

CS 240 – Data Structures and Data Management

Module 11: External Memory

A. Jamshidpey G. Kamath É. Schost

Based on lecture notes by many previous cs240 instructors

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

Spring 2020

References: Goodrich & Tamassia 14.1, Sedgewick 16.4

Outline

- 1 External Memory
 - Motivation
 - External Dictionaries
 - 2-3 Trees
 - (a, b) -Trees
 - B-Trees
 - External sorting

Outline

1 External Memory

- Motivation
- External Dictionaries
- 2-3 Trees
- (a, b) -Trees
- B-Trees
- External sorting

Different levels of memory

Current architectures:

- registers (very fast, very small)
- cache L1, L2 (still fast, less small)
- main memory
- external 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”).

New objective: revisit all ADTs/problems with the objective of minimizing page loads.

The External-Memory Model (EMM)

Two levels of memory:

- internal memory (CPU, . . .)
 - ▶ size M
 - ▶ fast access
- external memory
 - ▶ unlimited space
 - ▶ slow access
 - ▶ data moved to internal memory in *blocks* of B cells

Cost of computation:

“*I/O operations*” = # blocks transferred between internal and external memory

I/O operations are also called *page loads* or *block transfers*.

Outline

- 1 External Memory
 - Motivation
 - External Dictionaries
 - 2-3 Trees
 - (a, b) -Trees
 - B-Trees
 - External sorting

Dictionaries in external memory

Tree-based dictionary implementations have poor *memory locality*:
If an operation accesses m nodes, then it must access m spaced-out memory locations.

- In an AVL tree, $\Theta(\log n)$ pages are loaded in the worst case.
- Better solution: do more in single node \rightsquigarrow B-trees
- First consider special case of B-trees: *2-3 trees*
 - ▶ 2-3-trees would also be interesting for implementing ADT Dictionaries in main memory (may be even faster than AVL-trees)
 - ▶ We first analyze their performance in main memory, and then (for B-trees) in external memory.

Outline

1 External Memory

- Motivation
- External Dictionaries
- **2-3 Trees**
- (a, b) -Trees
- B-Trees
- External sorting

2-3 Trees

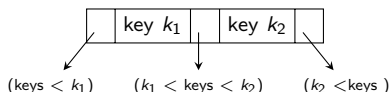
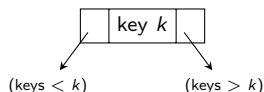
A 2-3 Tree is a balanced search tree that is not necessarily binary.

Structural properties:

- Every internal node is either
 - ▶ 1-node: *one KVP* and *two children*, or
 - ▶ 2-node: *two KVPs* and *three children*.
- The external nodes are *NIL* (do not store keys)
- All external nodes are at the same level.

Height-balance strictly enforced, but allow 2 types of nodes!

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



2-3 Tree operations

Search: The order-property determines the subtree to search in.

23TreeSearch($k, v \leftarrow \text{root}$)

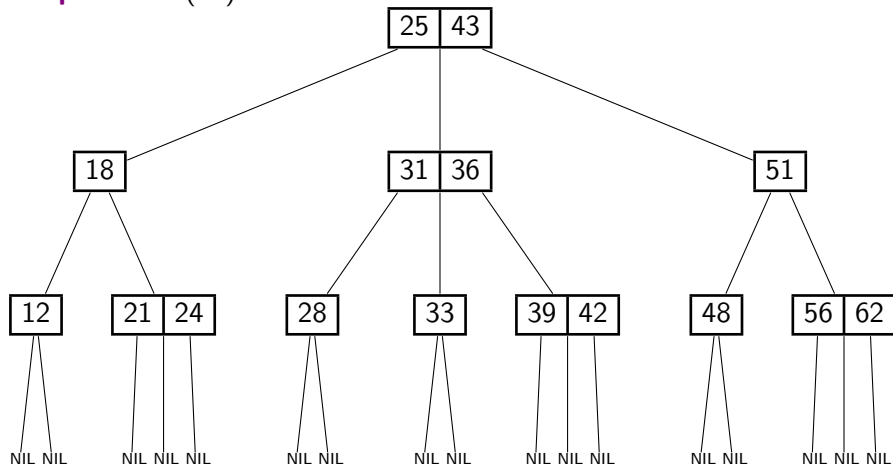
1. Let $c_0, k_1, \dots, k_d, c_d$ be keys and children at v , in order
2. **if** $k \geq k_1$
3. $i \leftarrow$ maximal index such that $k_i \leq k$
4. **if** $k_i = k$ **return** k_i
5. **else** $i \leftarrow 0$
6. *23TreeSearch*(k, c_i)

Insert: Nodes may grow from bottom to top.

- Search to find leaf ℓ where the new key k belongs.
- Add k and a NIL-child to ℓ . If ℓ now has 3 keys (**overflow**):
 - ▶ Split ℓ into two nodes ℓ, ℓ' with min and max key of ℓ
 - ▶ Move median key of ℓ into parent p of ℓ . Also make ℓ' child of p .
 - ▶ Recurse in p if it now has overflow.

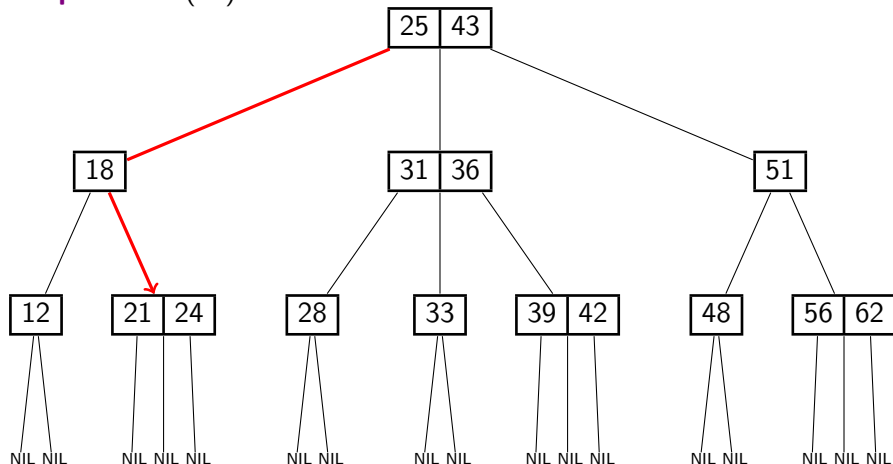
Example: Insertion in a 2-3 tree

Example: *insert*(19)



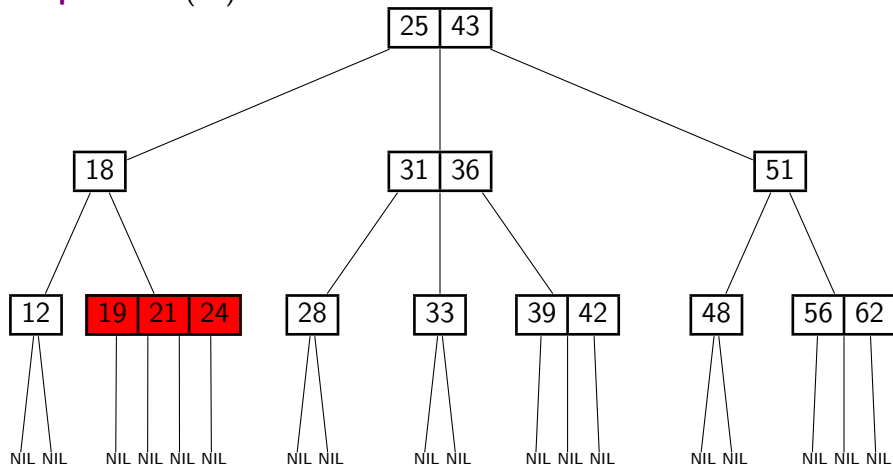
Example: Insertion in a 2-3 tree

Example: *insert*(19)



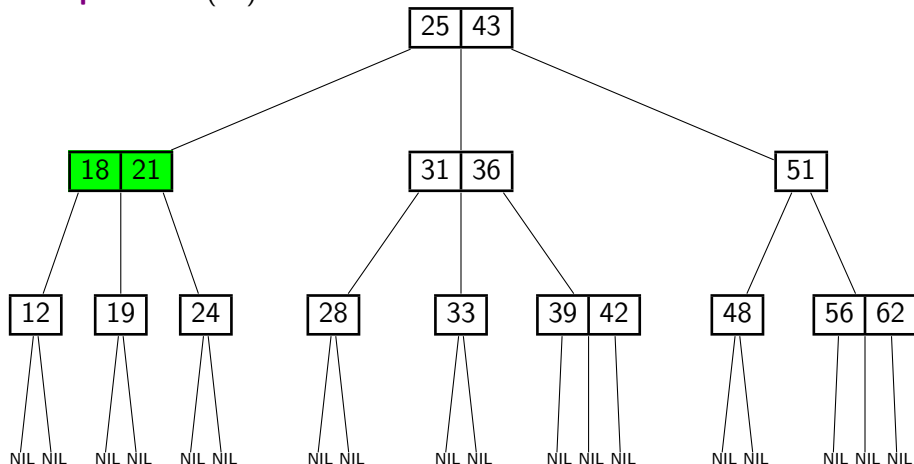
Example: Insertion in a 2-3 tree

Example: *insert*(19)



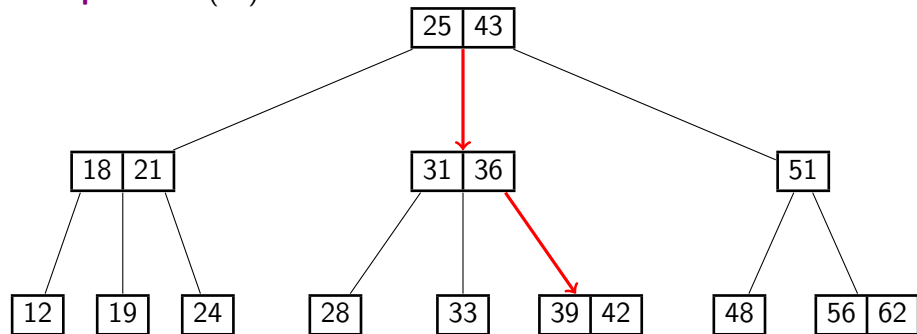
Example: Insertion in a 2-3 tree

Example: *insert*(19)



Example: Insertion in a 2-3 tree

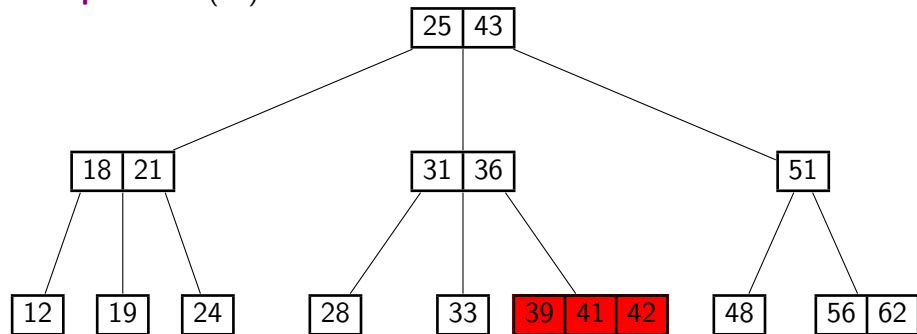
Example: *insert*(41)



(NIL-nodes not shown to simplify picture)

Example: Insertion in a 2-3 tree

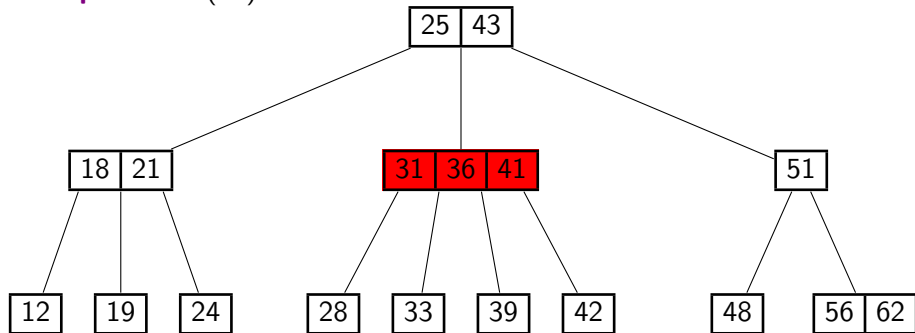
Example: *insert*(41)



(NIL-nodes not shown to simplify picture)

Example: Insertion in a 2-3 tree

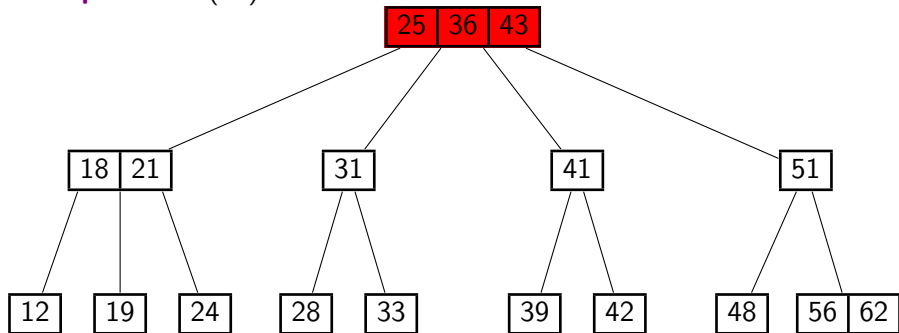
Example: *insert*(41)



(NIL-nodes not shown to simplify picture)

Example: Insertion in a 2-3 tree

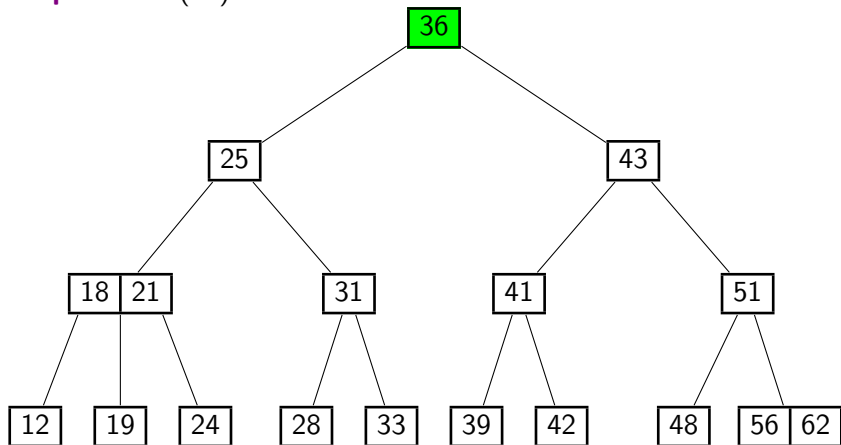
Example: *insert*(41)



(NIL-nodes not shown to simplify picture)

Example: Insertion in a 2-3 tree

Example: *insert*(41)



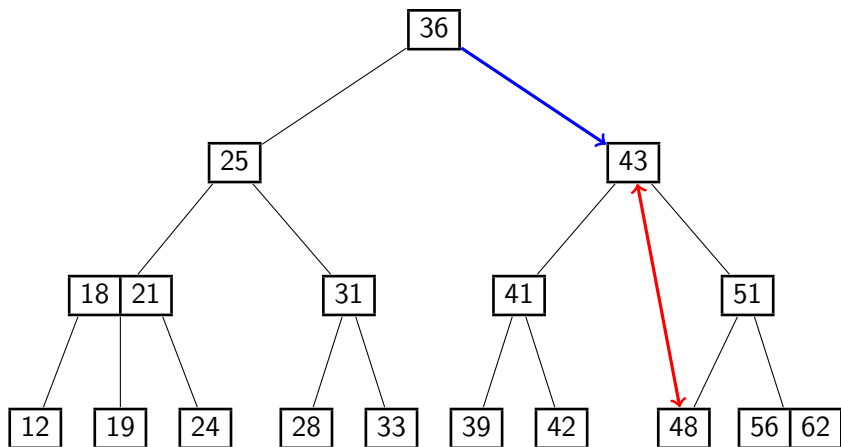
(NIL-nodes not shown to simplify picture)

Deletion from a 2-3 Tree

- As with BSTs and AVL trees, we first swap the KVP k with its successor, so that it is now at a leaf ℓ .
- Delete k and one NIL-child from ℓ .
- If ℓ now has 0 keys (*underflow*)
 - ▶ If ℓ is the root, simply delete it. Else let p be the parent of ℓ .
 - ▶ If some *immediate* sibling u is a 2-node, perform a *transfer*:
 - ★ Find the key k_p in p that is between keys of ℓ and u .
 - ★ “Rotate:” move k_p into ℓ , move adjacent KVP from u into p , and re-arrange children suitably.
 - ▶ Otherwise, we *merge* ℓ and a 1-node sibling u :
 - ★ Find the key k_p in parent p between keys of ℓ and u .
 - ★ Combine ℓ and u into one node and move k_p into it.
 - ★ Recurse in p if it now has underflow.

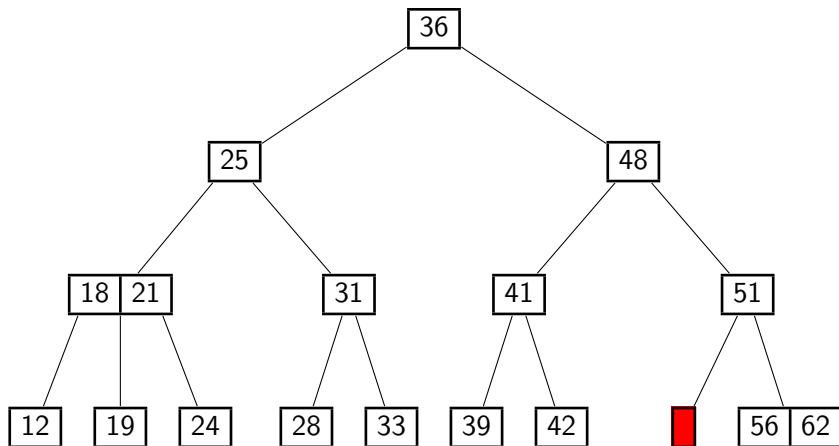
2-3 Tree Deletion

Example: *delete*(43)



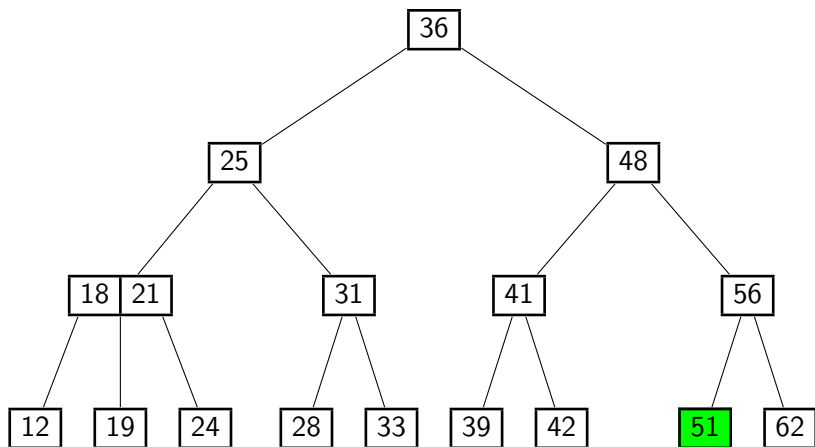
2-3 Tree Deletion

Example: *delete*(43)



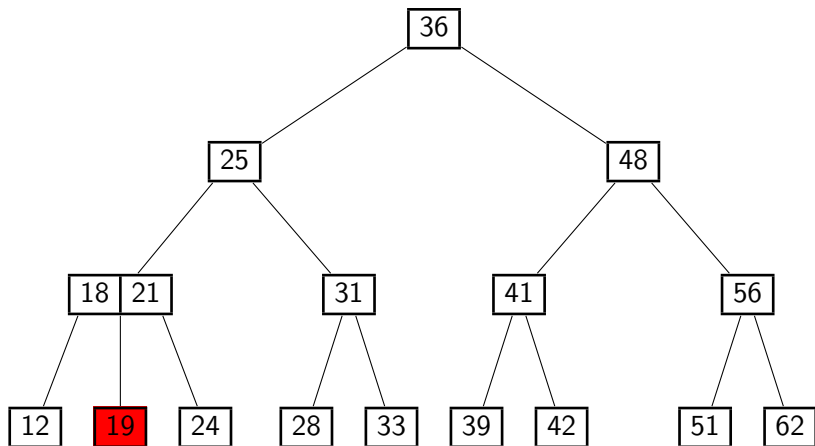
2-3 Tree Deletion

Example: *delete*(43)



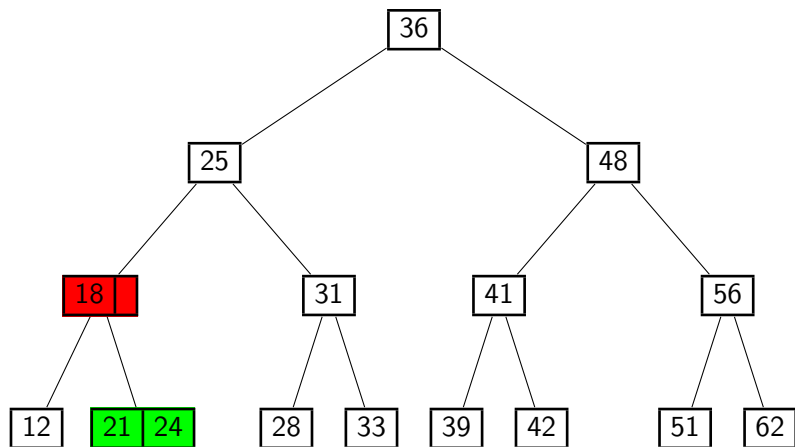
2-3 Tree Deletion

Example: *delete*(19)



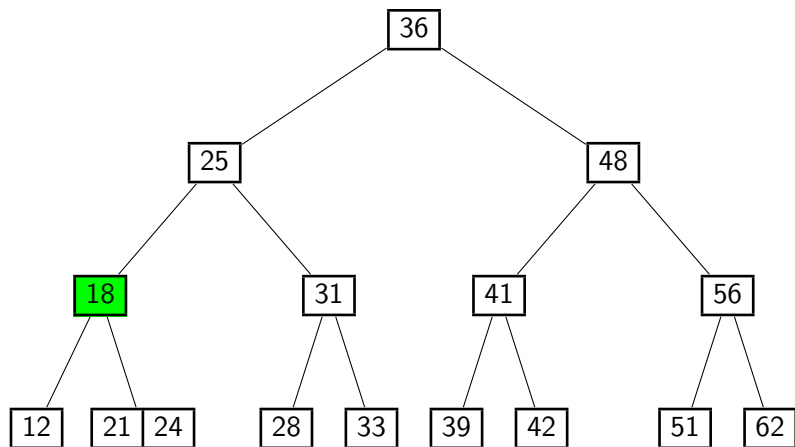
2-3 Tree Deletion

Example: *delete*(19)



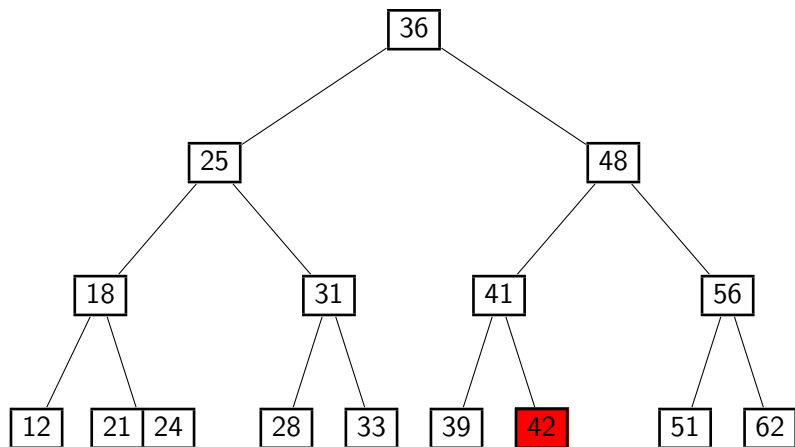
2-3 Tree Deletion

Example: *delete*(19)



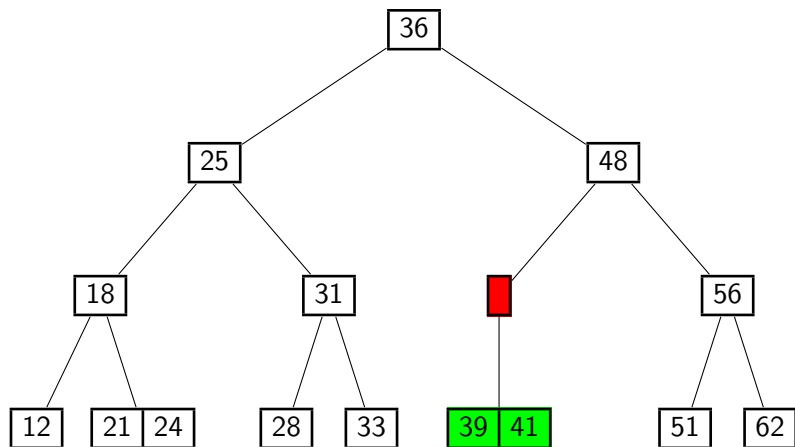
2-3 Tree Deletion

Example: *delete*(42)



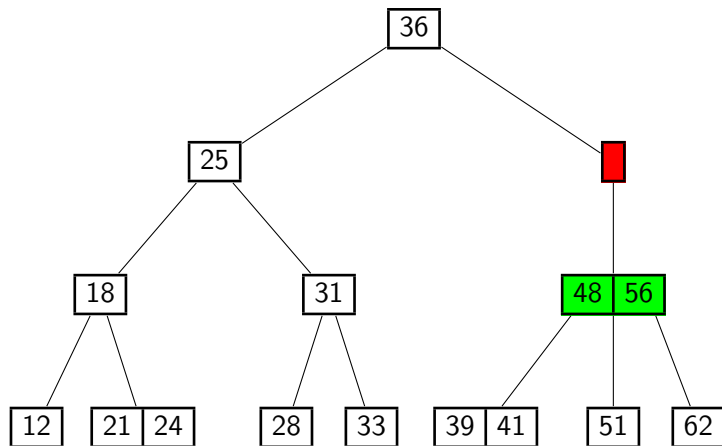
2-3 Tree Deletion

Example: *delete*(42)



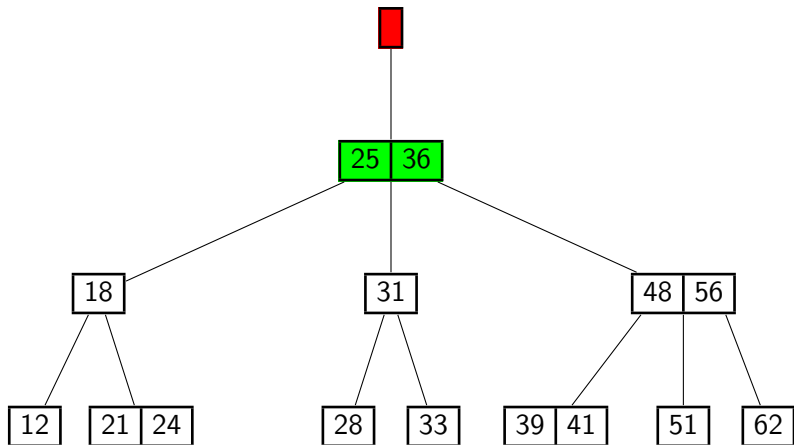
2-3 Tree Deletion

Example: *delete*(42)



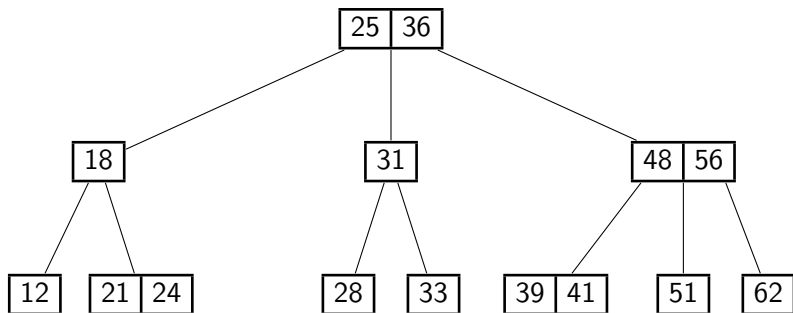
2-3 Tree Deletion

Example: *delete*(42)



2-3 Tree Deletion

Example: *delete*(42)



Outline

1 External Memory

- Motivation
- External Dictionaries
- 2-3 Trees
- (a, b) -Trees
- B-Trees
- External sorting

(a, b) -Trees

The 2-3 Tree is a specific type of (a, b) -tree:

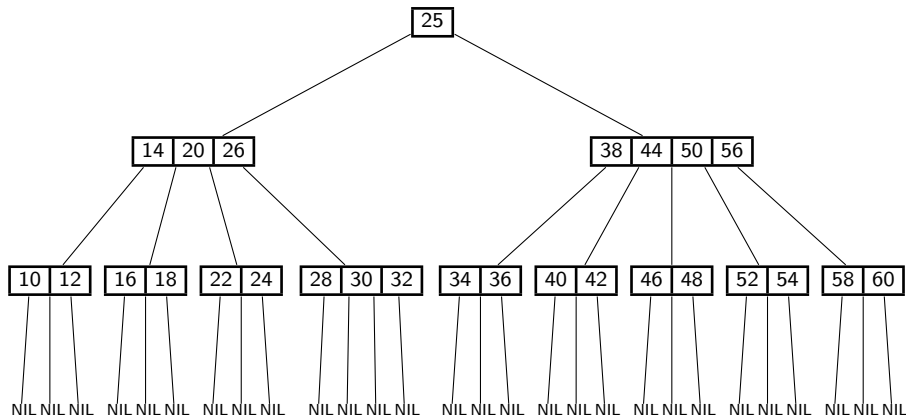
An (a, b) -tree satisfies:

- Each internal node has at least a children, unless it is the root. The root has at least 2 children.
- Each internal node has at most b children.
- If a node has k children, then it stores $k - 1$ key-value pairs (KVPs).
- External nodes store no keys and are at the same level.
- The keys in the node are between the keys in the corresponding children.

If $a \leq (b + 1)/2$, then *search*, *insert*, *delete* work just like for 2-3 trees, after re-defining underflow/overflow to consider the above constraints.

(a, b) -tree example

A $(3, 5)$ -tree (it is also a valid $(3, 6)$ -tree):



Height of an (a, b) -tree

What is the least number of KVPs in an (a, b) -tree of height- h ?
(Height = # levels **not** counting the NIL-level -1)

Level	Nodes \geq	Links/node \geq	KVP/node \geq	KVPs on level \geq
0	1	2	1	1
1	2	a	$a - 1$	$2(a - 1)$
2	$2a$	a	$a - 1$	$2a(a - 1)$
3	$2a^2$	a	$a - 1$	$2a^2(a - 1)$
...
h	$2a^{h-1}$	a	$a - 1$	$2a^{h-1}(a - 1)$

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

Therefore height of tree with n KVPs is $O(\log_a(n)) = O(\log(n)/\log(a))$.
Can also prove that the height is $\Omega(\log_b(n))$.

Outline

1 External Memory

- Motivation
- External Dictionaries
- 2-3 Trees
- (a, b) -Trees
- **B-Trees**
- External sorting

B-trees

A *B-tree of order m* is a $(\lceil m/2 \rceil, m)$ -tree.

A 2-3 tree is a B-tree of order 3.

Sedgwick uses M rather than m , but this is confusing since we set M to be the space in main memory.

Analysis (if entire B-tree is stored in main memory):

- *search*, *insert*, and *delete* each require $\Theta(\text{height})$ node operations.
- Height is $\Theta(\log n / \log m)$.
- Assume each node stores its KVPs and child-pointers in a dictionary that supports $\Theta(\log m)$ search.

Total cost for search is $\Theta\left(\frac{\log n}{\log m} \cdot (\log m)\right) = \Theta(\log n)$. This is no better than 2-3-trees or AVL-trees.

Insert and delete need a bit more care.

Dictionaries in external memory

Main applications of B-trees: Store dictionaries in external memory.

Recall: In an AVL tree or 2-3 tree, $\Theta(\log n)$ pages are loaded in the worst case.

Instead, use a B-tree of order m , where m is chosen so that an m -node fits into a single page.

Each operation can be done with $\Theta(\text{height})$ page loads.

The height of a B-tree is $\Theta(\log n / \log m)$.

This results in huge savings of page loads.

Outline

1 External Memory

- Motivation
- External Dictionaries
- 2-3 Trees
- (a, b) -Trees
- B-Trees
- External sorting

Sorting in external memory

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.

- Recall: Heapsort was optimal in time and space in RAM model
- **But:** Heapsort accesses A at indices that are far apart
 \rightsquigarrow typically one page loads per array access.
- Mergesort adapts well to an array stored in external memory.
- It can be made even more effective using **d-way merge**: Merge d sorted runs into one sorted run.

d-way merge

d-Way-Merge(S_1, \dots, S_d)

S_1, \dots, S_d are sorted sets (arrays/lists/stacks/queues)

1. $P \leftarrow$ empty min-priority queue
2. $S \leftarrow$ empty set
3. **for** $i \leftarrow 1$ to d **do**
4. $P.insert((\text{first element of } S_i, i))$
5. **while** P is not empty **do**
6. $(x, i) \leftarrow \text{deleteMin}(P)$
7. remove x from S_i and append it to S
8. **if** S_i is not empty **do**
9. $P.insert((\text{first element of } S_i, i))$

- Standard mergesort uses $d = 2$
- $d > 2$ could be used in internal memory as well, but the extra time to find minimum in the priority queue means the overall run-time is no better.

Mergesort in external memory

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 n/M sorted runs of length M .

Mergesort in external memory

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 n/M sorted runs of length M .

Mergesort in external memory

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 n/M sorted runs of length M .

Mergesort in external memory

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 →

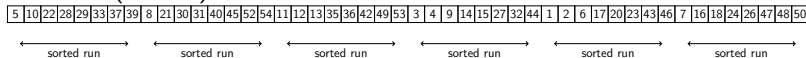
Internal ($M = 8$):



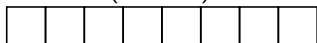
- 1 Create n/M sorted runs of length M .

Mergesort in external memory

External ($B = 2$):



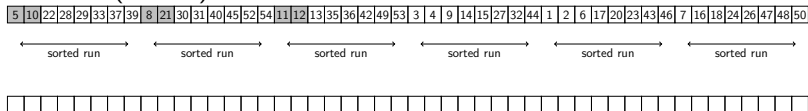
Internal ($M = 8$):



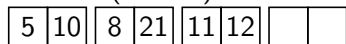
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):

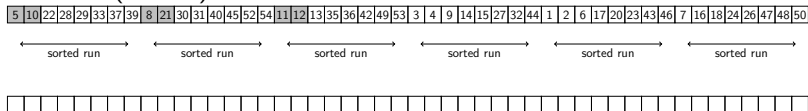


(priority queue not shown)

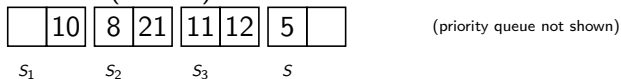
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



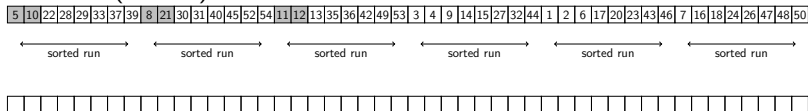
Internal ($M = 8$):



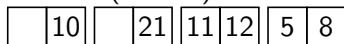
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):



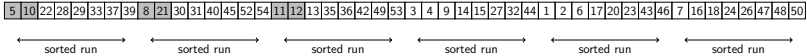
(priority queue not shown)

S_1 S_2 S_3 S

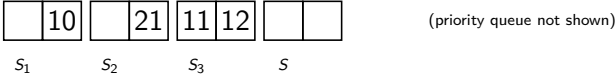
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



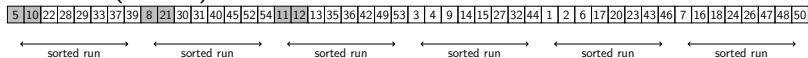
Internal ($M = 8$):



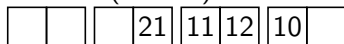
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):

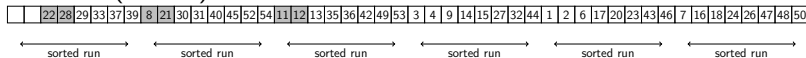


(priority queue not shown)

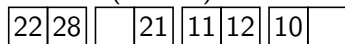
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):

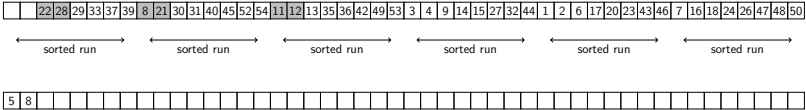


(priority queue not shown)

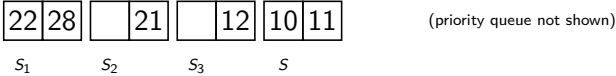
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



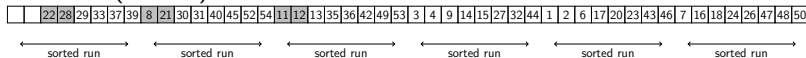
Internal ($M = 8$):



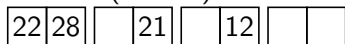
- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):

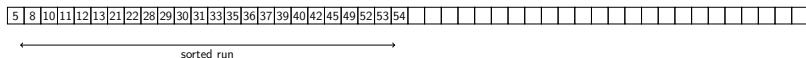
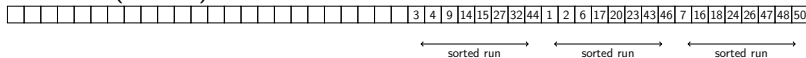


(priority queue not shown)

- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):

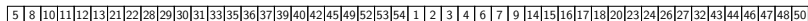


(priority queue not shown)

- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):

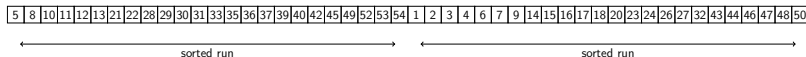
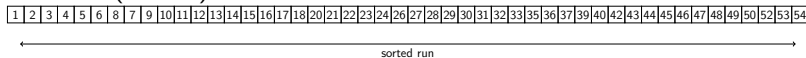


(priority queue not shown)

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

Mergesort in external memory

External ($B = 2$):



Internal ($M = 8$):



(priority queue not shown)

S_1 S_2 S_3 S

- 1 Create n/M sorted runs of length M . $\Theta(n/B)$ IO-operations
- 2 Merge the first $d \approx M/B - 1$ sorted runs using *d-Way-Merge*
- 3 Keep merging the next runs to reduce $\#$ runs by factor of d
 \rightsquigarrow one round of merging. $\Theta(n/B)$ IO-operations
- 4 $\log_d(n/M)$ rounds of merging create sorted array.

Mergesort with external memory

Total page loads: $O(\log_d(n) \cdot n/B)$.

Assuming the EMM, one can prove lower bounds!

- $\Omega(\frac{n}{B})$ I/Os required to **scan** n elements.
- $\Omega(\frac{n}{B} \log_{M/B}(\frac{n}{B}))$ I/Os required to **sort** n elements with comparisons.
 - ▶ We don't prove that here.

Mergesort with external memory

Total page loads: $O(\log_d(n) \cdot n/B)$.

Assuming the EMM, one can prove lower bounds!

- $\Omega(\frac{n}{B})$ I/Os required to **scan** n elements.
- $\Omega(\frac{n}{B} \log_{M/B}(\frac{n}{B}))$ I/Os required to **sort** n elements with comparisons.
 - ▶ We don't prove that here.
- d -way Mergesort with $d \approx M/B$ is optimal (up to constant factors)!