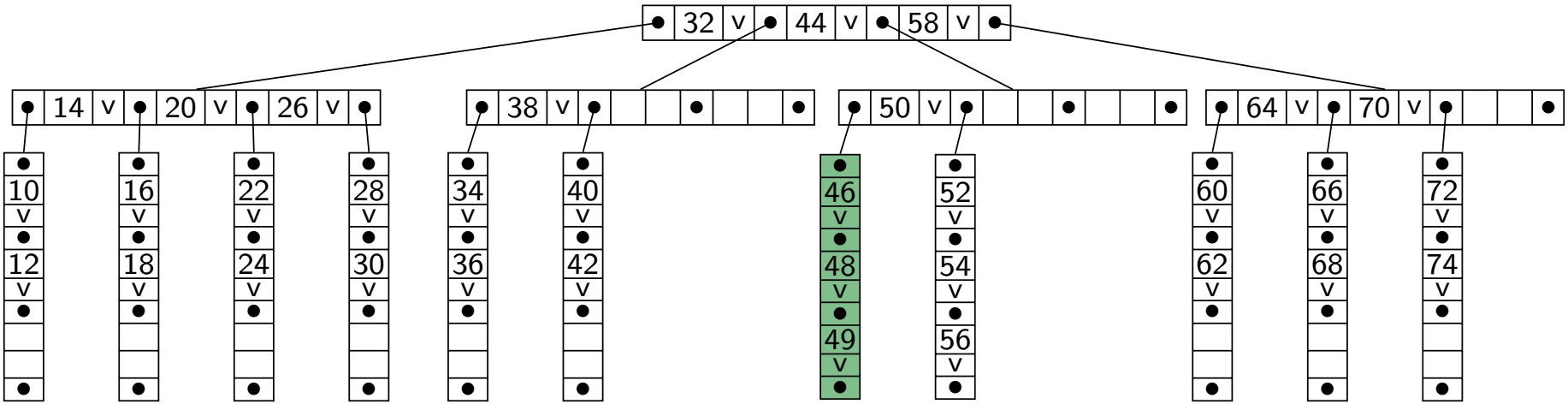
PreemptiveBTree::insert(49):



- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.
- We may have split unnecessarily. (But space is cheap.)

Pre-emptive splitting/merging example

PreemptiveBTree::insert(49):

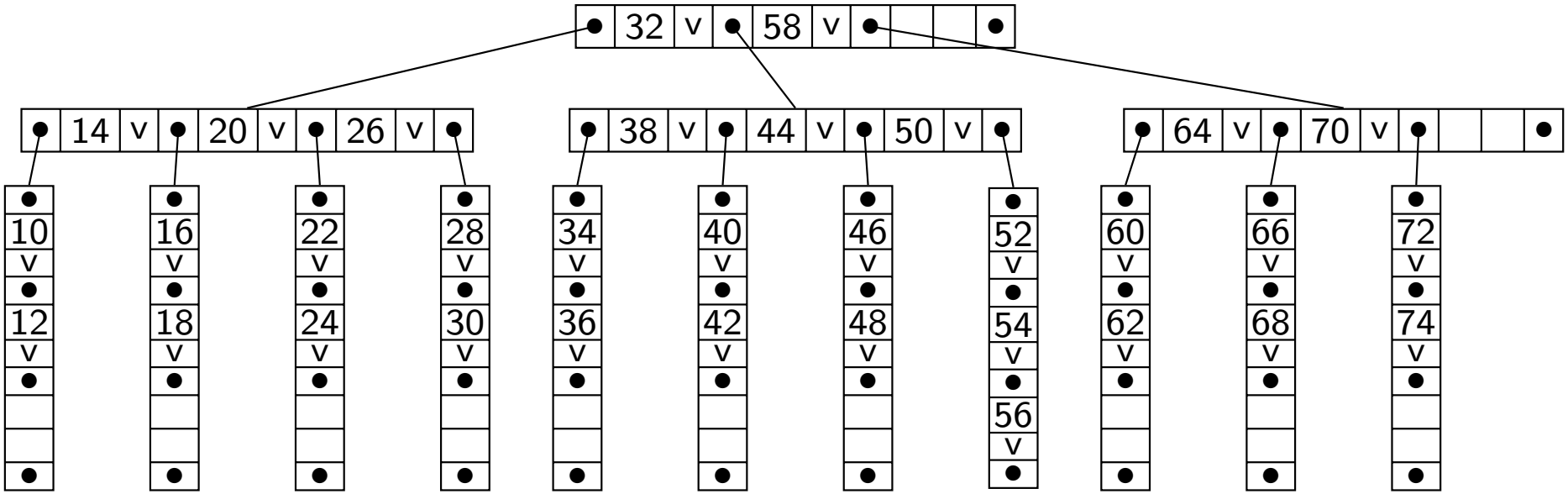


- If node is not full, keep searching.
- If node is full, immediately split.
- Then keep searching in appropriate new node.
- We may have split unnecessarily. (But space is cheap.)

- With this, we no longer need parent-references.

Towards B^+-trees

In a B-tree, each node is one block of memory. In this example, up to 10 keys/references fit into one block, so the order is 4.



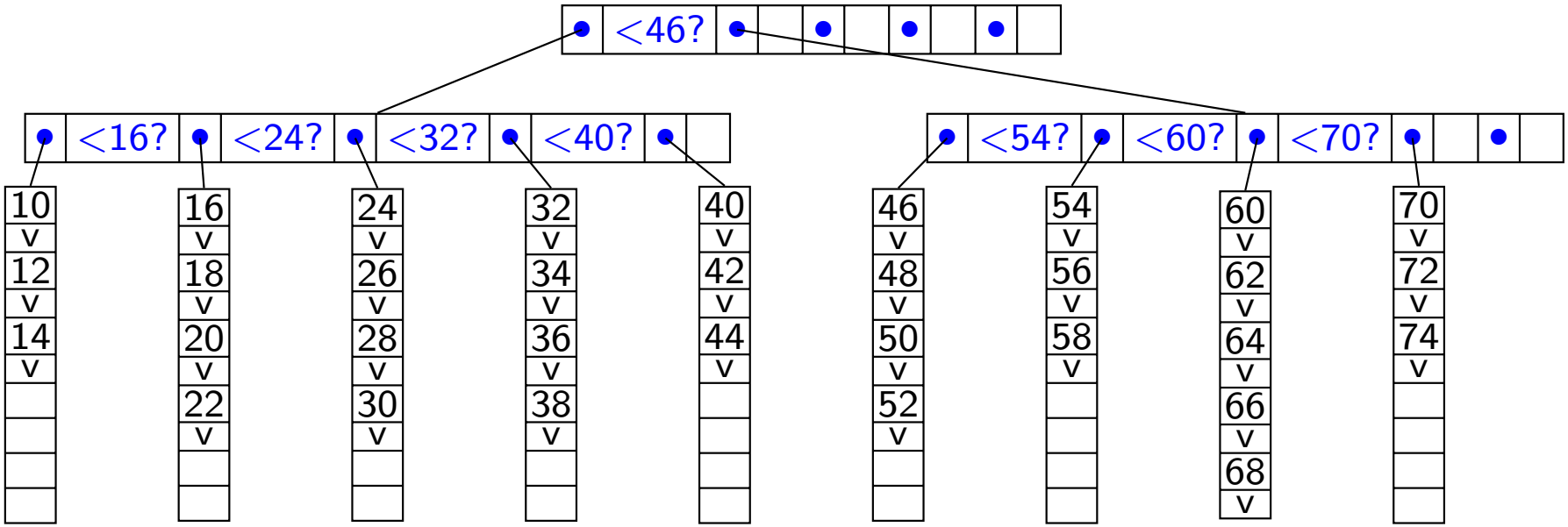
This B-tree could store up to 63 KVPs with height 2.

Two ideas to achieve smaller height:

1. The leaves are wasting space for references that will never be used.
2. Use a *decision-tree version* $\Rightarrow$ inner nodes can have more children.

B^+-trees

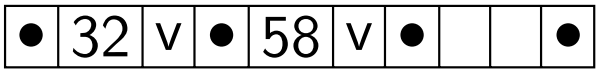
- Each node is one block of memory.
- All KVPs are stored at *leaves*. Each leaf is at least half full.
- *Interior nodes* store only keys for comparison during search.
- Interior (non-root) nodes have at least half of the possible subtrees.
- Use pre-emptive splitting.



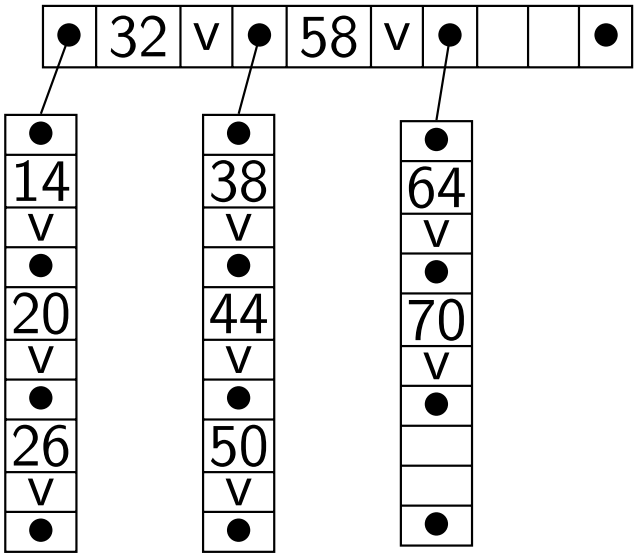
This B^+-tree could store up to 125 KVPs with height 2.

Towards LSM-trees

One block:



B-tree:



Internal

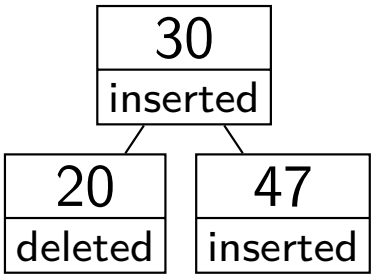
External

- Internal memory only requires 1-2 blocks at a time.
- Roughly $M - 2B$ space free.
- How can we use this to increase speed for updates?

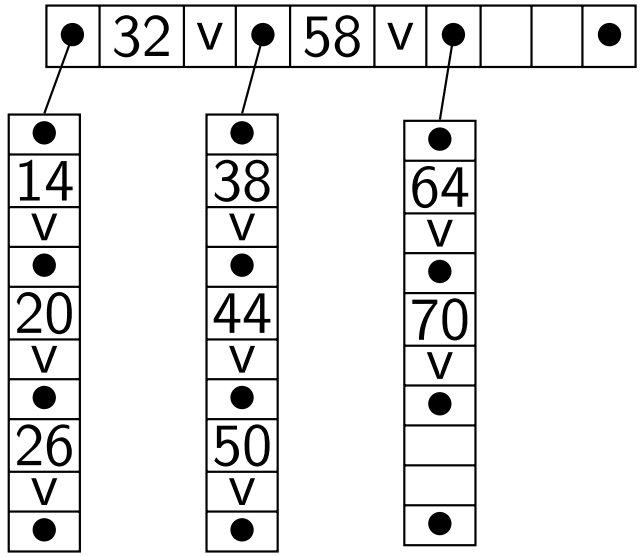
Log-structured Memory trees

One block:

C_0 (log of the changes):



C_1 (B-tree):



Internal | External

- Store dictionary in internal memory that logs all changes
- To search: first search in C_0, then (if needed) in C_1
- If internal memory full: do lots of updates in C_1 at once

Summary

- The RAM model is convenient for algorithm analysis.
- Many of its assumptions are unrealistic, for example
  - not all memory cells are equally quick to access,
  - not all numbers take equal space, and
  - not all primitive operations take equal time.
- Also, modern computer architectures permit more, for example
  - multi-threading
  - distributed computing
- There are other computer models that take these into account.
  - We saw here the EMM for different types of memory.
- The models get complicated (many parameters!) and the bounds are less helpful (tradeoffs between them).
- The main goal is to get the program-designer to think in the appropriate way.