

CS 240 – Data Structures and Data Management

Module 11: External Memory

T. Biedl É. Schost O. Veksler

Based on lecture notes by many previous cs240 instructors

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

Winter 2021

Outline

1 External Memory

- Motivation
- Stream-based algorithms
- External sorting
- External Dictionaries
- 2-4 Trees
- a - b -Trees
- B-Trees

Outline

1 External Memory

- Motivation
- Stream-based algorithms
- External sorting
- External Dictionaries
- 2-4 Trees
- a - b -Trees
- B-Trees

Different levels of memory

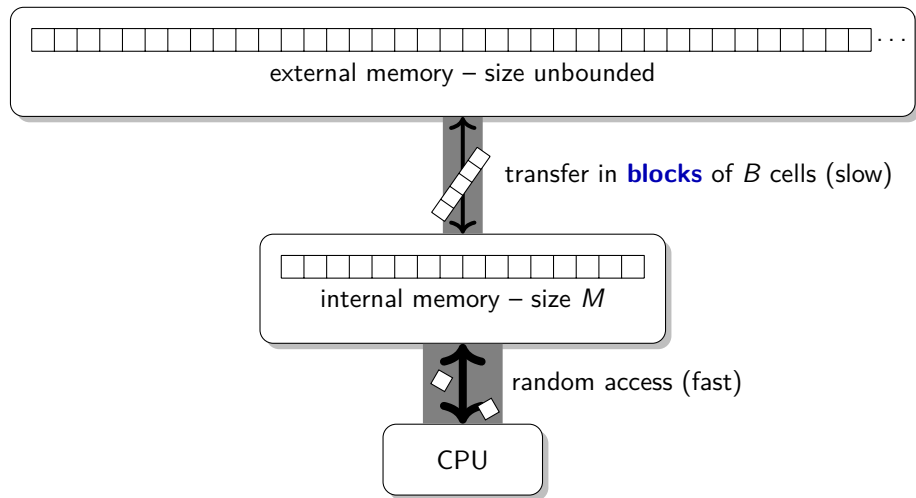
Current architectures:

- registers (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?

Observation: Accessing a single location in *external memory* (e.g. hard disk) automatically loads a whole **block** (or “page”).

The External-Memory Model (EMM)



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

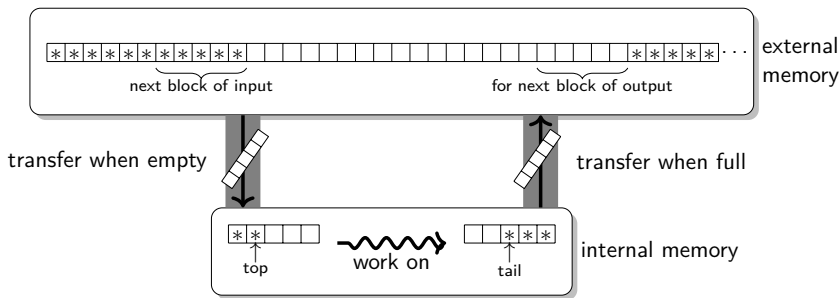
Outline

1 External Memory

- Motivation
- **Stream-based algorithms**
- External sorting
- External Dictionaries
- 2-4 Trees
- *a-b*-Trees
- B-Trees

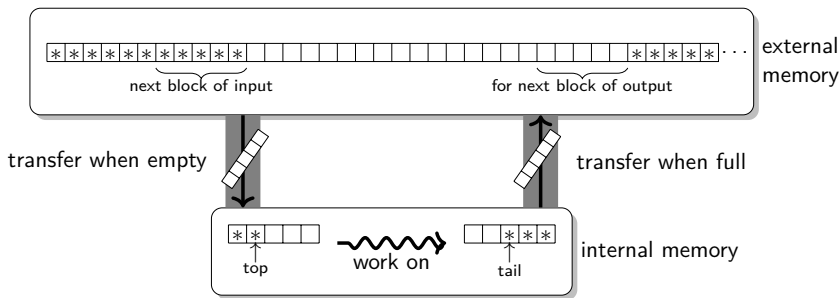
Streams and external memory

If input and output are handled via streams, then we automatically use $\Theta(\frac{n}{B})$ block transfers.



Streams and external memory

If input and output are handled via streams, then we automatically use $\Theta(\frac{n}{B})$ block transfers.



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

- Pattern matching: Karp-Rabin, Knuth-Morris-Pratt, Boyer-Moore (This assumes that pattern P fits into internal memory.)
- Text compression: Huffman, run-length encoding, Lempel-Ziv-Welch

Outline

1 External Memory

- Motivation
- Stream-based algorithms
- **External sorting**
- External Dictionaries
- 2-4 Trees
- a - b -Trees
- B-Trees

Sorting in external memory

Recall: The sorting problem:

Given an array A of n numbers, put them into sorted order.

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

- Heapsort was optimal in time and space in RAM model
- But: Heapsort accesses A at indices that are far apart
 - ↪ typically one block transfer per array access
 - ↪ typically $\Theta(n \log n)$ block transfers.

Can we do better?

Sorting in external memory

Recall: The sorting problem:

Given an array A of n numbers, put them into sorted order.

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

- Heapsort was optimal in time and space in RAM model
- But: Heapsort accesses A at indices that are far apart
 - ↪ typically one block transfer per array access
 - ↪ typically $\Theta(n \log n)$ block transfers.

Can we do better?

- Mergesort adapts well to external memory. Recall algorithm:
 - ▶ Split input in half
 - ▶ Sort each half recursively \rightarrow two sorted parts
 - ▶ Merge sorted parts.

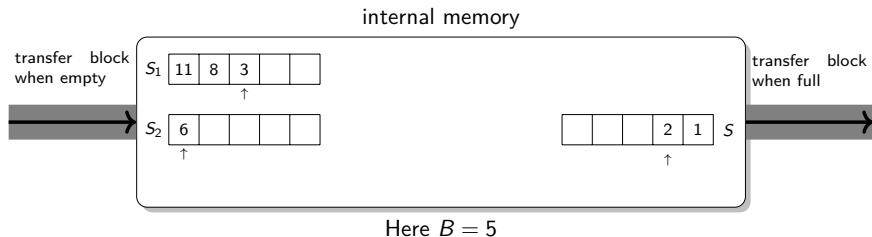
Key idea: Merge can be done with streams.

Merge

Merge(S_1, S_2, S)

S_1, S_2 : input streams that are 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** $S.append(S_2.pop())$



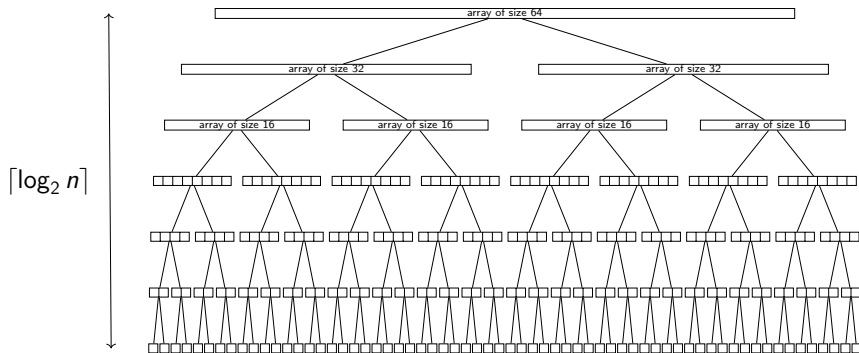
Mergesort in external memory

- *Merge* uses streams S_1, S_2, S .
⇒ Each block in the stream only transferred once.
 - So *Merge* takes $\Theta(\frac{m}{B})$ block-transfers to merge m elements
 - Recall: Mergesort uses $\lceil \log_2 n \rceil$ rounds of merging, each round merges n elements
- ⇒ Mergesort uses $O(\frac{n}{B} \cdot \log_2 n)$ block-transfers.

Not bad, but we can do better.

Towards d -way Mergesort

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

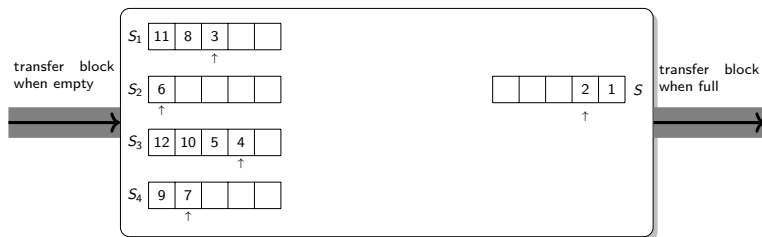


Towards d -way Mergesort

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.
- Here $d \approx \frac{M}{B} - 1$ so that $d+1$ blocks fit into internal memory.

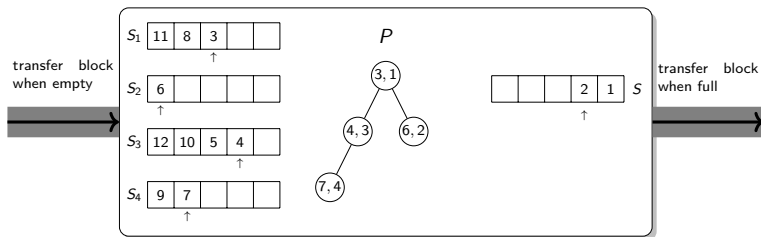


d-way merge

d-way-merge(S_1, \dots, S_d, S)

S_1, \dots, S_d : input streams that are 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-stream
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**
7. $P.insert((S_i.top(), i))$



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 $O(n \log d)$.
- 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 main memory.
- The number of *block transfers* then is again $O(\frac{n}{B})$.

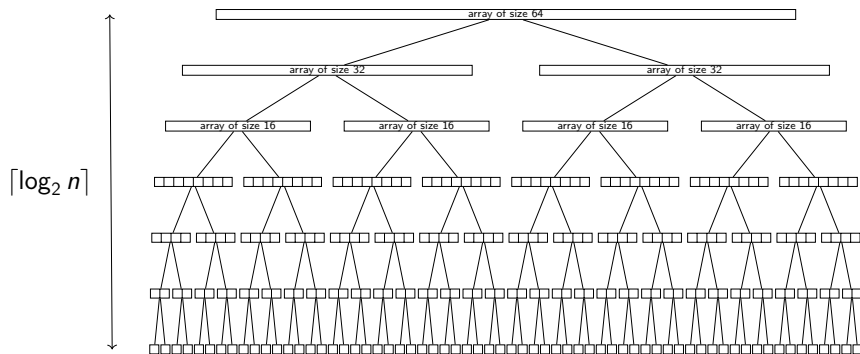
d-way merge

- 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 main memory.
- The number of *block transfers* then is again $O(\frac{n}{B})$.

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

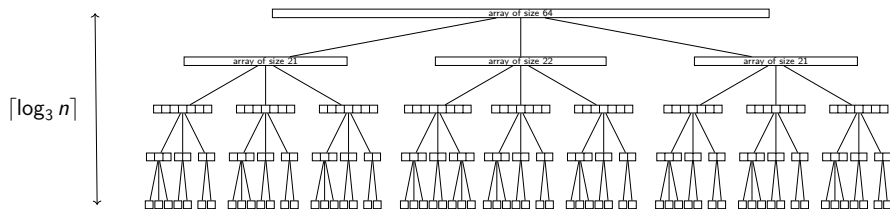
Towards d -way Mergesort

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



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 $\Theta\left(\frac{n}{B}\right)$ 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 immediately with runs of length M .

d-way mergesort

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	43	2	17	6	46	23	20	1	24	7	18	47	26	16	48	50
----	---	----	----	----	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	----	----	---	----	---	----	----	----	---	----	---	----	----	----	---	----	---	----	----	----	----	----	----

Internal ($M = 8$):

--	--	--	--	--	--	--	--

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

d-way mergesort

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	43	2	17	6	46	23	20	1	24	7	18	47	26	16	48	50
----	---	----	----	----	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	----	----	---	----	---	----	----	----	---	----	---	----	----	----	---	----	---	----	----	----	----	----	----

Internal ($M = 8$):

39	5	28	22	10	33	29	37
----	---	----	----	----	----	----	----

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

d-way mergesort

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	43	2	17	6	46	23	20	1	24	7	18	47	26	16	48	50
----	---	----	----	----	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	----	----	---	----	---	----	----	----	---	----	---	----	----	----	---	----	---	----	----	----	----	----	----

Internal ($M = 8$):

5	10	22	28	29	33	37	39
---	----	----	----	----	----	----	----

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

d-way mergesort

External ($B = 2$):

5	10	22	28	29	33	37	39	8	30	54	40	31	52	21	45	35	11	42	53	13	12	49	36	4	14	27	9	44	3	32	15	43	2	17	6	46	23	20	1	24	7	18	47	26	16	48	50
---	----	----	----	----	----	----	----	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	----	----	---	----	---	----	----	----	---	----	---	----	----	----	---	----	---	----	----	----	----	----	----

← sorted run →

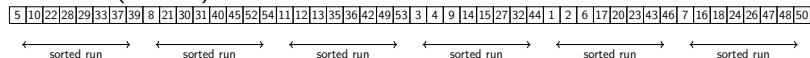
Internal ($M = 8$):

--	--	--	--	--	--	--	--

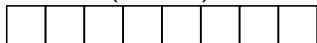
- 1 Create $\frac{n}{M}$ sorted runs of length M .

d-way mergesort

External ($B = 2$):



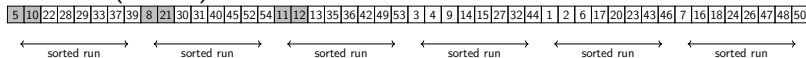
Internal ($M = 8$):



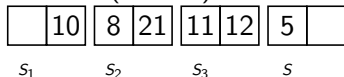
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

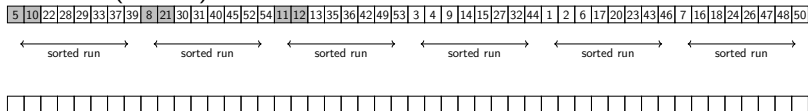


(priority queue not shown)

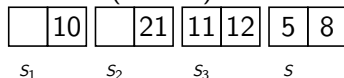
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

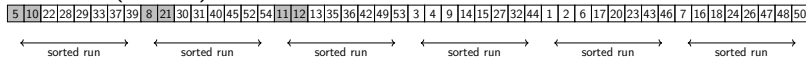


(priority queue not shown)

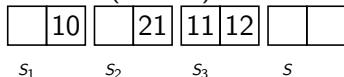
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

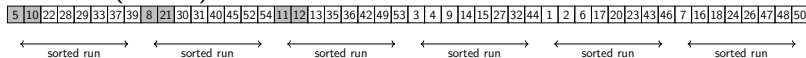


(priority queue not shown)

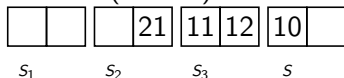
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

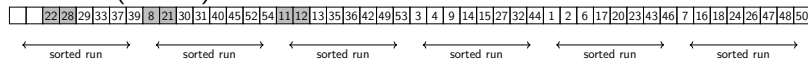


(priority queue not shown)

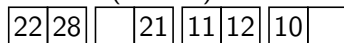
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):



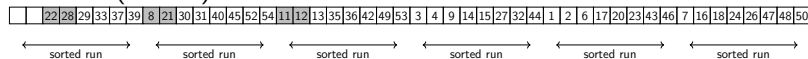
(priority queue not shown)

S_1 S_2 S_3 S

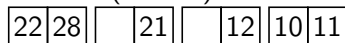
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):



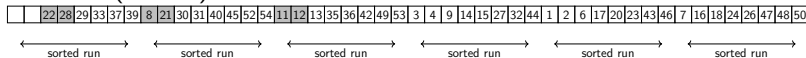
(priority queue not shown)

s_1 s_2 s_3 s

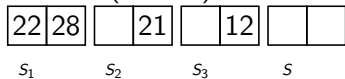
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

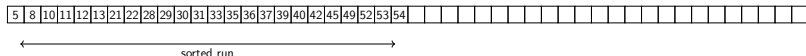
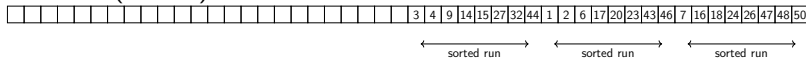


(priority queue not shown)

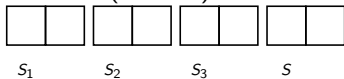
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

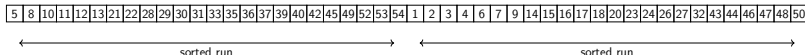


(priority queue not shown)

- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):

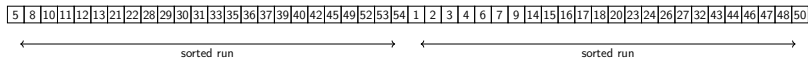
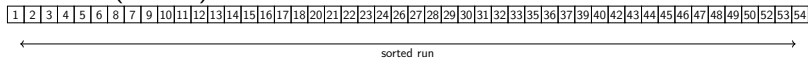


(priority queue not shown)

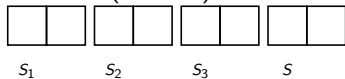
- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*
- 3 Keep merging the next runs to reduce # runs by factor of d
 \rightsquigarrow one round of merging. $\Theta(\frac{n}{B})$ block transfers

d-way mergesort

External ($B = 2$):



Internal ($M = 8$):



(priority queue not shown)

- 1 Create $\frac{n}{M}$ sorted runs of length M . $\Theta(\frac{n}{B})$ block transfers
- 2 Merge the first $d \approx \frac{M}{B} - 1$ sorted runs using *d-Way-Merge*
- 3 Keep merging the next runs to reduce $\#$ runs by factor of d
 \rightsquigarrow one round of merging. $\Theta(\frac{n}{B})$ block transfers
- 4 Keep doing rounds until only one run is left

d-way mergesort

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

d-way mergesort

- We have $\log_d\left(\frac{n}{M}\right)$ rounds of merging:
 - ▶ $\frac{n}{M}$ runs after initialization
 - ▶ $\frac{n}{M}/d$ runs after one round.
 - ▶ $\frac{n}{M}/d^k$ runs after k rounds $\Rightarrow k \leq \log_d\left(\frac{n}{M}\right)$.
- We have $O\left(\frac{n}{B}\right)$ block-transfers per round.
- $d \approx \frac{M}{B} - 1$.

\Rightarrow Total # block transfers is proportional to

$$\log_d\left(\frac{n}{M}\right) \cdot \frac{n}{B} \in O\left(\log_{M/B}\left(\frac{n}{M}\right) \cdot \frac{n}{B}\right)$$

d-way mergesort

- We have $\log_d\left(\frac{n}{M}\right)$ rounds of merging:
 - ▶ $\frac{n}{M}$ runs after initialization
 - ▶ $\frac{n}{M}/d$ runs after one round.
 - ▶ $\frac{n}{M}/d^k$ runs after k rounds $\Rightarrow k \leq \log_d\left(\frac{n}{M}\right)$.
- We have $O\left(\frac{n}{B}\right)$ block-transfers per round.
- $d \approx \frac{M}{B} - 1$.

\Rightarrow Total # block transfers is proportional to

$$\log_d\left(\frac{n}{M}\right) \cdot \frac{n}{B} \in O\left(\log_{M/B}\left(\frac{n}{M}\right) \cdot \frac{n}{B}\right)$$

One can prove lower bounds in the external memory model:

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

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

d-way mergesort

- We have $\log_d(\frac{n}{M})$ rounds of merging:
 - ▶ $\frac{n}{M}$ runs after initialization
 - ▶ $\frac{n}{M}/d$ runs after one round.
 - ▶ $\frac{n}{M}/d^k$ runs after k rounds $\Rightarrow k \leq \log_d(\frac{n}{M})$.
- We have $O(\frac{n}{B})$ block-transfers per round.
- $d \approx \frac{M}{B} - 1$.

\Rightarrow Total # block transfers is proportional to

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

One can prove lower bounds in the external memory model:

We **require** $\Omega(\log_{M/B}(\frac{n}{M}) \cdot \frac{n}{B})$ 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

1 External Memory

- Motivation
- Stream-based algorithms
- External sorting
- **External Dictionaries**
- 2-4 Trees
- *a-b*-Trees
- B-Trees

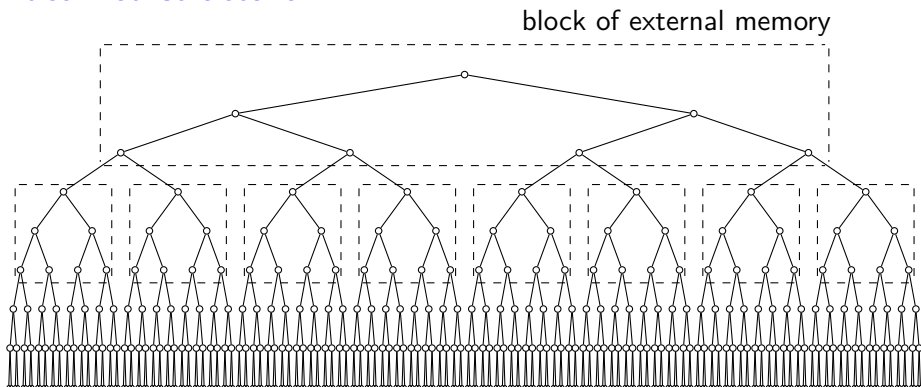
Dictionaries in external memory

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.

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

Idealized structure



Idea: Store subtrees in one block of memory.

- If block can hold subtree of size $b-1$, then block covers height $\log b$
- ⇒ Search-path hits $\frac{\Theta(\log n)}{\log b}$ blocks ⇒ $\Theta(\log_b n)$ block-transfers
- Block acts as one node of a *multiway-tree* ($b-1$ KVPs, b subtrees)

Towards B -trees

- **Idea:** Define *multiway-tree*
 - ▶ One node stores many KVPs
 - ▶ Always true: $b-1$ KVPs $\Leftrightarrow b$ subtrees
- To allow *insert/delete*, we permit varying numbers of KVPs in nodes
- This gives much smaller height than for AVL-trees
 \Rightarrow fewer block transfers
- Study first one special case: *2-4-trees*
 - ▶ Also useful for dictionaries in internal memory
 - ▶ May be faster than AVL-trees even in internal memory

Outline

1 External Memory

- Motivation
- Stream-based algorithms
- External sorting
- External Dictionaries
- **2-4 Trees**
- *a-b*-Trees
- B-Trees

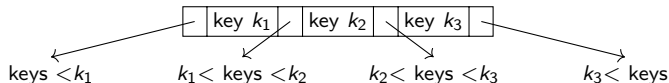
2-4 Trees

Structural property: 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).

Order property: The keys at a node are between the keys in the subtrees.

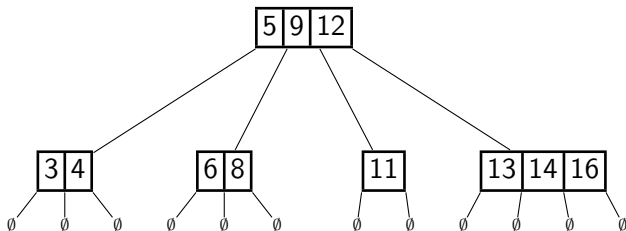
- With this, search is much like in binary search trees.



Another structural property: All empty subtrees are at the same level.

- This is important to ensure small height.

2-4 Tree example

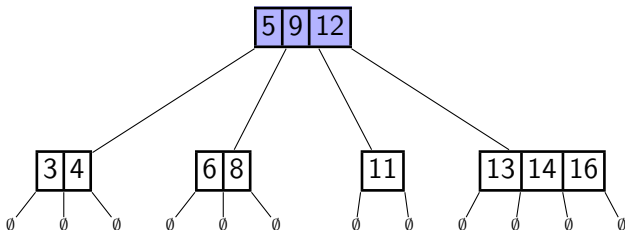


- Empty trees do not count towards height
 - ▶ This tree has height 1
- Easy to show: Height is in $O(\log n)$, where $n = \#$ KVPs.
 - ▶ Layer i has at least 2^i nodes for $i = 0, \dots, h$
 - ▶ Each node has at least one KVP.

2-4 Tree Operations

- Search is similar to BST:
 - ▶ Compare search-key to keys at node
 - ▶ If not found, recurse in appropriate subtree

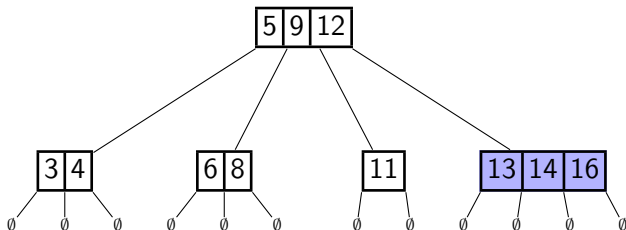
Example: *search*(15)



2-4 Tree Operations

- Search is similar to BST:
 - ▶ Compare search-key to keys at node
 - ▶ If not found, recurse in appropriate subtree

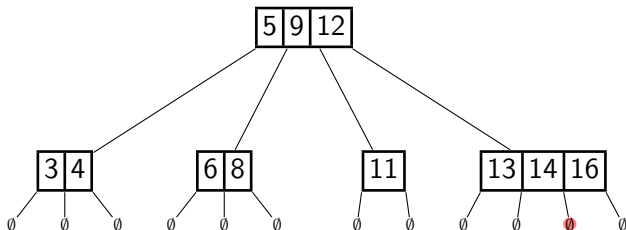
Example: *search*(15)



2-4 Tree Operations

- Search is similar to BST:
 - ▶ Compare search-key to keys at node
 - ▶ If not found, recurse in appropriate subtree

Example: *search(15) not found*



2-4 Tree operations

24Tree::search($k, v \leftarrow \text{root}, p \leftarrow \text{NIL}$)

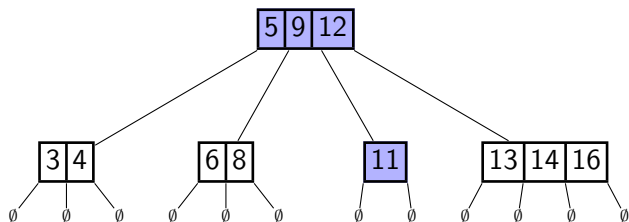
k : key to search, v : node where we search, p : parent of v

1. **if** v represents empty subtree
2. **return** “not found, would be in p ”
3. Let $\langle T_0, k_1, \dots, k_d, T_d \rangle$ be key-subtree list at v
4. **if** $k \geq k_1$
5. $i \leftarrow$ maximal index such that $k_i \leq k$
6. **if** $k_i = k$
7. **return** key-value pair at k_i
8. **else** *24Tree::search*(k, T_i, v)
9. **else** *24Tree::search*(k, T_0, v)

Insertion in a 2-4 tree

Example: *insert*(10)

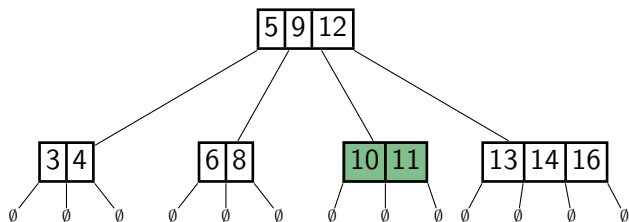
- Do *24Tree::search* and add key and empty subtree at leaf.



Insertion in a 2-4 tree

Example: *insert*(10)

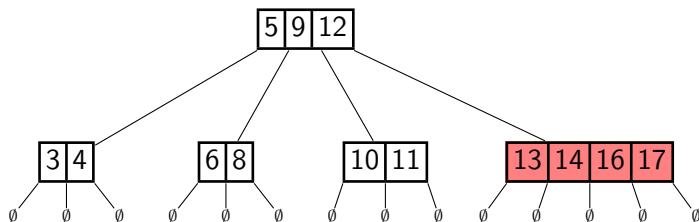
- Do *24Tree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.



Insertion in a 2-4 tree

Example: *insert*(17)

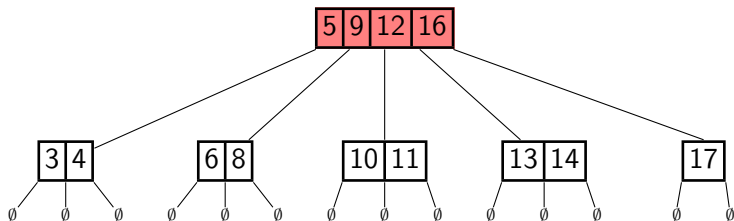
- Do *24Tree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.



Insertion in a 2-4 tree

Example: *insert*(17)

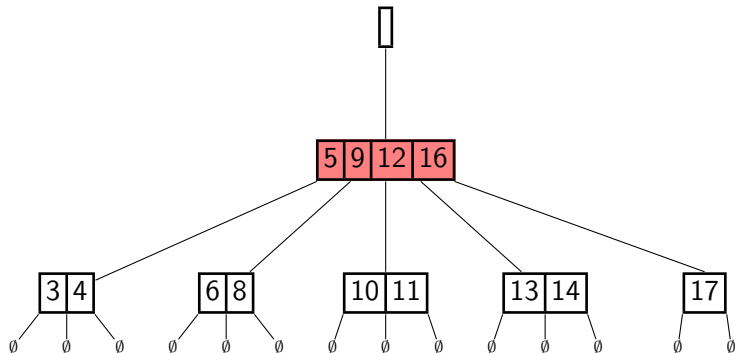
- Do *24Tree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.



Insertion in a 2-4 tree

Example: *insert*(17)

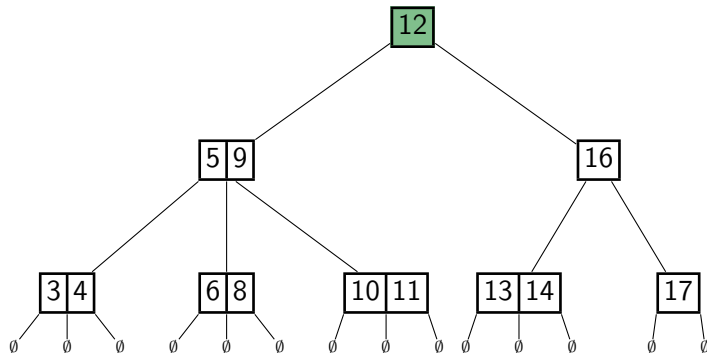
- Do *24Tree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.



Insertion in a 2-4 tree

Example: *insert*(17)

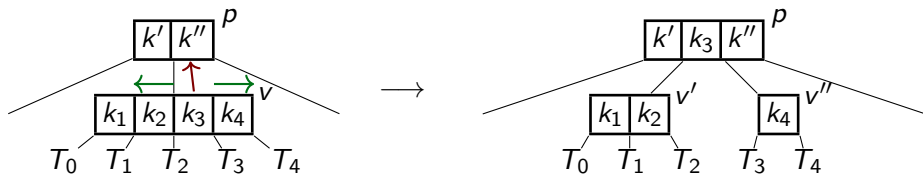
- Do *24Tree::search* and add key and empty subtree at leaf.
- If the leaf had room then we are done.
- Else **overflow**: More keys/subtrees than permitted.
- Resolve overflow by **node splitting**.



2-4 Tree operations

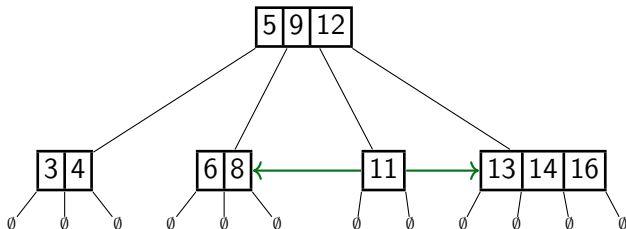
24Tree::insert(k)

1. $v \leftarrow 24Tree::search(k)$ // leaf where k should be
2. Add k and an empty subtree in key-subtree-list of v
3. **while** v has 4 keys (**overflow** \rightsquigarrow **node split**)
4. Let $\langle T_0, k_1, \dots, k_4, T_4 \rangle$ be key-subtree list at v
5. **if** (v has no parent) create a parent of v without KVPs
6. $p \leftarrow$ parent of v
7. $v' \leftarrow$ new node with keys k_1, k_2 and subtrees T_0, T_1, T_2
8. $v'' \leftarrow$ new node with key k_4 and subtrees T_3, T_4
9. Replace $\langle v \rangle$ by $\langle v', k_3, v'' \rangle$ in key-subtree-list of p
10. $v \leftarrow p$



Towards 2-4 Tree Deletion

- For deletion, we symmetrically will have to handle **underflow** (too few keys/subtrees)
- Crucial ingredient for this: **immediate sibling**

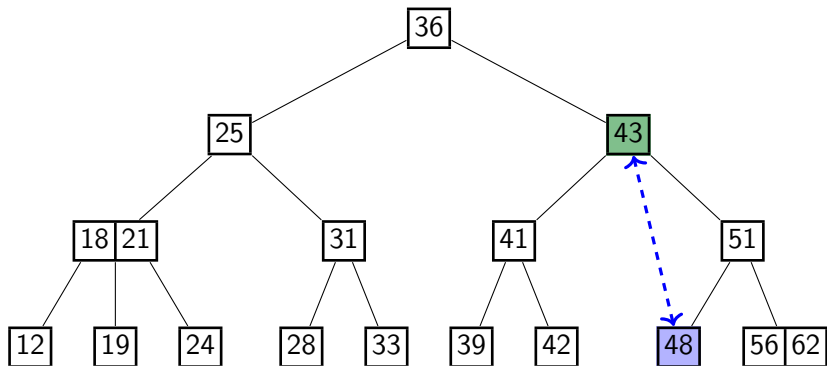


- **Observe:** Any node except the root has an immediate sibling.

2-4 Tree Deletion

Example: *delete*(43)

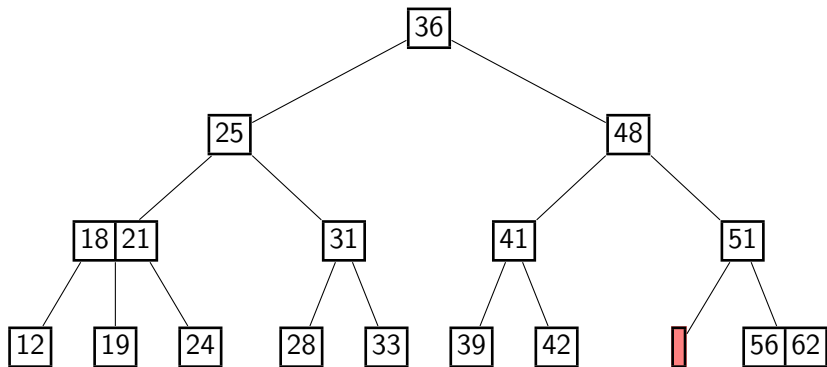
- *24Tree::search*, then trade with successor if KVP is not at a leaf.



2-4 Tree Deletion

Example: *delete*(43)

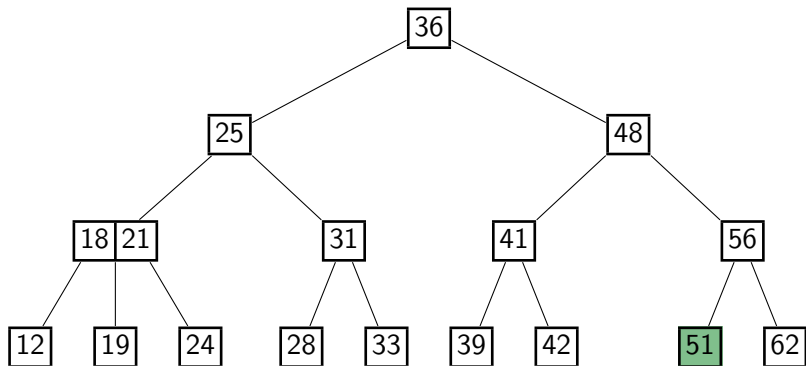
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:



2-4 Tree Deletion

Example: *delete*(43)

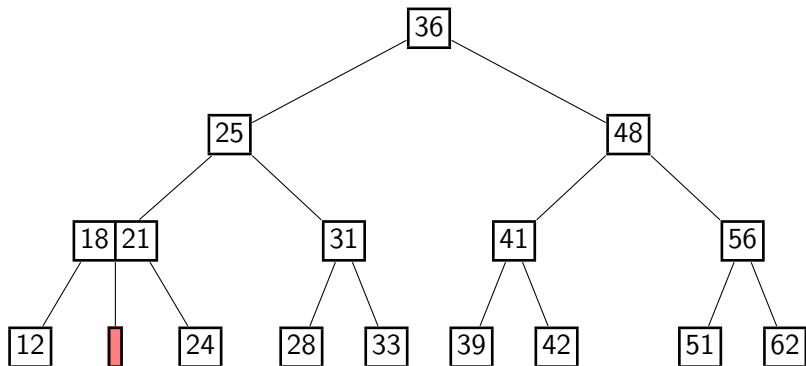
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**



2-4 Tree Deletion

Example: *delete*(19)

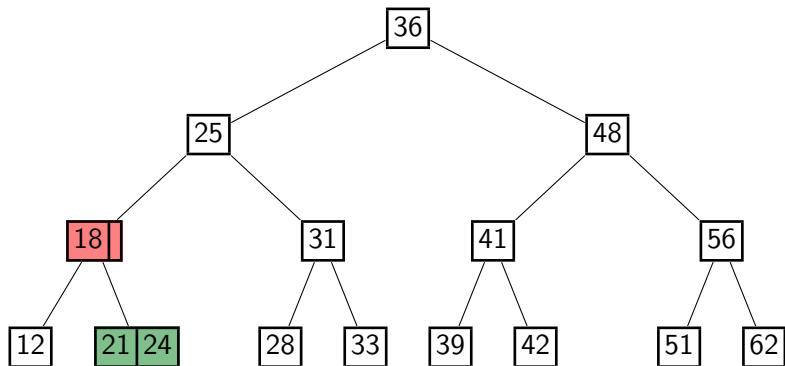
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**



2-4 Tree Deletion

Example: *delete*(19)

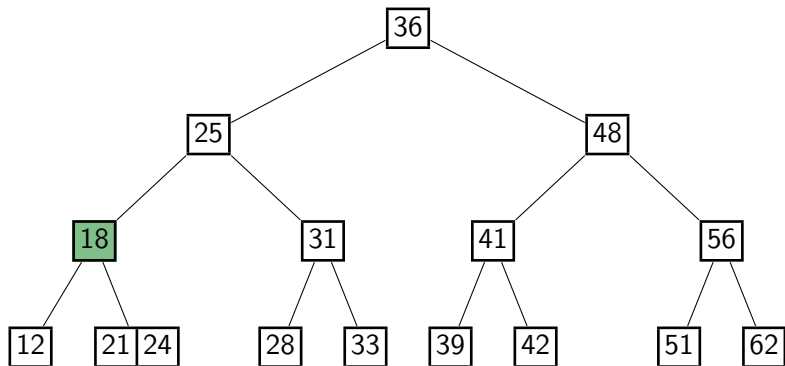
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



2-4 Tree Deletion

Example: *delete*(19)

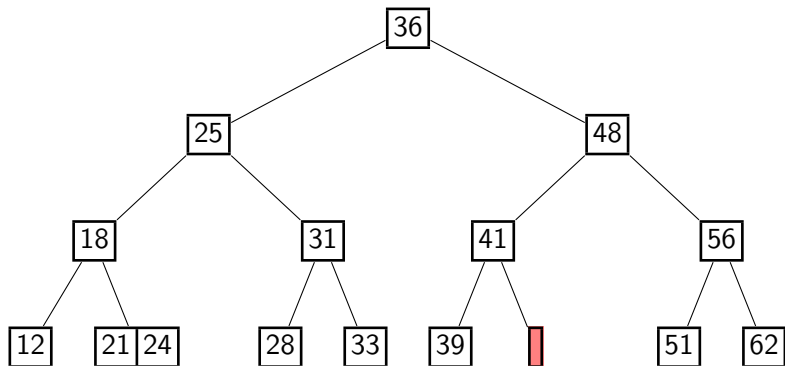
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



2-4 Tree Deletion

Example: *delete*(42)

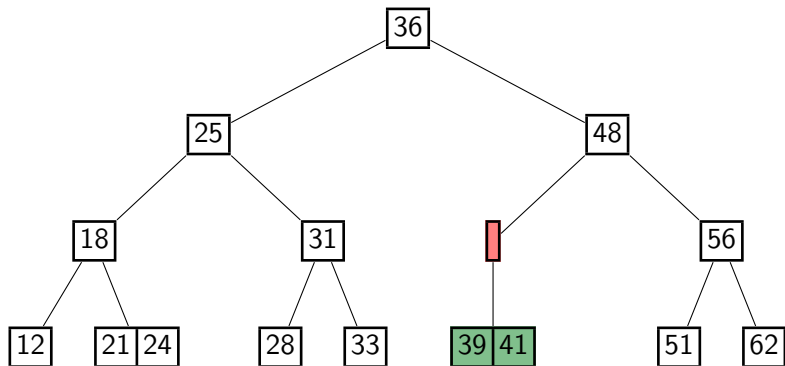
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



2-4 Tree Deletion

Example: *delete*(42)

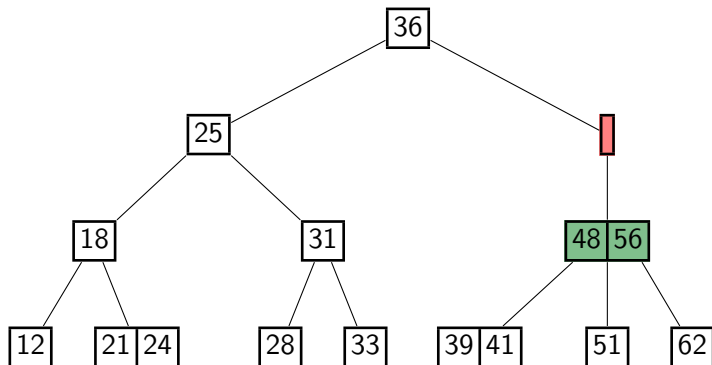
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



2-4 Tree Deletion

Example: *delete*(42)

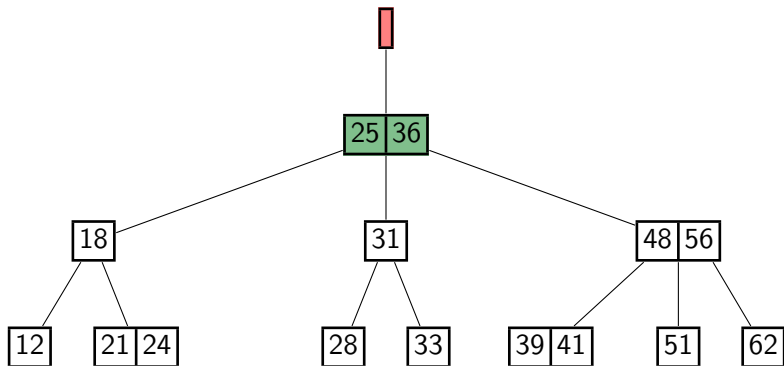
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



2-4 Tree Deletion

Example: *delete*(42)

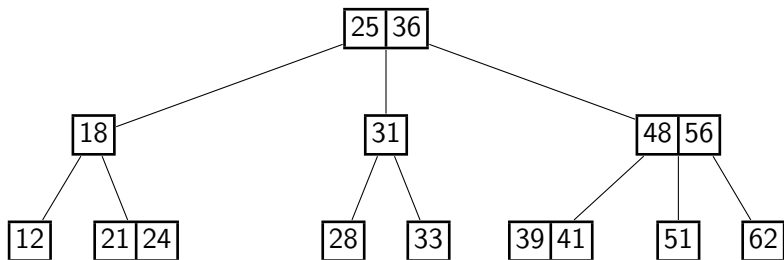
- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



2-4 Tree Deletion

Example: *delete*(42)

- *24Tree::search*, then trade with successor if KVP is not at a leaf.
- If underflow:
 - ▶ If immediate sibling has extras, **rotate/transfer**
 - ▶ Else **node merge** (this affects the parent!)



Deletion from a 2-4 Tree

24Tree::delete(k)

1. $v \leftarrow 24Tree::search(k)$ // node containing k
2. **if** v is not leaf
3. swap k with its successor k' and v with leaf containing k'
4. delete k and one empty subtree in v
5. **while** v has 0 keys (**underflow**)
6. **if** parent p of v is NIL, delete v and **break**
7. **if** v has immediate sibling u with 2 or more keys (**transfer/rotate**)
8. transfer the key of u that is nearest to v to p
9. transfer the key of p between u and v to v
10. transfer the subtree of u that is nearest to v to v
11. **break**
12. **else** (**merge & repeat**)
13. $u \leftarrow$ immediate sibling of v
14. transfer the key of p between u and v to u
15. transfer the subtree of v to u
16. delete node v and set $v \leftarrow p$

2-4 Tree summary

- A 2-4 tree has height $O(\log n)$
 - ▶ In internal memory, all operations have run-time $O(\log n)$.
 - ▶ This is no better than AVL-trees in theory.
(Though 2-4-trees are faster than AVL-trees in practice, especially when converted to binary search trees called *red-black trees*. No details.)
- A 2-4 tree has height $\Omega(\log n)$
 - ▶ Level i contains at most 4^i nodes
 - ▶ Each node contains at most 3 KVPs
- So not significantly better than AVL-trees w.r.t. block transfers.
- But we can generalize the concept to decrease the height.

Outline

1 External Memory

- Motivation
- Stream-based algorithms
- External sorting
- External Dictionaries
- 2-4 Trees
- *a-b*-Trees
- B-Trees

a - b -Trees

A 2-4 tree is an a - b -tree for $a = 2$ and $b = 4$.

An a - b -tree satisfies:

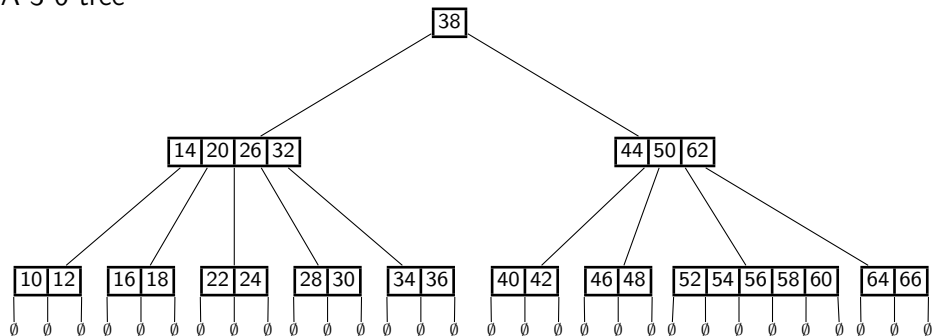
- Each node has at least a subtrees, unless it is the root. The root has at least 2 subtrees.
- Each node has at most b subtrees.
- If a node has d subtrees, then it stores $d-1$ key-value pairs (KVPs).
- Empty subtrees are at the same level.
- The keys in the node are between the keys in the corresponding subtrees.

Requirement: $a \leq \lceil b/2 \rceil = \lfloor (b+1)/2 \rfloor$.

search, *insert*, *delete* then work just like for 2-4 trees, after re-defining underflow/overflow to consider the above constraints.

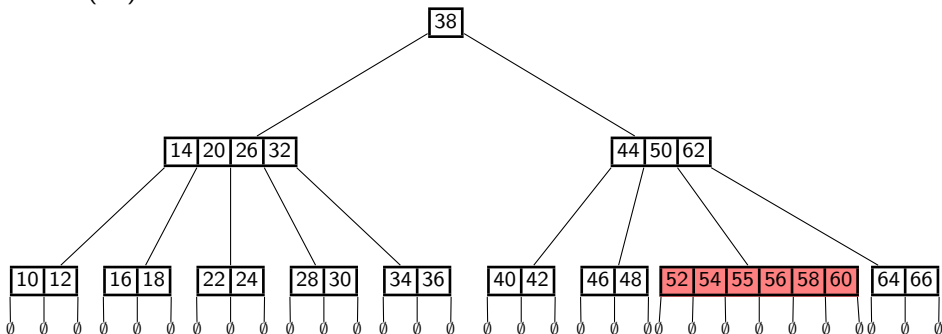
a-b-tree example

A 3-6-tree



a-*b*-tree insertion

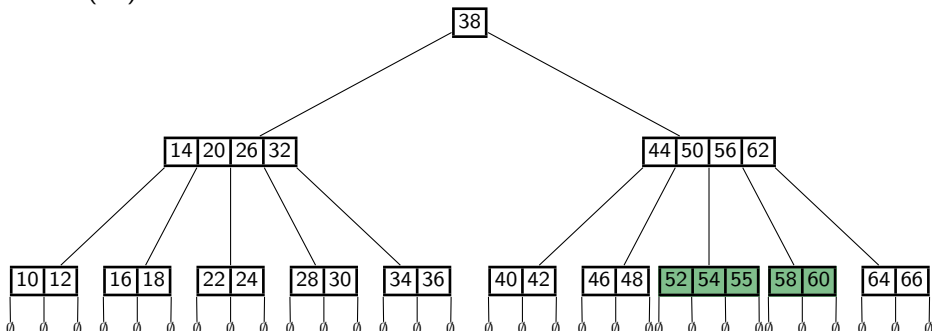
insert(55):



- Overflow now means b keys (and $b + 1$ subtrees)

a-*b*-tree insertion

insert(55):



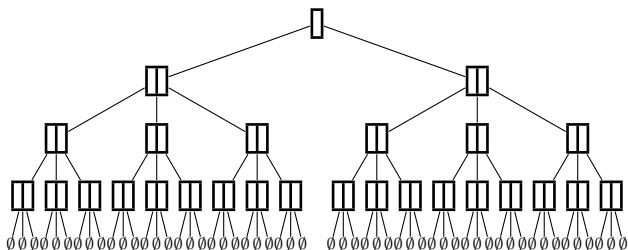
- Overflow now means b keys (and $b + 1$ subtrees)
- Node split \Rightarrow new nodes have $\geq \lfloor (b-1)/2 \rfloor$ keys
- Since we required $a \leq \lfloor (b+1)/2 \rfloor$, this is $\geq a-1$ keys as required.

Height of an a - b -tree

Recall: n = numbers of KVPs (*not* the number of nodes)

What is smallest possible number of KVPs in an a - b -tree of height- h ?

Level	Nodes
0	≥ 1
1	≥ 2
2	$\geq 2a$
3	$\geq 2a^2$
...	...
h	$\geq 2a^{h-1}$



$$\# \text{ nodes} \geq \underbrace{1}_{\text{root: } \geq 1 \text{ KVP}} + \underbrace{\sum_{i=0}^{h-1} 2a^i}_{\text{others: } \geq a-1 \text{ KVPs}}$$

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

Therefore the height of an a - b -tree is $O(\log_a(n)) = O(\log n / \log a)$.

a-b-trees as implementations of dictionaries

Analysis (if entire *a-b*-tree is stored in internal memory):

- *search*, *insert*, and *delete* each requires visiting $\Theta(\text{height})$ nodes
- Height is $O(\log n / \log a)$.
- Recall: $a \leq \lceil b/2 \rceil$ required for *insert* and *delete*

\Rightarrow choose $a = \lceil b/2 \rceil$ to minimize the height.

- Work at node can be done in $O(\log b)$ time.

$$\text{Total cost: } O\left(\frac{\log n}{\log a} \cdot (\log b)\right) = O\left(\log n \cdot \frac{\log b}{\log b - 1}\right) = O(\log n)$$

This is still no better than AVL-trees.

The main motivation for *a-b*-trees is *external memory*.

Outline

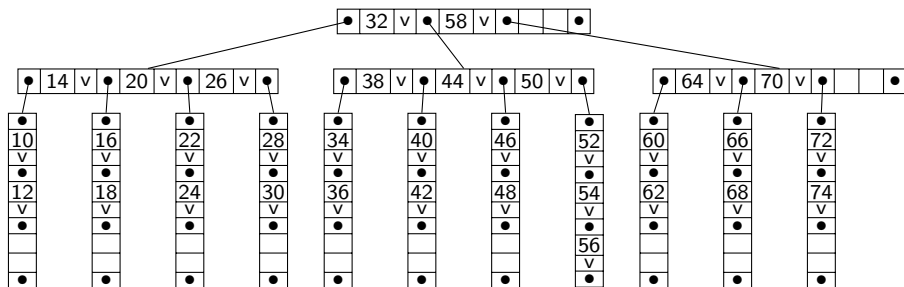
1 External Memory

- Motivation
- Stream-based algorithms
- External sorting
- External Dictionaries
- 2-4 Trees
- a - b -Trees
- B-Trees

B-trees

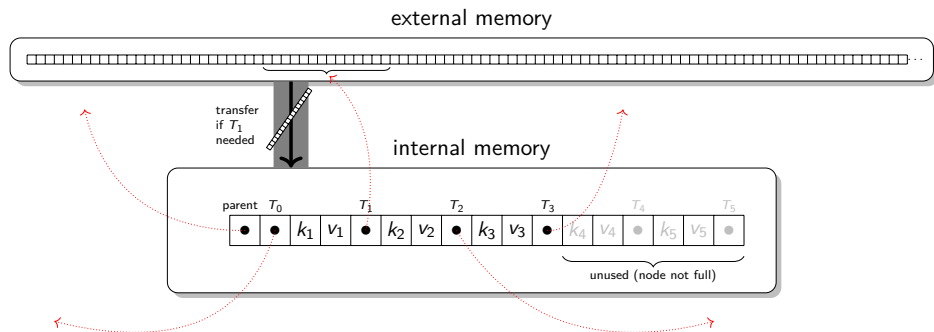
A **B-tree** is an a - b -tree tailored to the external memory model.

- Every node is one block of memory (of size B).
- b is chosen maximally such that a node with $b-1$ KVPs (hence $b-1$ value-references and b subtree-references) fits into a block.
 b is called the **order** of the B -tree. Typically $b \in \Theta(B)$.
- a is set to be $\lceil b/2 \rceil$ as before.



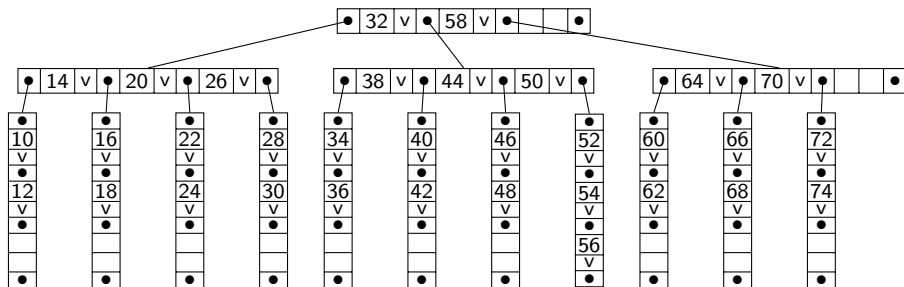
B-tree in external memory

Close-up on one node in one block:



In this example: 17 computer-words fit into one block, so the B -tree can have order 6.

B-tree analysis



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

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.
- In practice, height is a small constant.
 - ▶ Say $n = 2^{50}$, and $B = 2^{15}$. So roughly $b = 2^{14}$, $a = 2^{13}$.
 - ▶ B -tree of height 4 would have $\geq 1 + 2a^4 > 2^{50}$ KVPs.
 - ▶ So height is 3.
- There are some variations that are even better in practice (no details).
- B -trees are hugely important for storing data bases (\rightsquigarrow cs448)