CS 240 – Data Structures and Data Management

Module 11: External Memory

M. Petrick O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2020

References: Goodrich & Tamassia 20.1-20.3, Sedgewick 16.4

Outline

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

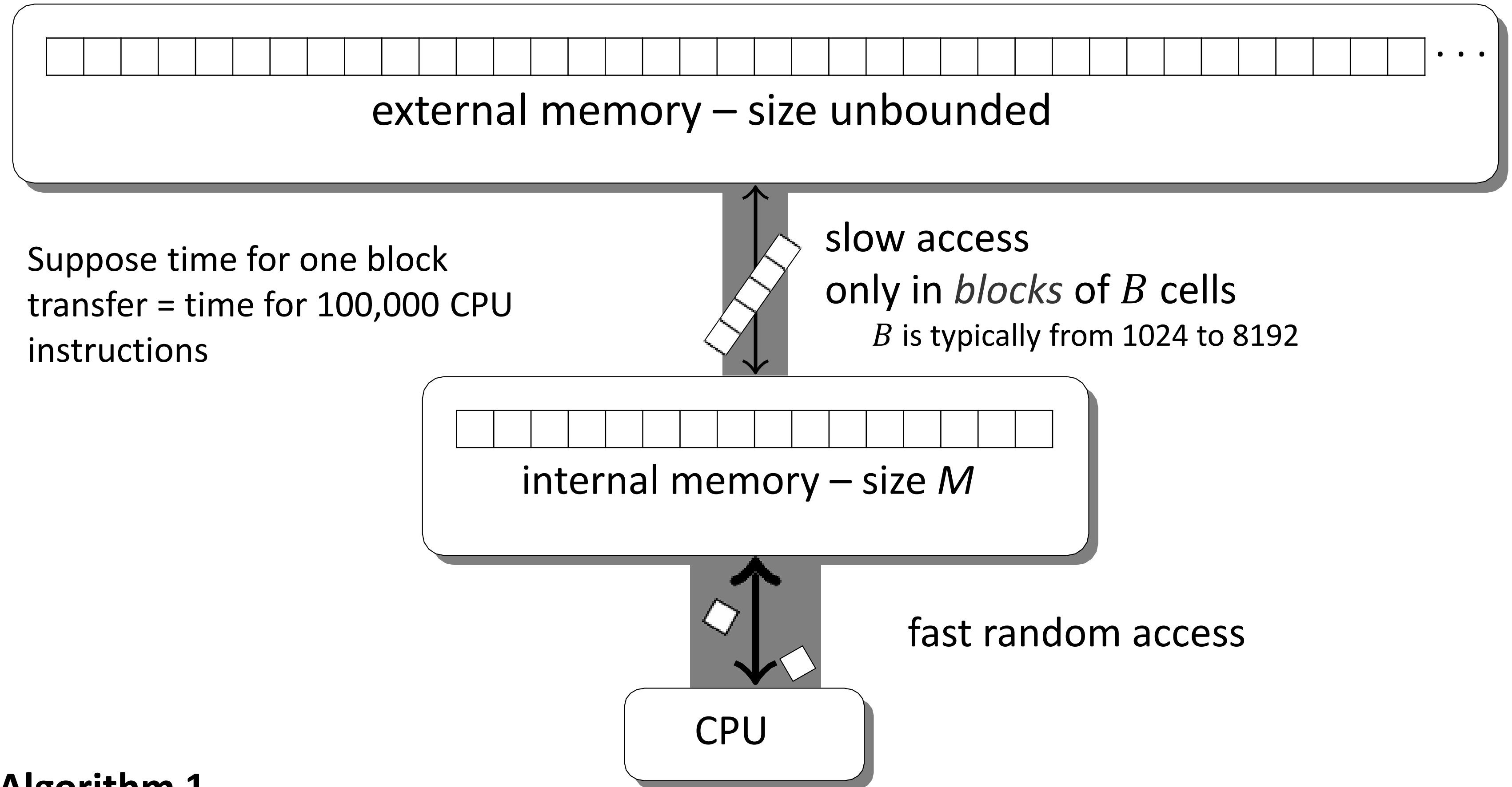
Outline

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

Different levels of memory

- Memory hierarchy for current computer architectures
  - Registers: super fast, very small
  - cache L1, L2: very fast, less small
  - main memory: fast, large
  - disk or cloud: slow, very large
    - from 1000 to 1,000,000 times slower than main memory
- Desirable to minimize transfer between slow/fast memory
- Focus on 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
  - new objective
    - revisit ADTs/problems with the objective of minimizing block transfers ("probes", "disk transfers", "page loads")

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 = $\cancel{1{,}000}+1{,}000\cdot 100{,}000 = \cancel{10^3} + 10^8$

  dominating factors

- **Algorithm 2**

  ~~10,000 CPU instructions~~ + 10 block transfers = $\cancel{10{,}000}+10\cdot 100{,}000 = \cancel{10^4} + 10^6$

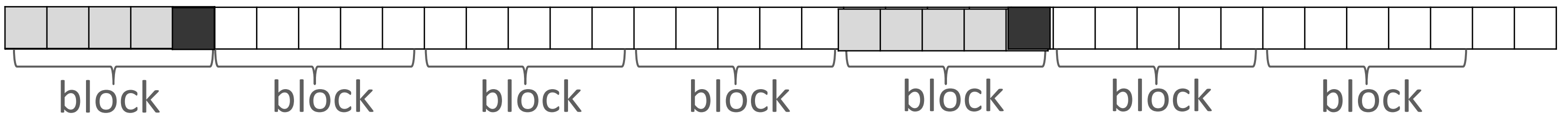
- **Cost of computation:** number of blocks transferred between internal and external memory

Outline

- External Memory
  - Motivation
  - External sorting
  - External Dictionaries
    - 2-4 Trees
    - (*a*, *b*)-Trees
    - B-Trees
  - Extendible Hashing

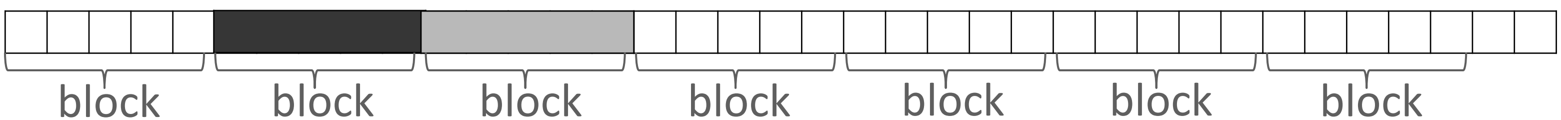
Sorting in external memory

- Sort array A of n numbers
  - assume 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: each iteration can access far apart indices of A



  - accesses 2 blocks, but put only 2 elements in order
  - and all the other data read in the block is not used
  - heapsort does not adapt well to data stored in external memory

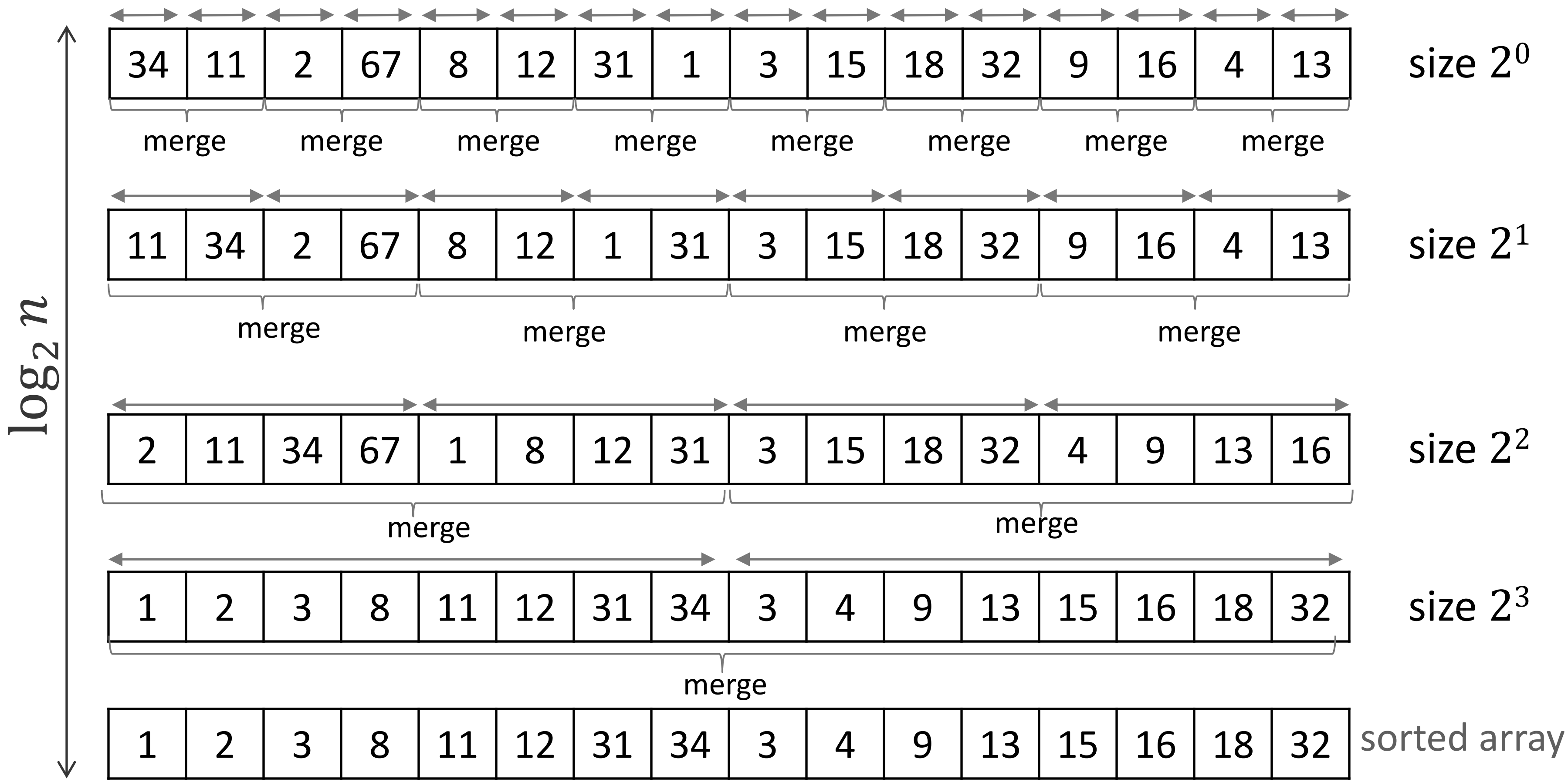
- Mergesort adapts well to array stored in external memory
  - access consecutive locations of A, ideal for reading in blocks



  - accesses 2 blocks, and puts all their elements in order

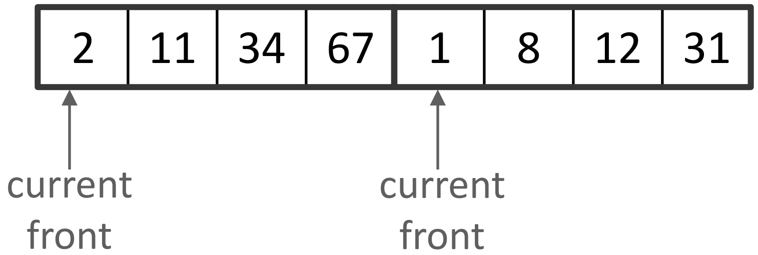
Mergesort: non-recusive view

- 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



2-way Merge

- Two sorted runs



- Put a pointer at the front of each sorted run
  - call it ‘current front’
- Repeatedly find the smallest element among current fronts
  - move the smallest element into sorted result array
  - advance current front of corresponding sorted run
- Array to store sorted result

1							

2-way Merge

- Two sorted runs

2	11	34	67	1	8	12	31

- Put a pointer at the front of each sorted run
  - call it ‘current front’
- Repeatedly find the smallest element among current fronts
  - move the smallest element into sorted result array
  - advance current front of corresponding sorted run
- Array to store sorted result

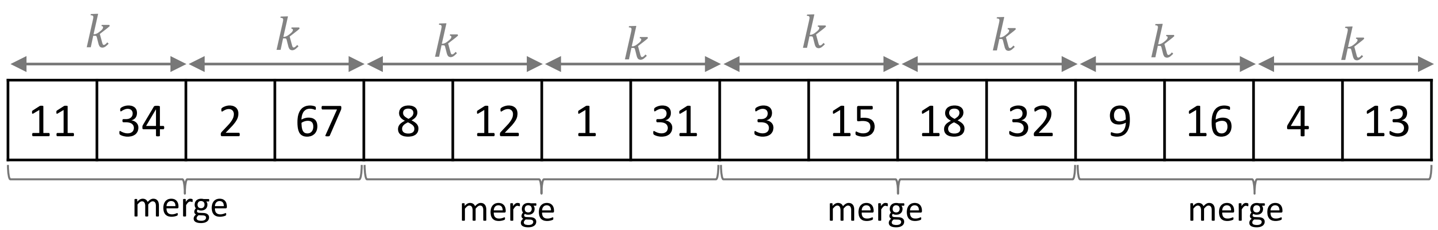
1	2	8	11	12	31	34	64

done!

- Time to merge two sequences each of size k is $\Theta(2k)$

Running time of MergeSort with 2-way Merge

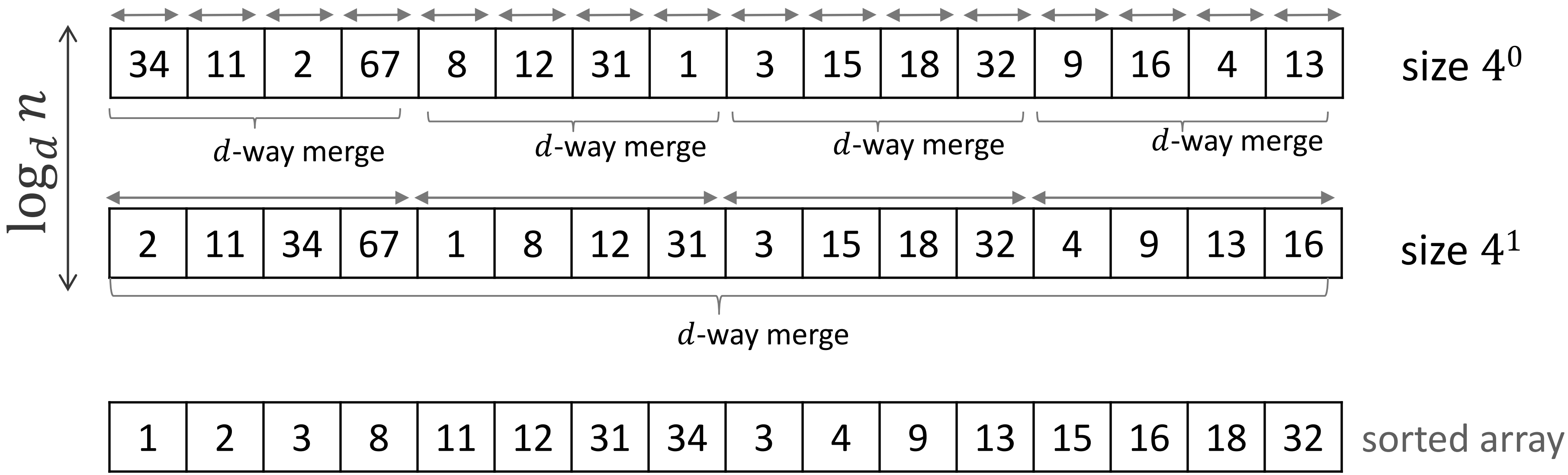
- $\Theta(\log_2 n)$ rounds
- Time for each round
  - time to merge 2 sequences each of size k is $\Theta(2k)$
  - in one round, need to merge $n/(2k)$ sequences pairs



  - one round of merge sort takes $\Theta\big(2k \cdot n/(2k)\big) = \Theta(n)$ time
- Total time for mergesort is $\Theta(n \log_2 n)$

d-way Mergesort

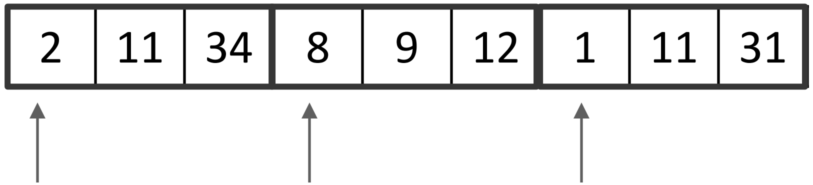
- Can generalize mergesort to merge d sorted runs at one time
  - $d = 2$ gives standard mergesort
- Example: $d = 4$



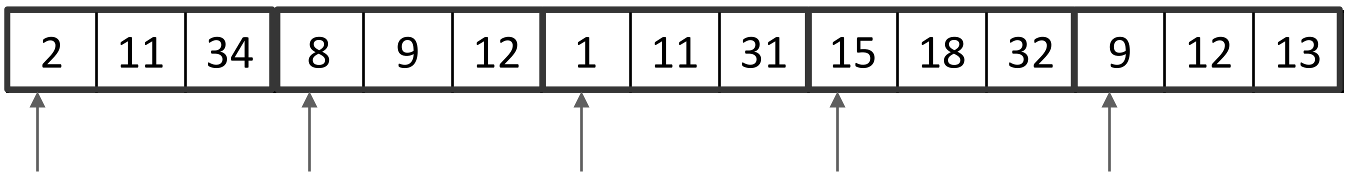
- $\log_d n = \frac{\log_2 n}{\log_2 d}$ rounds
  - the larger is d the less rounds
- How to merge d sorted runs efficiently?
  - d-Way merge

d-way Merge

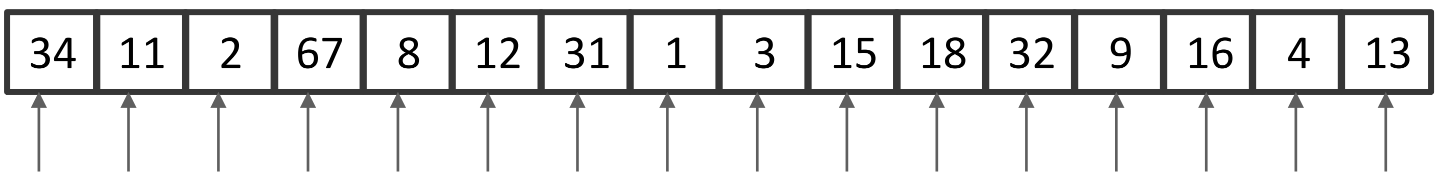
- $d = 3$



- $d = 5$



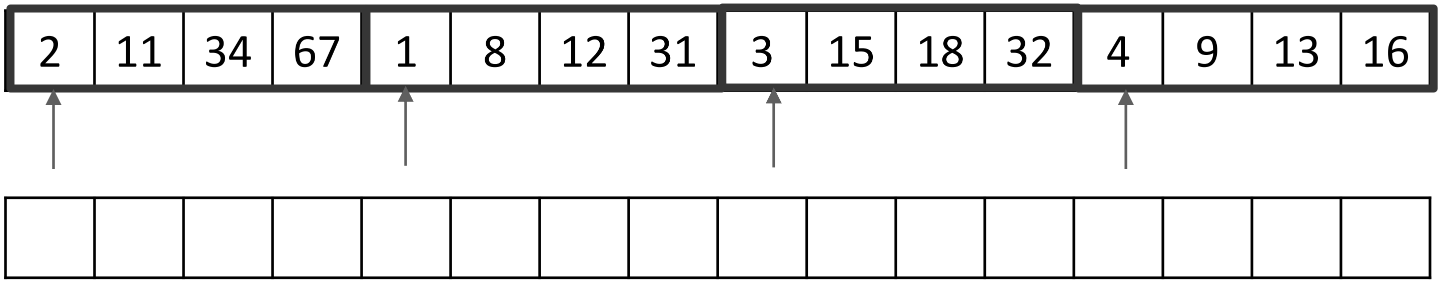
- $d = 16$



- Need efficient data structure to find the minimum among d current fronts

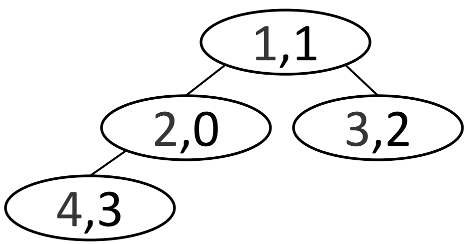
d-way Merge with Min-Heap

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

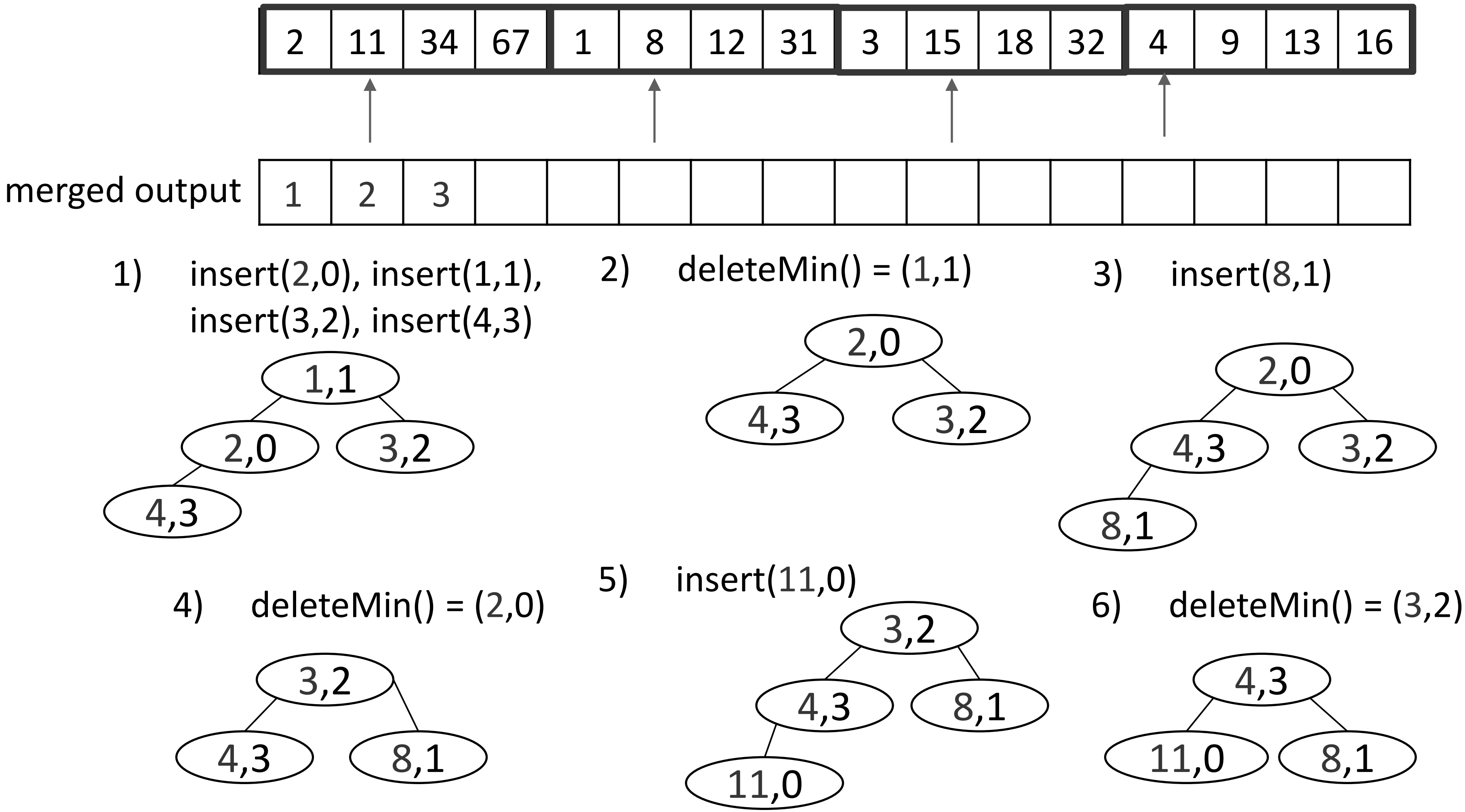


merged output

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



d-way Merge with Min-Heap

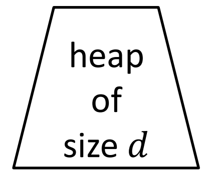


d-way Merge with Min Heap Pseudo Code

d-Way-Merge$(S_1, ..., S_d)$
$S_1, ..., S_d$ are sorted sets (arrays/lists/stacks/queues)
    $P \leftarrow$ empty min-priority queue
    $S \leftarrow$ empty set
    // P always holds current front elements of $S_1, ..., Sd$
    for $i \leftarrow 1$ to d **do**
        P.insert((first element of S_i, i))
    while P is not empty **do**
        $(x, i) \leftarrow$ *deleteMin*(P) // removes current front of S_i from P
        remove x from S_i and append it to S
        if S_i is not empty **do**
        // current front of S_i is not represented in P, add it
            P.insert((first element of S_i, i))

d-way Merge with Min Heap Time Complexity

- Merging d sequences each of size k
- dk iterations, at each iteration
    - one deleteMin() on heap of size d
        - $\Theta(\log_2 d)$
    - one insert() on heap of size d
        - $\Theta(\log_2 d)$
- Total time is $\Theta(dk \log_2 d)$



d-way Mergesort Complexity In Internal Memory

- $\log_d n$ rounds
- Time complexity for one round
  - time to merge d sequences of size is k is $\Theta(kd \log_2 d)$
  - for one round of mergesort , have to do $n/(dk)$ of these merges
  - time for one round is $\Theta\left(\frac{n}{dk} kd \ \log_2 d\right) = \Theta(n \log_2 d)$
- Total time $\Theta(\log_d n \cdot n \log_2 d) = \Theta\left(\frac{\log_2 n}{\cancel{\log_2 d}} \cdot \cancel{n \log_2 d}\right) = \Theta(n \log_2 n)$ no advantage in internal memory

d-way Mergesort Complexity In External Memory

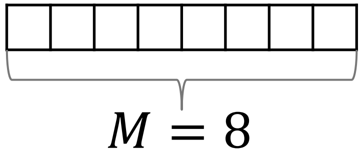
- How do we gain advantage in external memory?
  - we only count block accesses
- $\log_d n$ rounds
  - time for each round is $\Theta(\cancel{n \log_2 d})$ $\Theta(n)$, or better, in block accesses
- Total time $\Theta(\log_d n \cdot \cancel{n \log_2 d}) = \Theta(\cancel{n \log_2 n})$

  $\Theta(n)$ block accesses

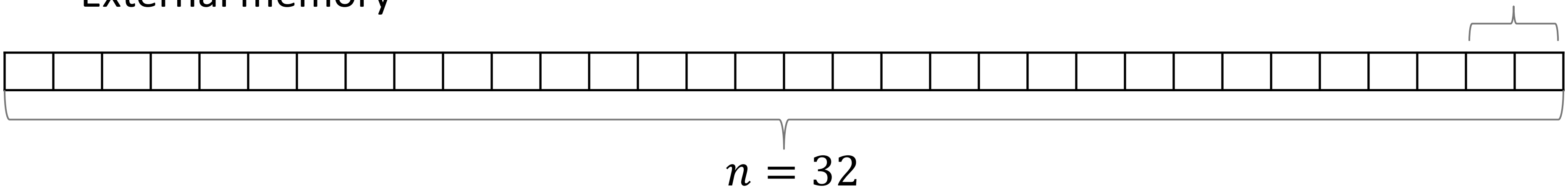
  $\Theta(n \log_d n)$ block accesses

d-Way Mergesort in External Memory

- Internal memory



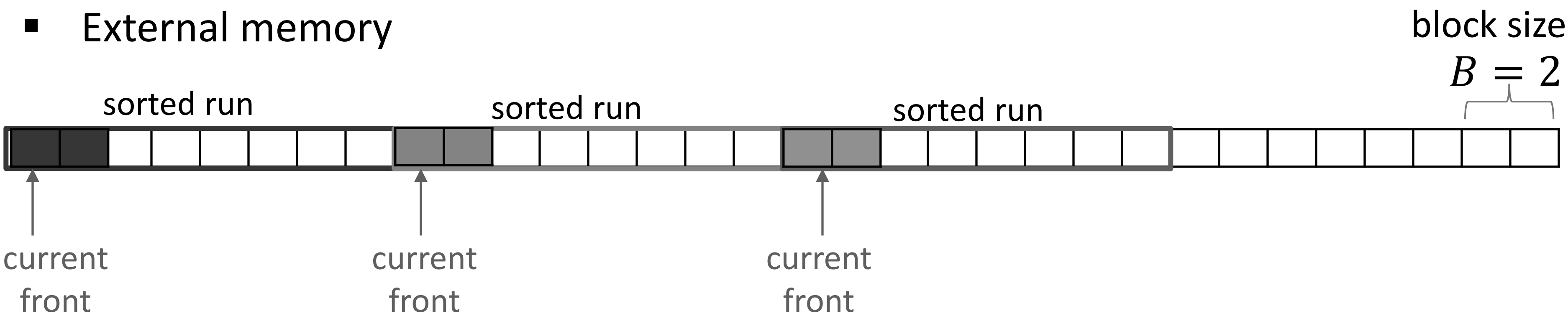
- External memory



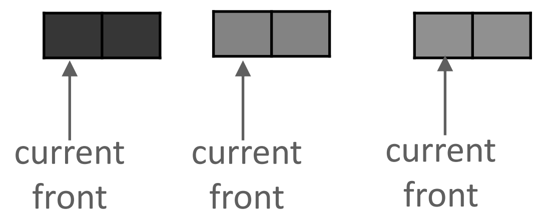
- Cannot merge in external memory directly, have to transfer to internal memory
  - only internal memory has access to CPU
- Algorithm is largely the same, but for maximum block access efficiency
  - make d as large as possible
    - less rounds of mergesort
  - for any transferred block, all data from that block should be used for sorting

d-Way Merge in External Memory

- External memory



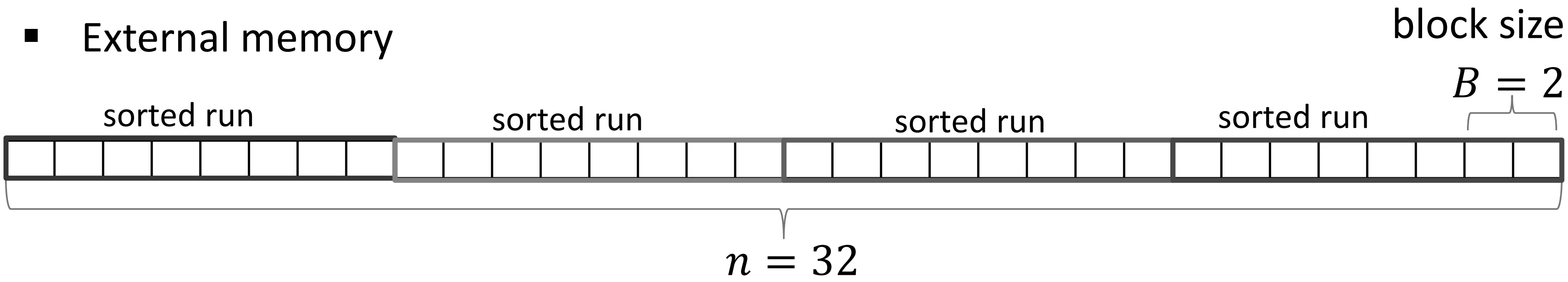
- Internal memory



- Key observation
  - do not need to transfer the full sorted run in internal memory to do d-way merge
    - at some point sorted runs will become so large that even one sorted run will not fit into the internal memory
  - enough to transfer the block that contains current front from each sorted run
    - let is call it the *active block*
  - could transfer more than one block, but transferring exactly one block lets us perform d-way merge with a larger d

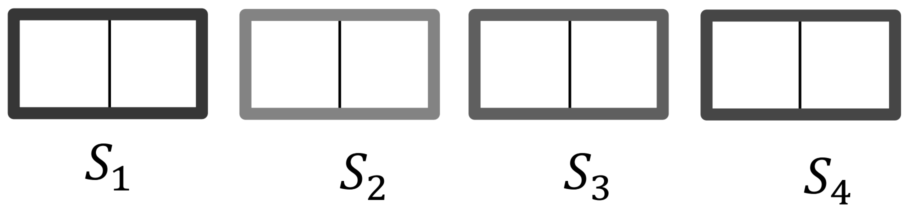
d-Way Merge in External Memory

- External memory



- Partition internal memory

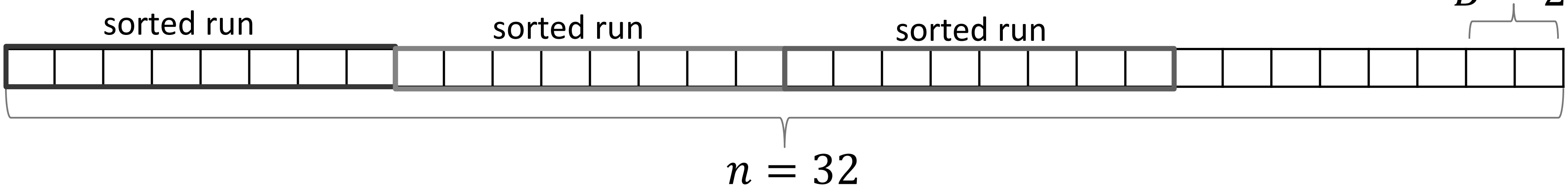
Internal (M = 8):



- In our example, looks like can perform 4-way merge ($d = 4$)
- But no, need to have some space for merged result
  - again, one block of memory is enough

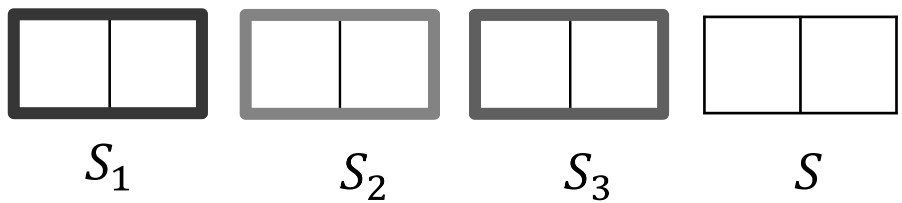
d-Way Merge in External Memory

- External memory



- Partition internal memory

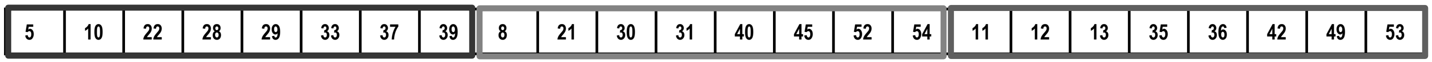
Internal (M = 8):

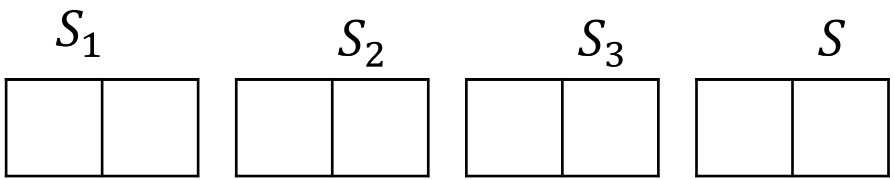


- In the example, can perform 3-way merge
- In general
  - partition in approximately $\frac{M}{B}$ sequences
  - perform $d \approx \frac{M}{B} - 1$ way merge
    - first d sequences for storing active blocks of sorted runs
    - last sequence for storing results of the merged result

d-Way merge in External Memory

- External $(B = 2)$



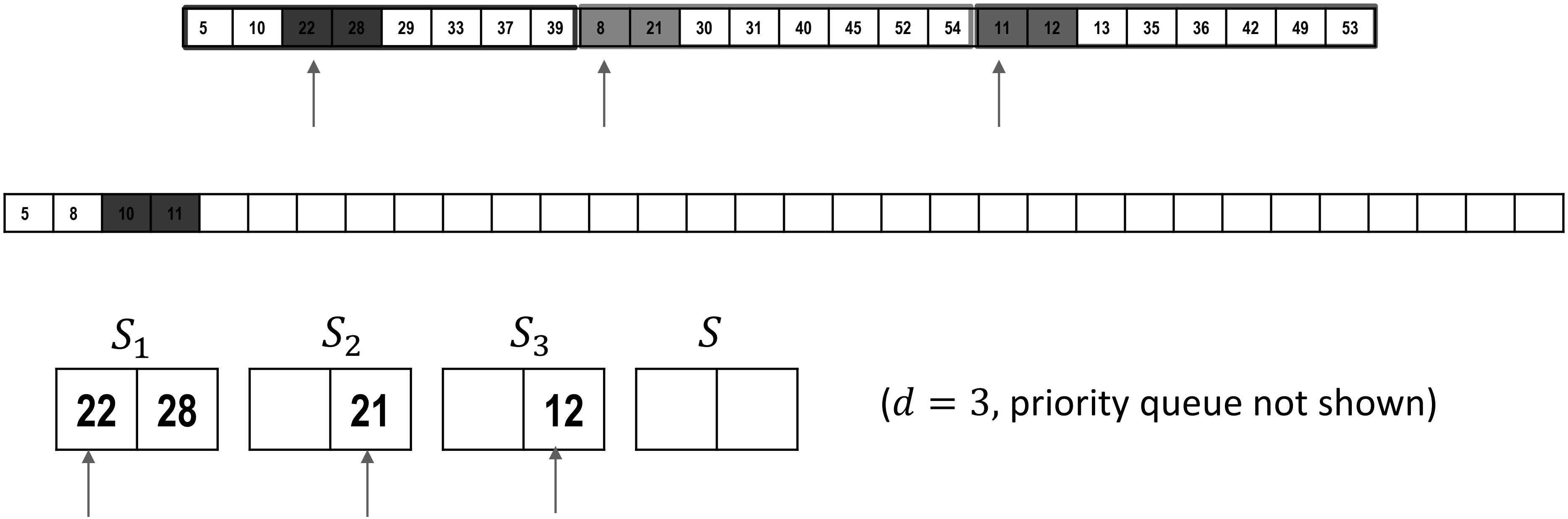


$(d = 3$, priority queue not shown)

- Example: 3-way merge
    - always bring elements from/to external memory in full blocks

d-Way merge in External Memory

- External $(B = 2)$



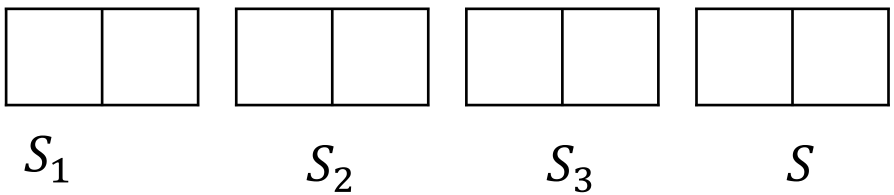
$(d = 3$, priority queue not shown)

- Example: 3-way merge
  - always bring elements from/to external memory in full blocks
  - merge in internal memory until any sequence becomes full/empty

d-Way merge in External Memory

- External $(B = 2)$

Internal (M = 8):



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

- Example: 3-way merge
  - always bring elements from/to external memory in full blocks
  - merge in internal memory until any sequence becomes full/empty
- Done with the first 3 sorted runs, continue with all other sorted runs in sets of 3
  - until all sorted runs are processed
- Total number of block transfers for one round is $\Theta(n/B)$
  - external array has size n, brought into internal memory in full blocks of size B
  - copied back to external memory in full blocks of size B

d-way Mergesort In External Memory

- $\log_d n = \frac{\log_2 n}{\log_2 d}$ rounds
- Each round makes $\Theta(n/B)$ external memory block accesses
  - with d-way merge sort, $\Theta\left(\frac{n}{B} \cdot \log_d n\right) = \Theta\left(\frac{n}{B} \cdot \frac{\log_2 n}{\log_2 d}\right)$ block accesses
    - 2-way (standard) mergesort, $\Theta\left(\frac{n}{B} \cdot \log_2 n\right)$ block accesses
    - d-way mergesort has savings factor $\log_2 d$ over 2-way mergesort
  - we made d as large as possible so that one round makes $\Theta(n/B)$ block accesses
    - n/B is the smallest number of block accesses needed to do one round of mergesort
    - if we made d any larger would need more than n/B block accesses for each round

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):

- Smart initialization can further reduce block transfers
- Mergesort starts with initial runs of size 1 and creates sorted runs of size d after one round

d	d	d	d	d	d	d	d

  - cost of one round is $\Theta(n/B)$ block transfers

- The larger the initial sorted runs are, the less rounds mergesort takes
- Can we create sorted runs of size larger than d using only $\Theta(n/B)$ of block transfers?
  - i.e. the same computational cost as the first round of mergesort

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):

- Can created sorted runs of size M using only $\Theta(n/B)$ of block transfers
  - $M > d \approx \frac{M}{B} - 1$
- Sort external memory chunks that fit into internal memory (size M chunks)

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

- Smart initialization can further reduce block transfers
- Sort external memory chunks that fit into internal memory (size M chunks)
    - copy the next chunk
    - sort in internal memory
    - copy back to external memory
- Copy, sort, copy back the rest of them

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):

- Smart initialization creates sorted runs of length M
  - $\Theta(n/B)$ block transfers
    - each chunk of size M is copied in full blocks of size B

Mergesort in External Memory: Total Cost in Block Transfers

- Initialization creates n/M sorted runs of length M
  - $\Theta(n/B)$ block transfers
- Each round increases size of a sorted run by a factor of d

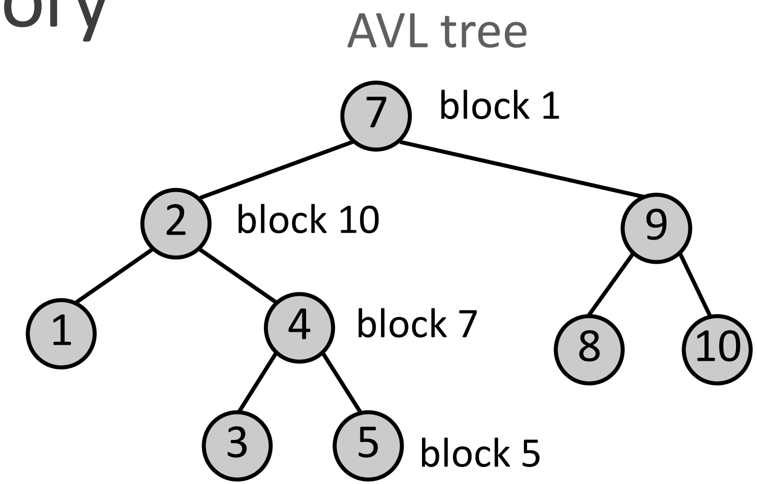
$$M \cdot \underbrace{d \cdot d \cdot \ldots \cdot d}_{d^t} = n \quad \Rightarrow \quad d^t = \frac{n}{M} \quad \Rightarrow \quad t = \log_d \frac{n}{M}$$

- At most $\log_d n/M$ rounds of merging create sorted array
  - each round $\Theta(n/B)$ block transfers
- Total number of block transfers: $O\left(\frac{n}{B} \log_d n/M\right)$
  - better than $\Theta\left(\frac{n}{B} \cdot \log_d n\right)$ without smart initialization
- Can show that d-way Mergesort with $d \approx M/B$ is optimal to minimize block transfers for sorting in external memory
  - up to constant factors

Outline

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

Dictionaries in External Memory



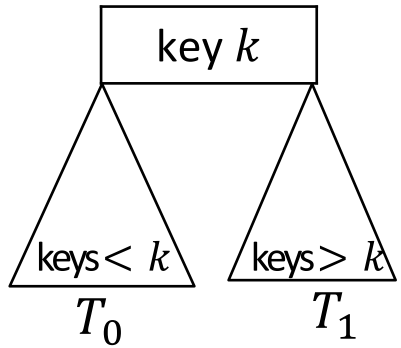
- Tree-based dictionary implementations have poor *memory locality*
  - if an operation accesses m nodes, it must access m spaced-out memory locations
- In an AVL tree, $\Theta(\log n)$ blocks are loaded in the worst case
- Better solution
  - trees that store more keys inside a node, smaller height
  - B-trees is one example
  - first consider special case of B-trees: *2-4 trees*
    - 2-4 trees also used for dictionaries in internal memory
      - may be even faster than AVL-trees
    - first analyze their performance in internal memory, and then (for B-trees) in external memory

Outline

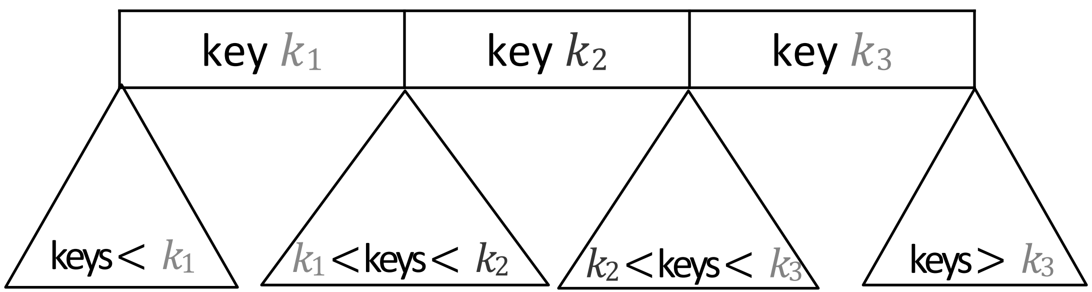
- External Memory
  - Motivation
  - External sorting
  - External Dictionaries
  - 2-4 Trees
  - (*a*, *b*)-Trees
  - B-Trees

2-4 Trees Motivation

- Binary Search tree supports efficient search with special key ordering

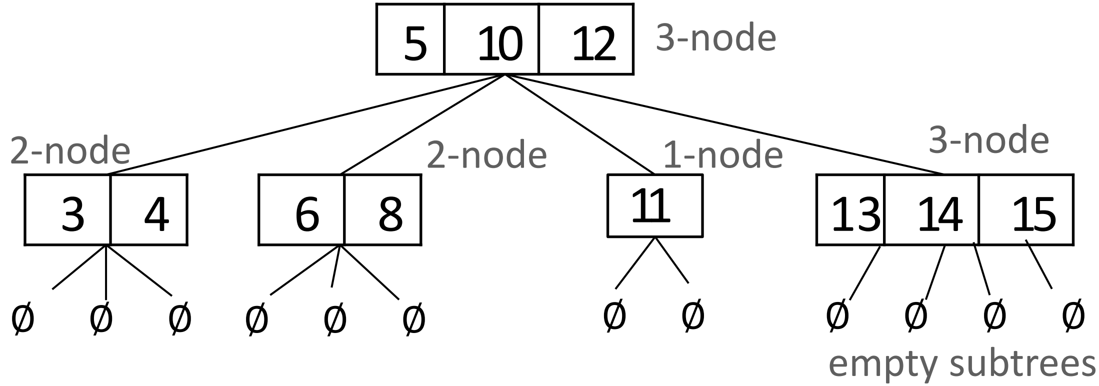


- Need nodes that store more than one key
    - how to support efficient search?

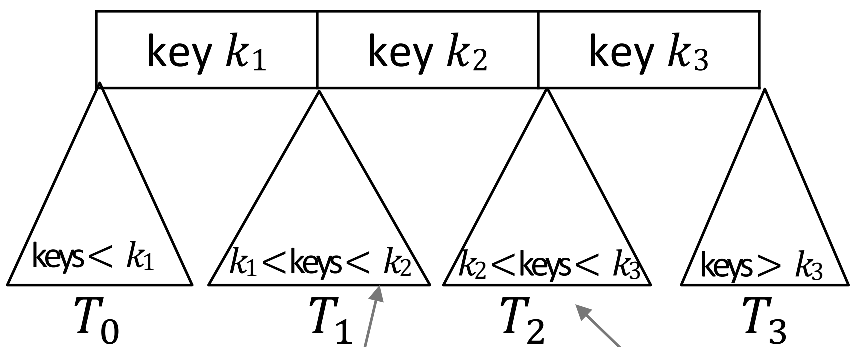


- Need more properties to ensure tree is balanced and *insert*, *delete* are efficient

2-4 Trees

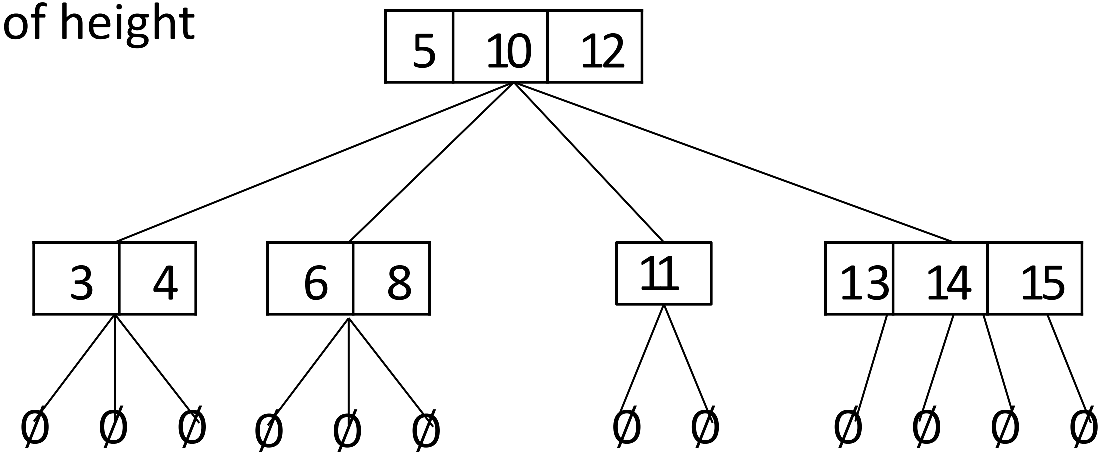


- **Structural properties**
  - Every node is either
    - 1-node: *one KVP* and *two subtrees* (possibly empty), or
    - 2-node: *two KVPs* and *three subtrees* (possibly empty), or
    - 3-node: *three KVPs* and *four subtrees* (possibly empty)
    - allowing 3 types of nodes simplifies insertion/deletion
  - 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

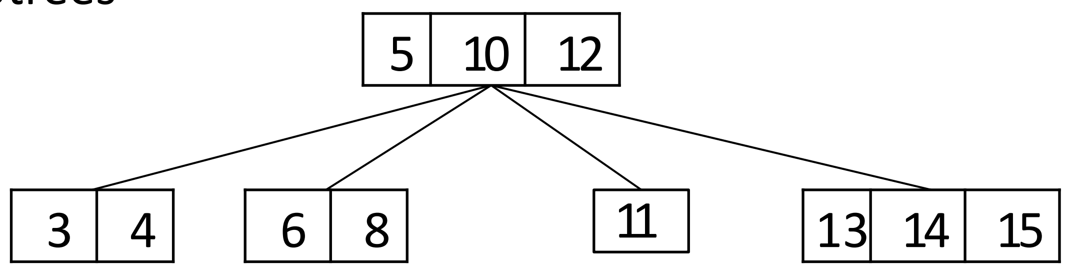


2-4 Tree Example

- Empty subtrees are not part of height computation
    - height = 1

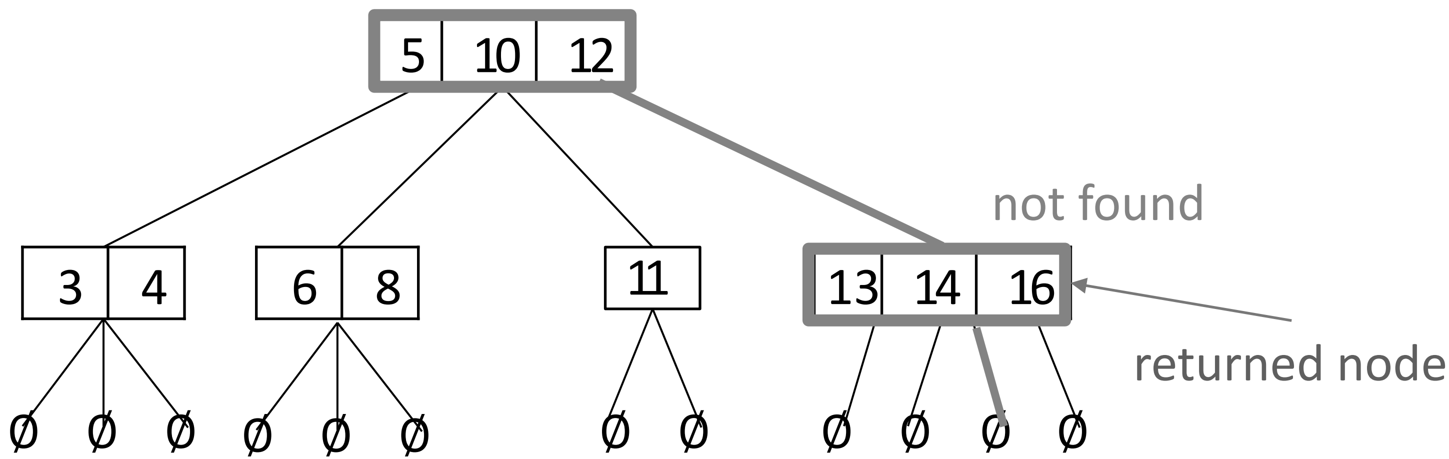


- Often do not show empty subtrees



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)



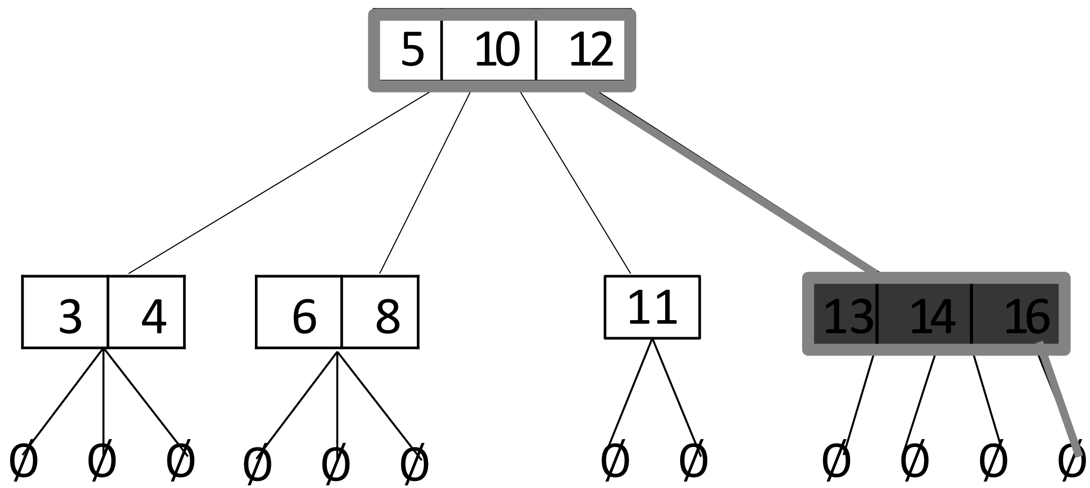
2-4 Tree operations

24TreeSearch$(k, v \leftarrow$root$, p \leftarrow$empty subtree$)$

- **if** v represents empty subtree
  - **return** "not found, would be in p"
- let $T_0, k_1, \ldots, k_d, T_d$ be keys and subtrees at v, in order
- **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** *24TreeSearch*(k, T_i, v)
- **else** *24TreeSearch*(k, T_0, v)

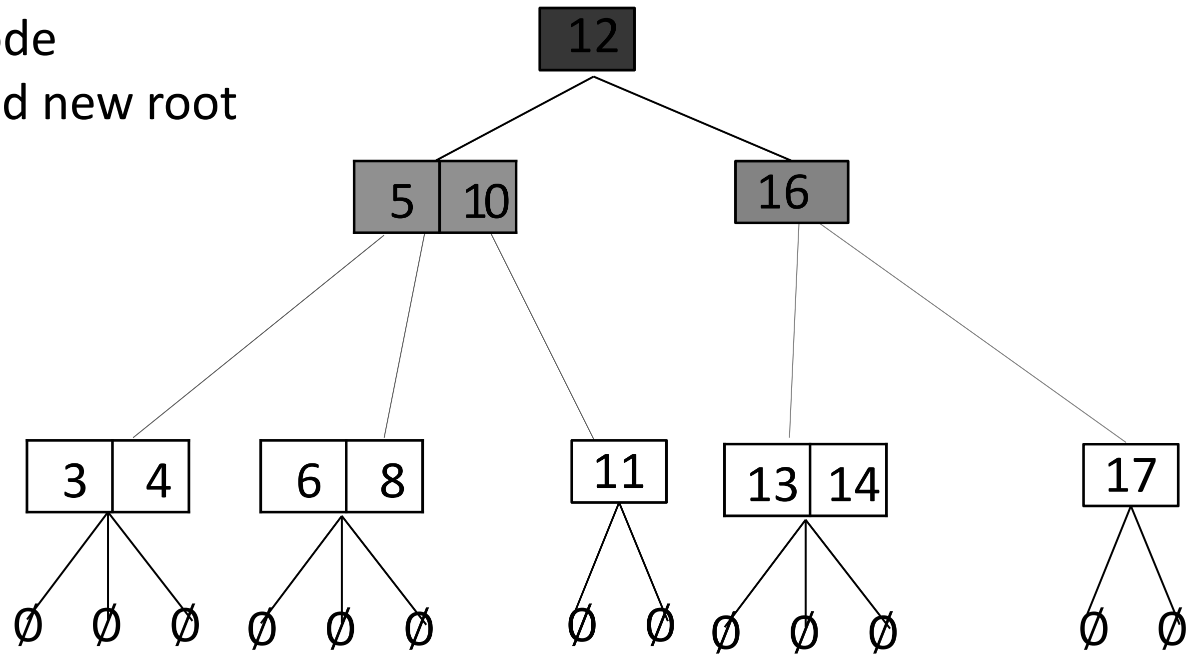
Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
  - first step is 24TreeSearch(17)



Example: 2-4 tree Insert

- **Example**: *24TreeInsert*(17)
- Split root node
    - need new root



2-4 Tree Insert Pseudocode

24TreeInsert(k)

$v \leftarrow$ *24TreeSearch*(k) //node 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 keys and subtrees at v , in order

if (v has no parent) create a parent of v (empty)

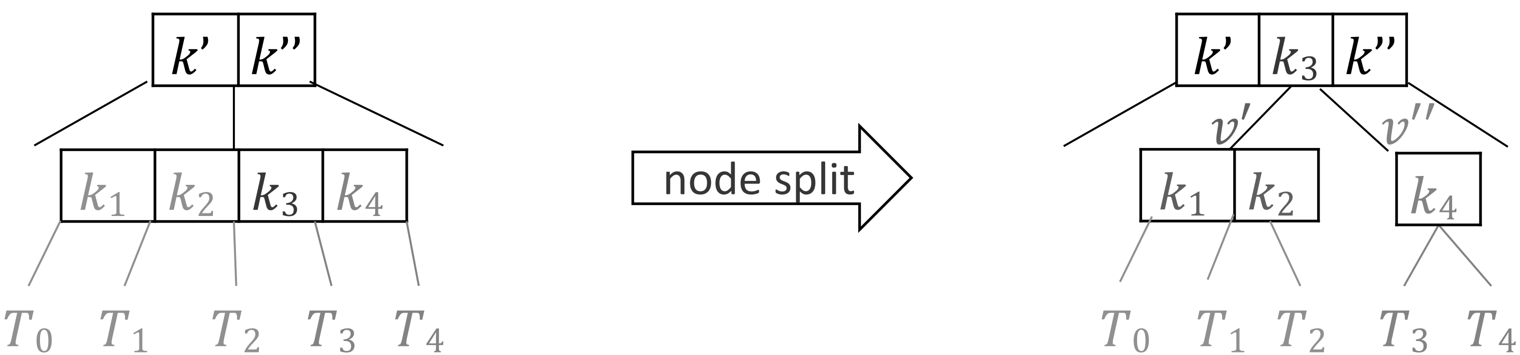
$p \leftarrow$ parent of v

$v' \leftarrow$ 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

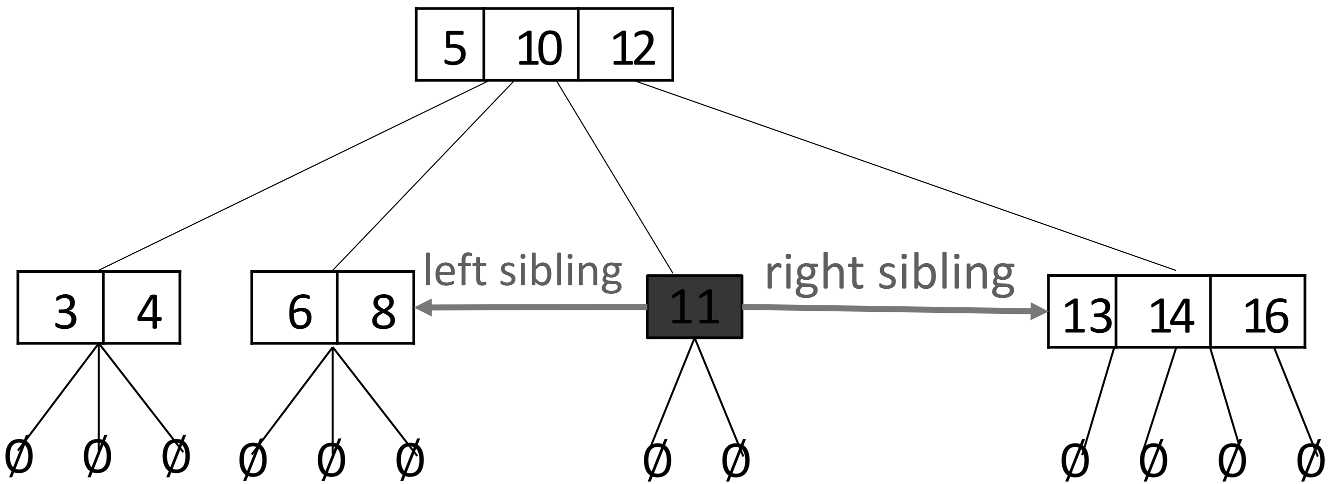
replace $< v >$ by $< v', k_3, v'' >$ in key-subtree-list of p

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

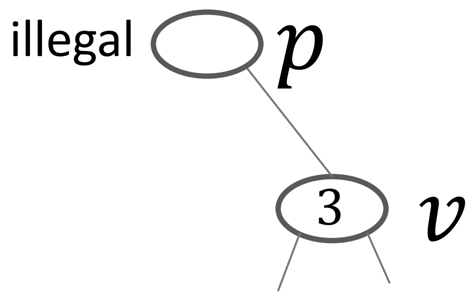


2-4 Tree: Left and Right Sibling

- Left sibling of a node is a subtree tree of the parent node which is immediately to the left
- Right sibling of a node is a subtree tree of the parent node which is immediately to the right

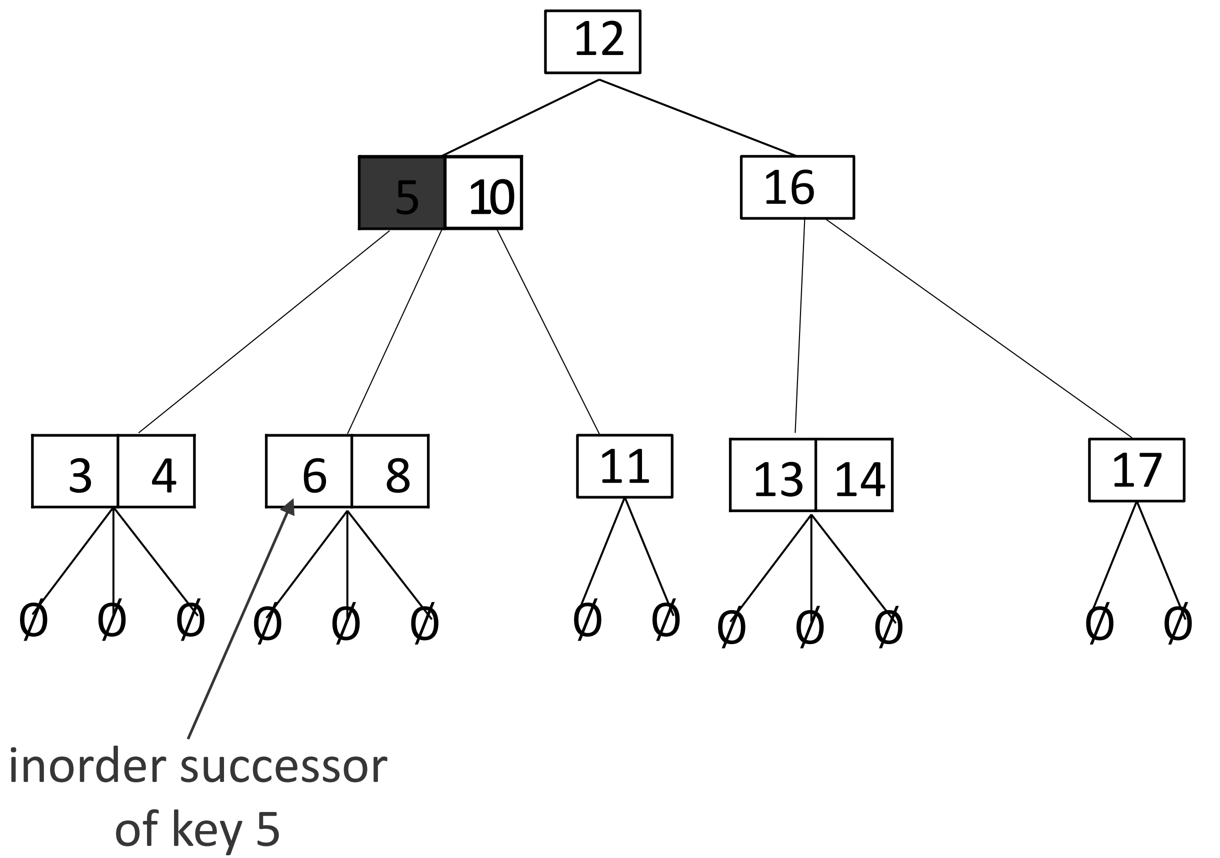


- Any node (except the root) must have a left or a right sibling (or both)



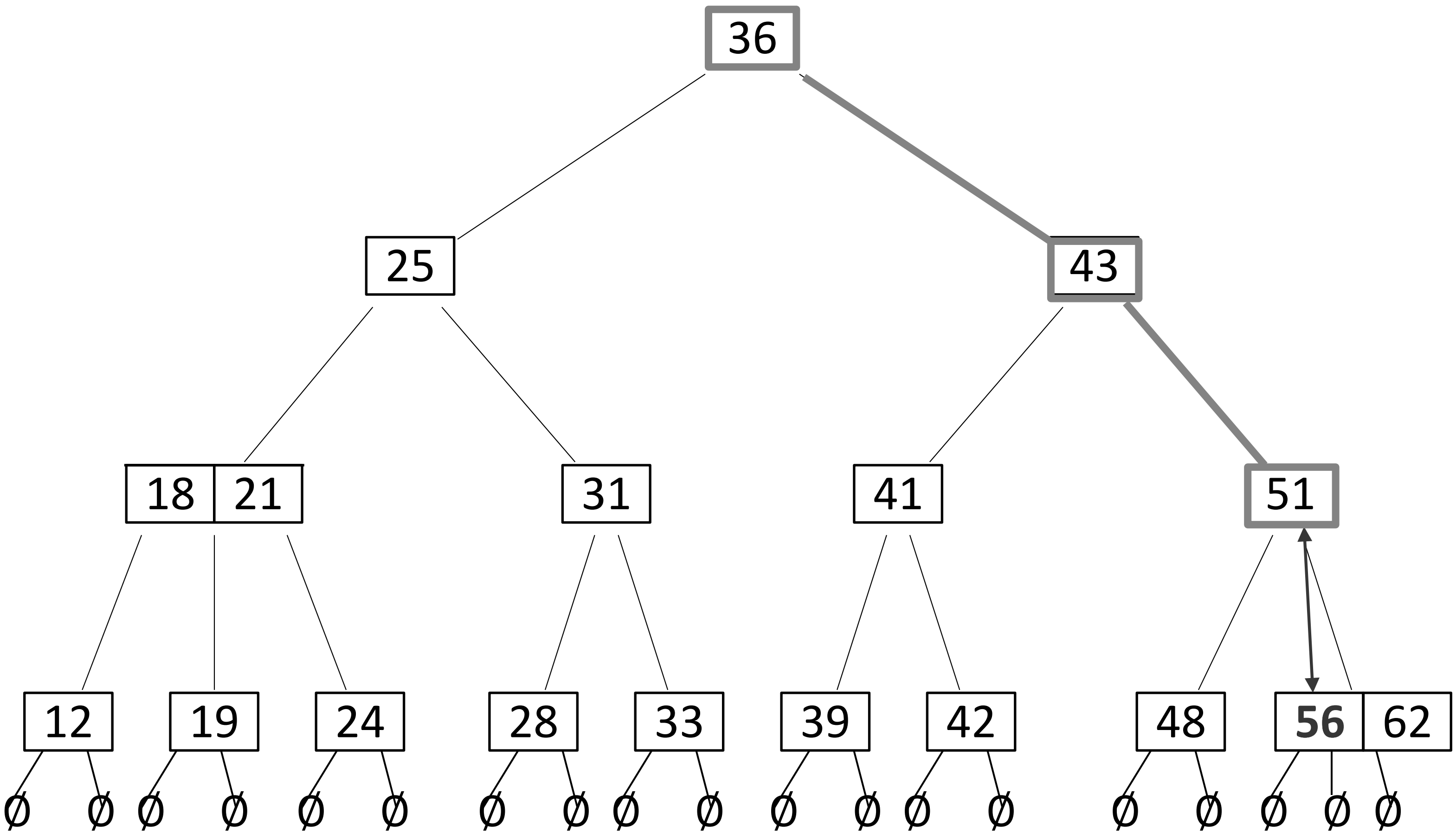
2-4 Tree: Inorder Successor

- Inorder successor of key k stored in node v is the smallest key in the subtree of v "immediately to the right" of k



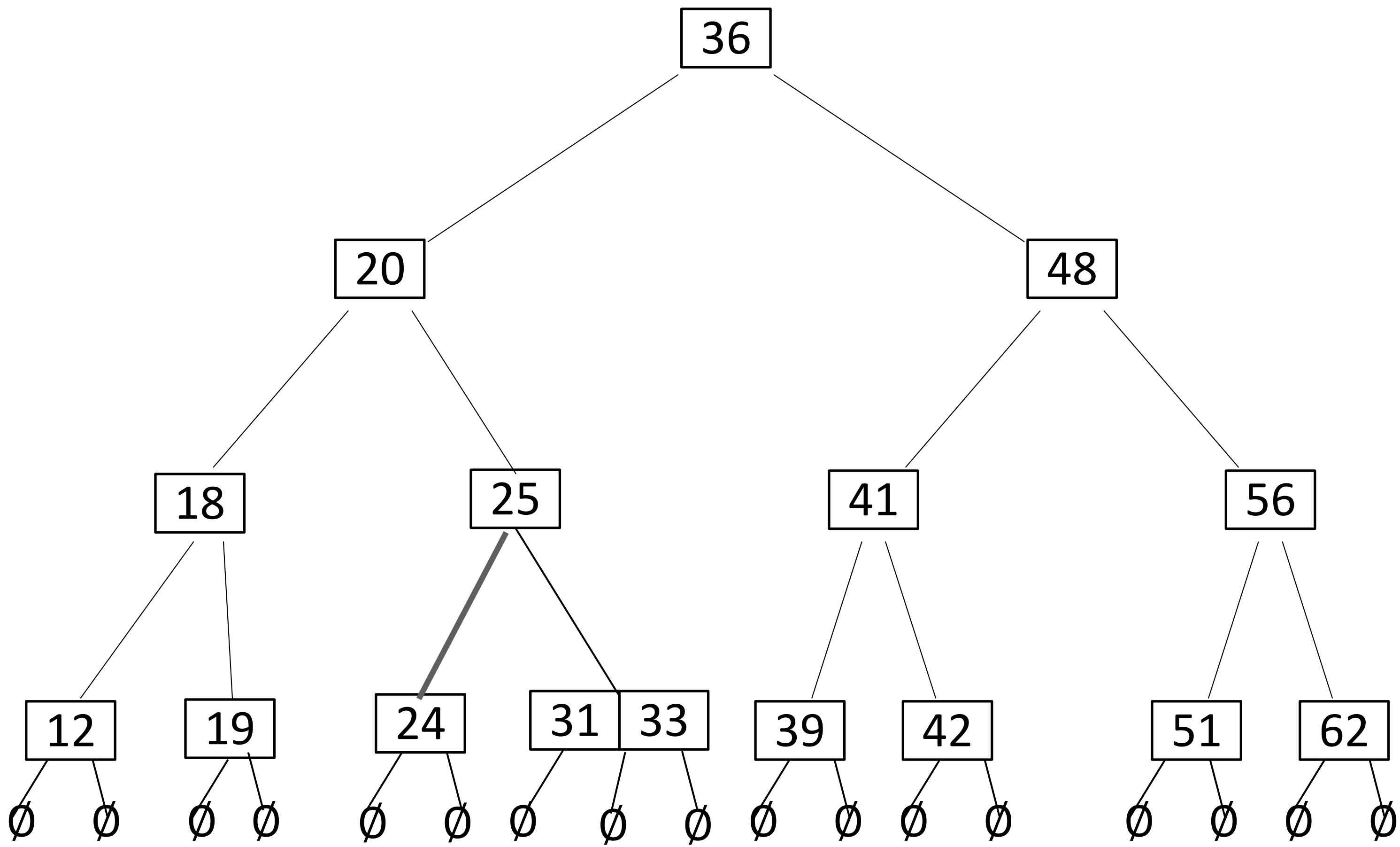
2-4 Tree Delete

- Example: *delete*(51)
- Search for key to delete
  - can delete keys only from a node with empty subtrees
  - replace key with in-order successor



2-4 Tree Delete

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



2-4 Tree Delete Summary

- If key not at a node with empty subtrees, swap with inorder successor
- Delete key and one empty subtree from node
- If underflow
    - If there is a 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 siblings have only one key, merge
        - there must be at least one sibling, unless root
            - if root, delete
        - convention: if both siblings have only one key, merge with the right sibling
        - merge may cause underflow at the parent node, continue to the parent and fix it, if necessary

Deletion from a 2-4 Tree

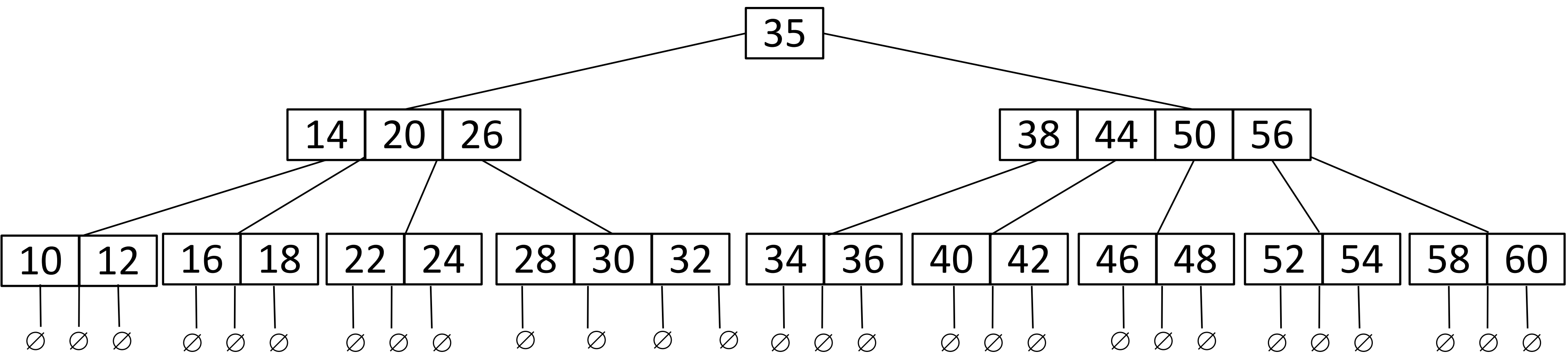
```
24TreeDelete(k)
        w ← 24TreeSearch(k)  //node containing k
        if w is not a node with only leaf children
                v ← leaf containing predecessor or successor k′ of k
                replace k by k’ in w
        delete k’ and an empty subtree in key-subtree-list of v
        while v has 0 keys // underflow
            if v is the root, delete it and break
            p ← parent of v
            if v has sibling u with 2 or more keys  // transfer/rotate
                let u be that sibling
                  if u is a right sibling // say p contains < v, k, u >
                        replace key k in p by u.k_1
                      remove < u.T_0, u.k_1 >  from u and append < k, u.T_0 > to v
                  else // symmetrical procedure if u is a left sibling
            else // merge/repeat
                if v has a right sibling
                        v’ ←new node with list (v.T_0, k, u.T_0, u.k_1, u.T_1)
                        replace < v, k, u >  by < v > in p
                        v ← p
                else ... // symmetrically with left sibling
```

Outline

- External Memory
  - Motivation
  - External sorting
  - External Dictionaries
    - 2-4 Trees
    - (a, b)-Trees
    - B-Trees

(a, b)-Trees

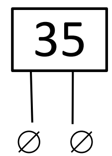
- 2-4 Tree is a specific type of (a, b)-tree
- (a, b)-tree satisfies
  - each node has at least a subtrees, unless it is the root
    - root must have at least 2 subtrees
  - each node has at most b subtrees
  - if node has k subtrees, then it stores $k - 1$ key-value pairs (KVPs)
  - all empty subtrees are at the same level
  - keys in the node are between keys in the corresponding subtrees



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

(a, b)-Trees: Root

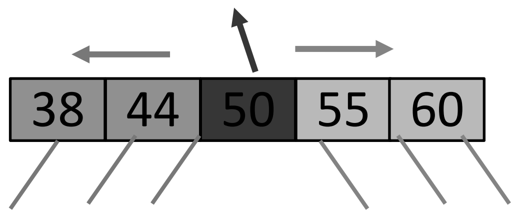
- Why special condition for the root?
- Needed for (a,b)-trees storing very few KVP
- (3,5) tree storing only 1 KVP



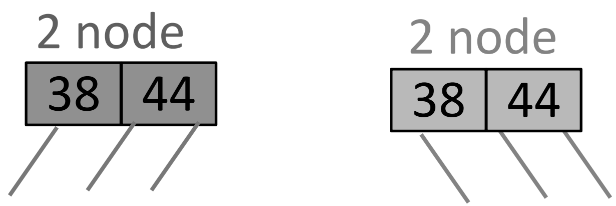
- Could not build it if forced the root to have at least 3 children
    - remember # keys at any node is one less than number of subtrees

(a, b)-Trees

- If $a \leq [b/2]$, then *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
    - each node is at least a 2-node, at most a 4-node
  - during insert, overflow if get a 5-node



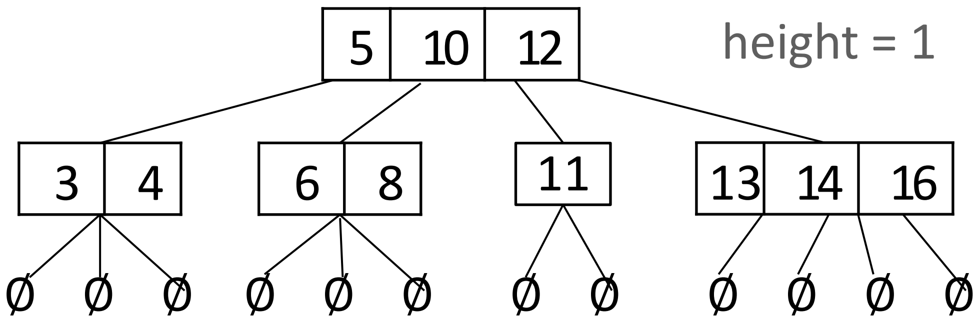
  - split results in 2-nodes, and 2-nodes are smallest allowed nodes



- If $a > [b/2]$, for example $(4,5)$-tree, cannot split like before
  - equal (best possible) split results in two 2 nodes, which is not allowed

Height of (a, b)-tree

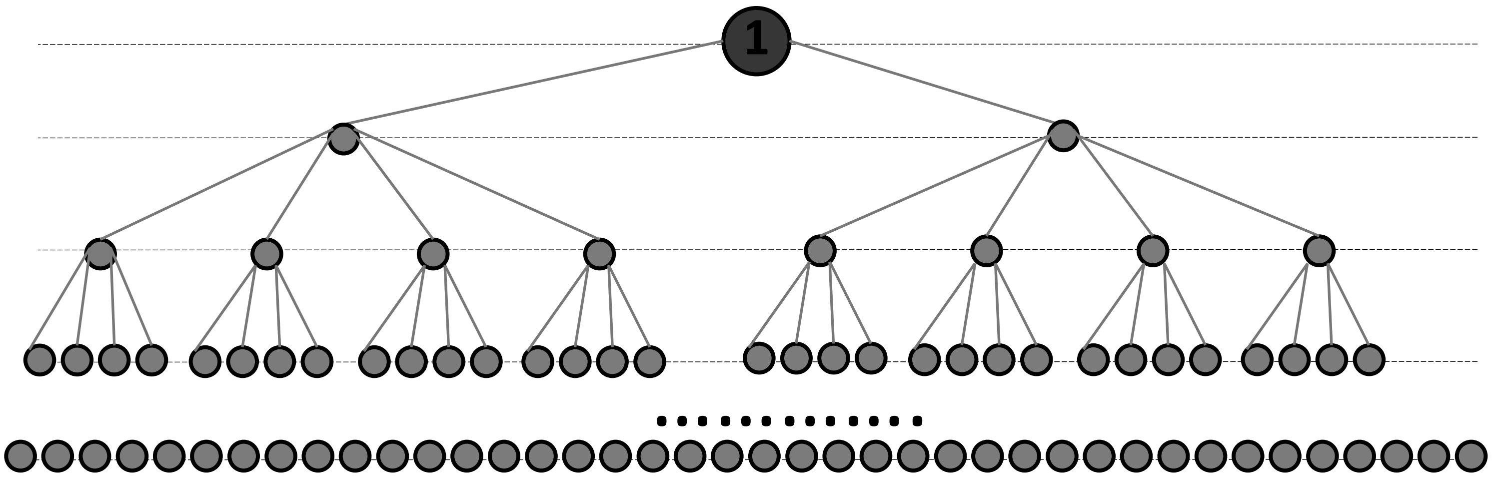
- Height = number of levels **not** counting empty subtrees



Height of (a, b)-tree

- Consider (a,b)-tree with 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	$2a^{h-1}$



$$\mathbf{1} + \sum_{i=0}^{h-1} 2a^i(a-1) = \mathbf{1} + 2(a-1)\underbrace{\sum_{i=0}^{h-1} a^i}_{\frac{a^h-1}{a-1}} = 2a^h - 1$$

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

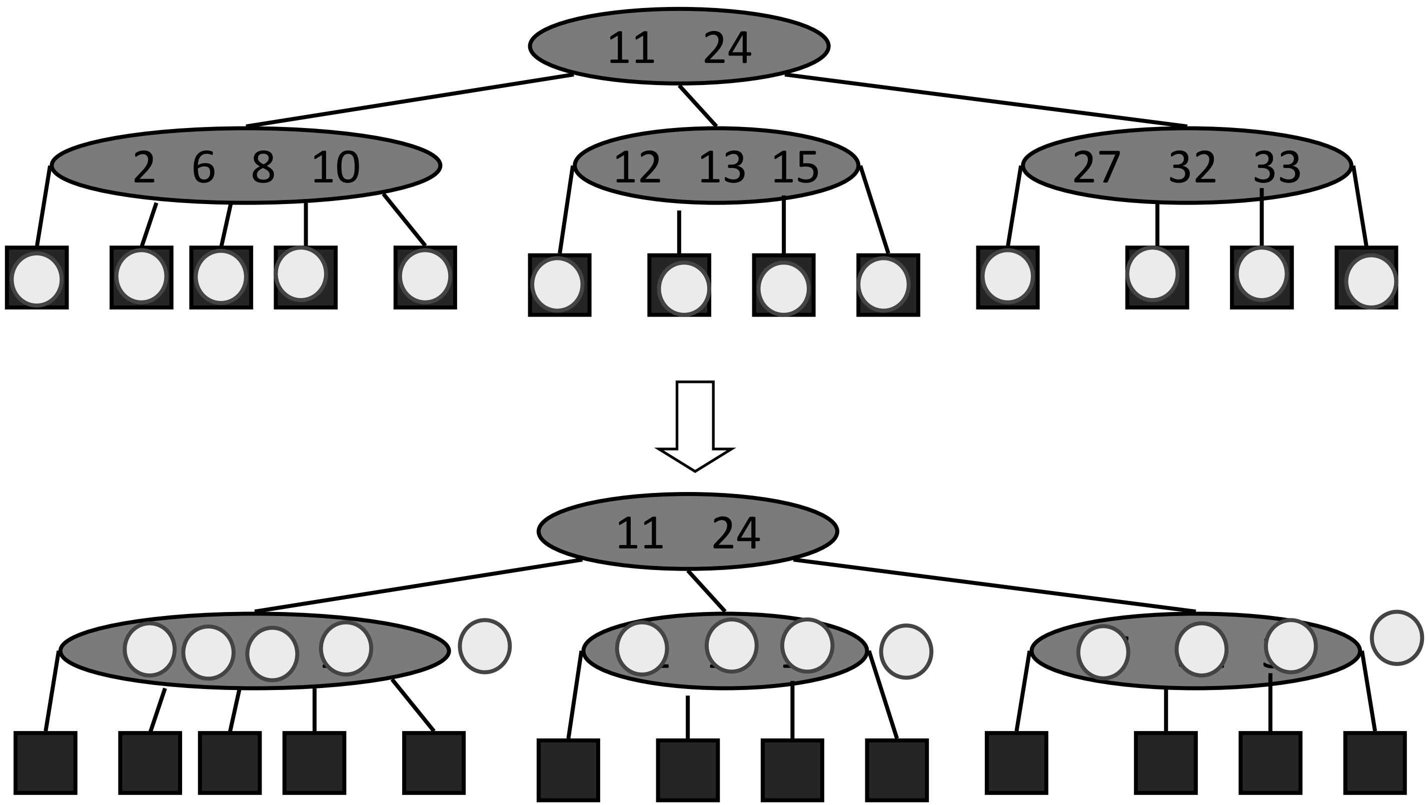
$$n \geq 2a^h - 1 \quad \text{and, therefore, } \log_a \frac{n+1}{2} \geq h$$

- Height of tree with n KVPs is $O(\log_a n)$

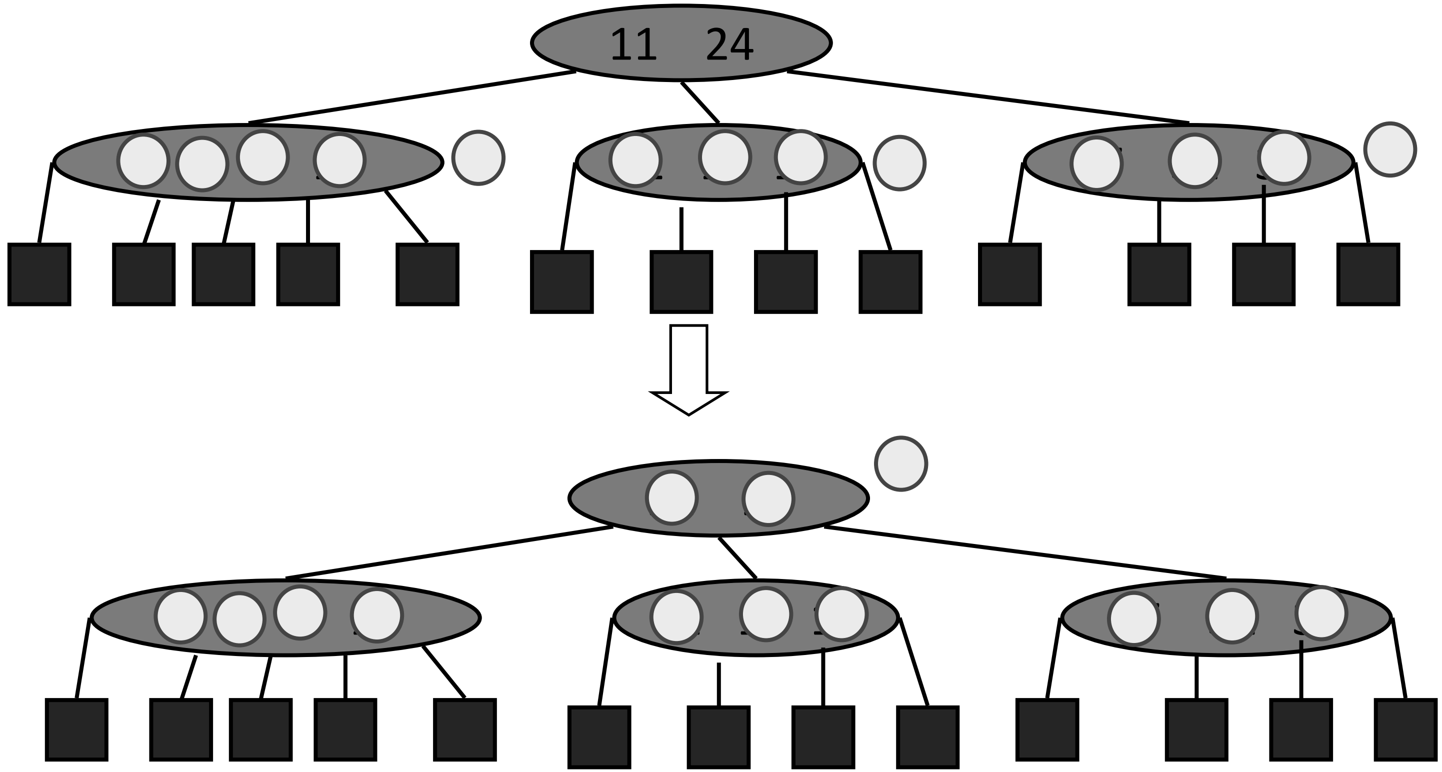
Useful Fact about (a, b)-trees

- 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

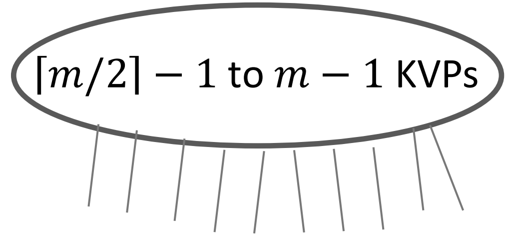


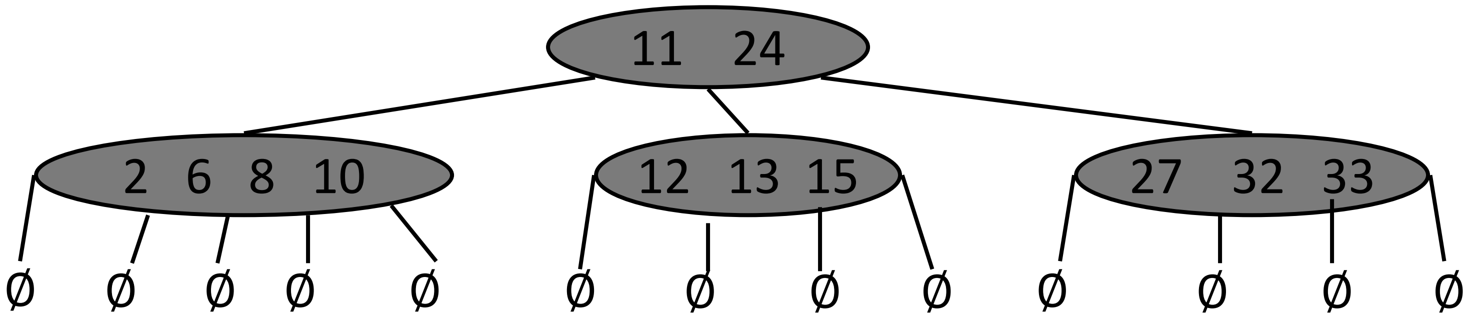
Outline

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

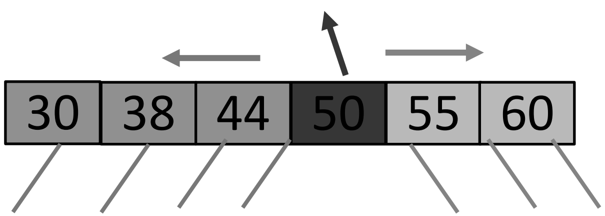
B-trees

- A *B-tree of order m* is a $(\lceil m/2 \rceil, m)$-tree
- 2-4 tree is a B-tree of order 4
  - at least 2, at most 4 subtrees
- Example: B-tree of order 6
  - at least 3, at most 6 subtrees
    - node must be at least 2-node, at most 5-node

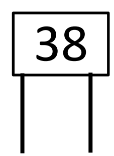




- Overflow if get a 6-node

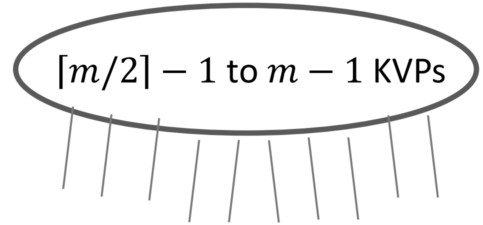


- Underflow if get a 1-node



  - transfer, if have a 3, 4 or 5-node sibling, merge if all siblings are 2-nodes

B-trees in Internal Memory



- A *B-tree of order* m is a $(\lceil m/2 \rceil, m)$-tree
  - Sedgewick uses M rather than m
- Analysis if stored in internal memory
  - each node stores its KVPs in a dictionary that supports $O(\log m)$ search, insert, and delete

5	7	9	12	14	27	29

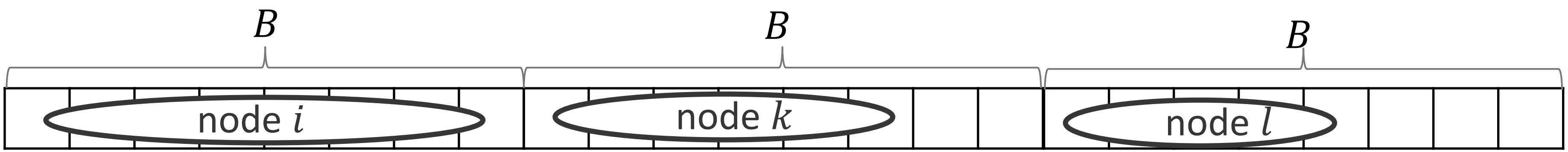
  - *search* require $\Theta(height)$ node operations
  - height is $O(\log_a n) = O\left(\frac{\log n}{\log m/2}\right) = O\left(\frac{\log n}{\log m}\right)$
  - each node operation is $O(\log m)$ time
  - total cost for each *search*

$$O\left(\frac{\log n}{\log m} \cdot \log m\right) = O(\log n)$$

  - analysis for *insert* and *delete* is the same
- No better than 2-4-trees or AVL-trees

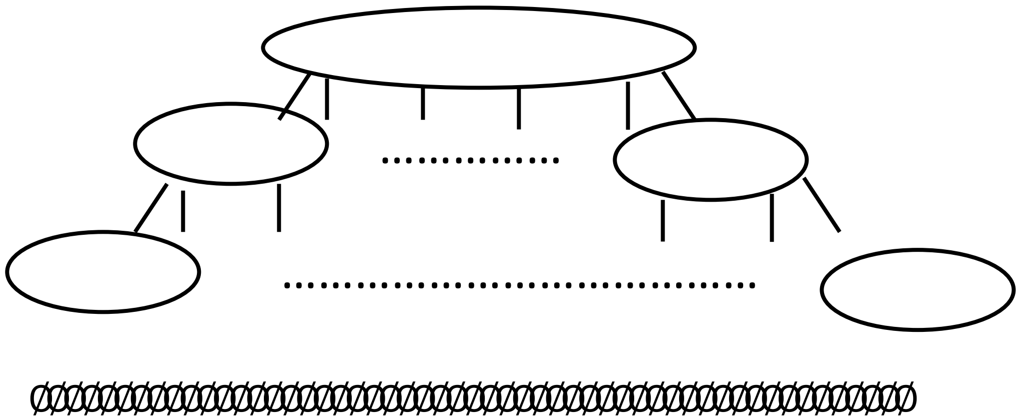
Dictionaries in External Memory

- Main applications of B-trees is to store dictionaries in external memory
- AVL tree or 2-4 tree, need to load $\Theta(\log n)$ blocks in the worst case
- Instead, use a B-tree of order m
  - m is chosen so that an m-node fits into a single block
  - typically $m \in \Theta(B)$



- Node that if m-node fills block B completely, then blocks are at least half-full
  - since each node is at least an $\lceil m/2 \rceil$-node
  - not much storage wasted
- Each operation can be done with $\Theta(height)$ block transfers
- The height of a B-tree is $\Theta(\log_m n) = \Theta(\log_B n)$
  - $\Theta(\log_B n) = \Theta\left(\frac{\log n}{\log B}\right)$
- Large savings of block transfers, $\log B$ factor compared to AVL trees

Example of B-tree usage



1 node (root)

200 nodes

200^2 nodes

200^3 empty subtrees

- B-tree of order 200
  - node fits into one block of external memory
  - B-tree of order 200 and height 2 can store up to $200^3 - 1$ KVPs
    - from the ‘useful fact’ proven before
  - if store root in internal memory, then only 2 block reads are needed to retrieve any item

B-tree variations

- For practical purposes, some variations are better
  - B-trees with **pre-emptive splitting/merging**
      - during search for insert, split *any* node close to overflow
      - during search for delete, merge *any* node close to underflow
    - can insert/delete at leaf and stop, this halves block transfers
  - **B⁺-trees**: Only leaves have KVPs, link leaves sequentially
      - interior nodes store duplicates of keys to guide search-path
      - twice as many items
    - larger m since interior nodes do not hold values
  - **Cache-oblivious** trees: What if we do not know *B*?
      - build a hierarchy of binary trees
        - each node *v* in binary tree T "hides" a binary tree T' of size $\Theta(\sqrt{n})$
    - achieves $\Theta(\log_B n)$ block transfers *without* knowing B