Height of an AVL tree

Define $N(h)$ to be the *least* number of nodes in a height-h AVL tree.

One subtree must have height at least $h-1$, the other at least $h-2$:

$$N(h) = \begin{cases} 1 + N(h-1) + N(h-2), & h \geq 1 \\ 1, & h = 0 \\ 0, & h = -1 \end{cases}$$

What sequence does this look like? The Fibonacci sequence!

$$N(h) = F_{h+3} - 1 = \left\lceil \frac{\varphi^{h+3}}{\sqrt{5}} \right\rfloor - 1, \text{ where } \varphi = \frac{1+\sqrt{5}}{2}$$

AVL Tree Analysis

Easier lower bound on $N(h)$:

$$N(h) > 2N(h-2) > 4N(h-4) > 8N(h-6) > \cdots > 2^i N(h-2i) \geq 2^{\lfloor h/2 \rfloor}$$

AVL Tree Analysis

Easier lower bound on $N(h)$:

$$N(h) > 2N(h-2) > 4N(h-4) > 8N(h-6) > \cdots > 2^i N(h-2i) \geq 2^{\lfloor h/2 \rfloor}$$

Since $n > 2^{\lfloor h/2 \rfloor}$, $h \leq 2 \lg n$,
and thus an AVL tree with n nodes has height $O(\log n)$.
Also, $n \leq 2^{h+1} - 1$, so the height is $\Theta(\log n)$.

$\Rightarrow$ *search*, *insert*, *delete* all cost $\Theta(\log n)$.

2-3 Trees

A 2-3 Tree is like a BST with additional structual properties:

- Every node either contains *one KVP* and *two children*,
  or *two KVPs* and *three children*.
- All the leaves are at the same level.
  (A leaf is a node with empty children.)

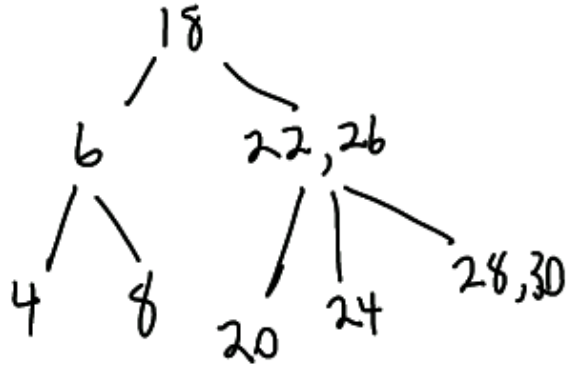
Searching through a 1-node is just like in a BST.
For a 2-node, we must examine both keys and follow the appropriate path.

B-trees

level 0

1

2



An (a,b) B-tree

1- An "ordered tree"

2- Each internal node has at least a and at most b children

Except root which has at least 2 and at most b children

3. a node with k children has k-1 kvps

4. All leaves have same level

A B-tree of 'order M' $\Rightarrow (\lceil \frac{M}{2} \rceil, M)$

B-tree → insert:

1 → insert at a leaf

2 → For overfilled nodes ⇒ send middle key to the parent split to two separate nodes

B-tree Delete(g)

1- like BSTs, replace g w/ its successor / predecessor if required

2- if a node becomes underloaded

↳ if ∃ a direct sibling with extra key (more than 'a' keys) take a key from parent
parent gets a key from sibling

(a, b) - tree → at least a
most b children / node

each node → at least a-1
at most b-1 KVPs.

if $a = \lceil \frac{M}{2} \rceil$
$b = M$
then we have B-tree of order - M

If M=3 → a=2
↳ b=3 → 2-3 tree

Insertion in a 2-3 tree

First, we search to find the leaf where the new key belongs.

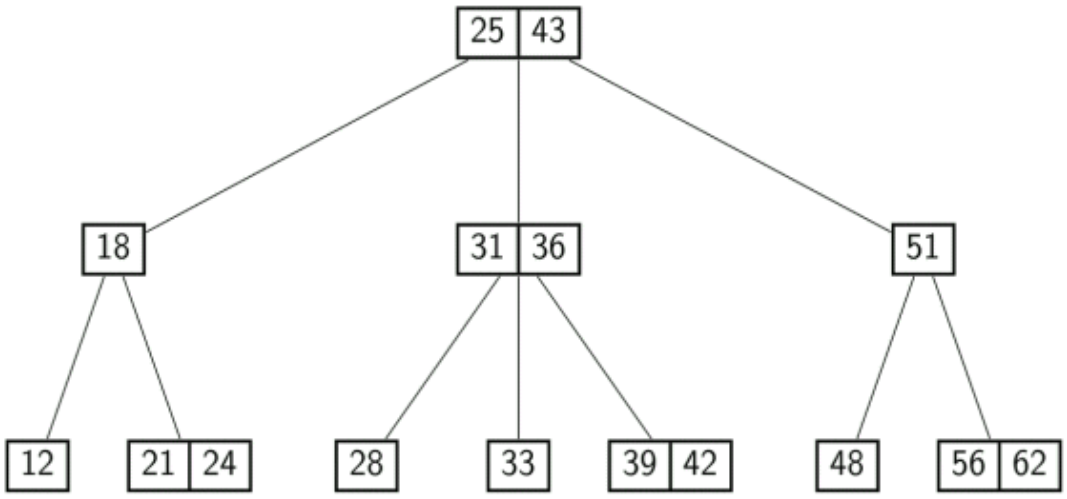
If the leaf has only 1 KVP, just add the new one to make a 2-node.

Otherwise, order the three keys as $a < b < c$.
Split the leaf into two 1-nodes, containing a and c,
and (recursively) insert b into the parent along with the new link.

at most $b-1$ keys per node to locate the right path to follow $\rightarrow \log b \rightarrow$ insertion takes $O(h \log b)$ (binary search)

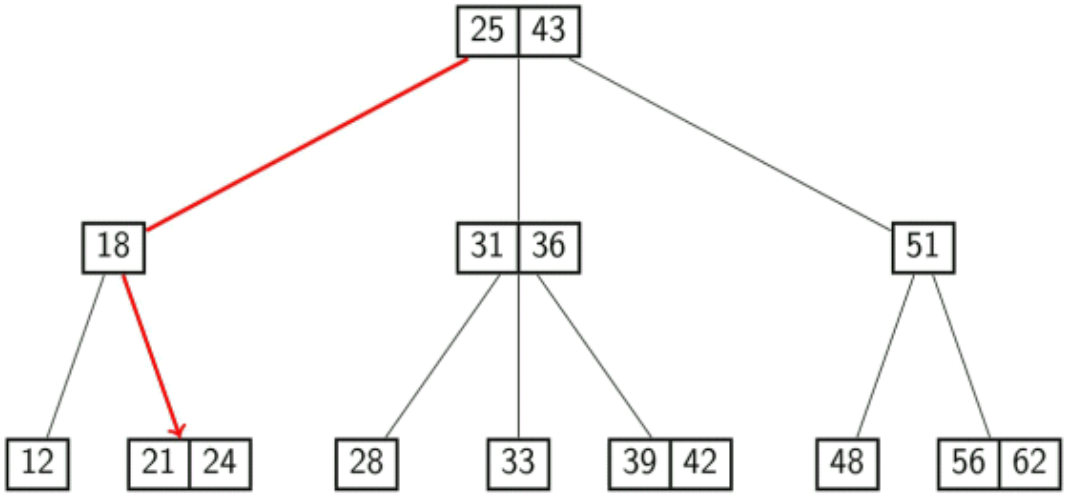
2-3 Tree Insertion

Example: *insert*(19)



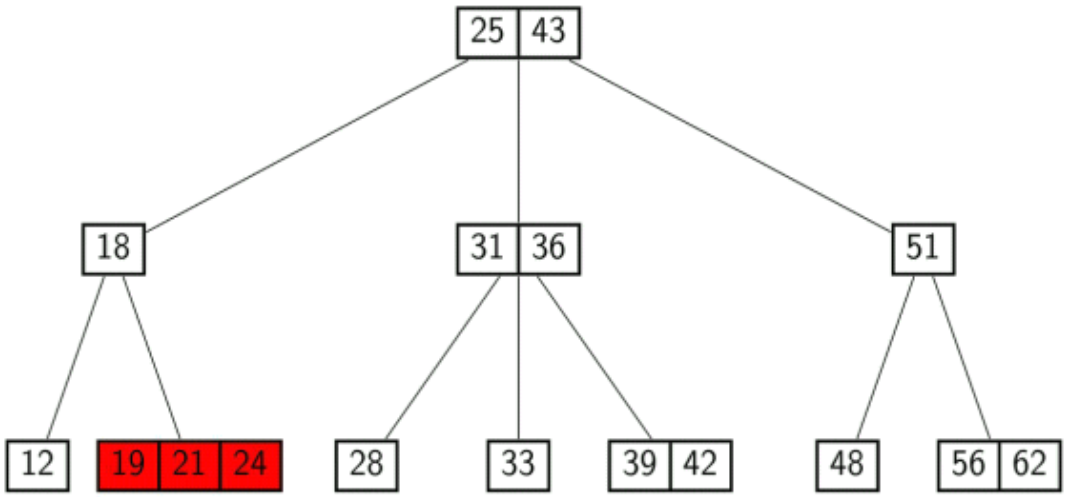
2-3 Tree Insertion

Example: *insert*(19)



2-3 Tree Insertion

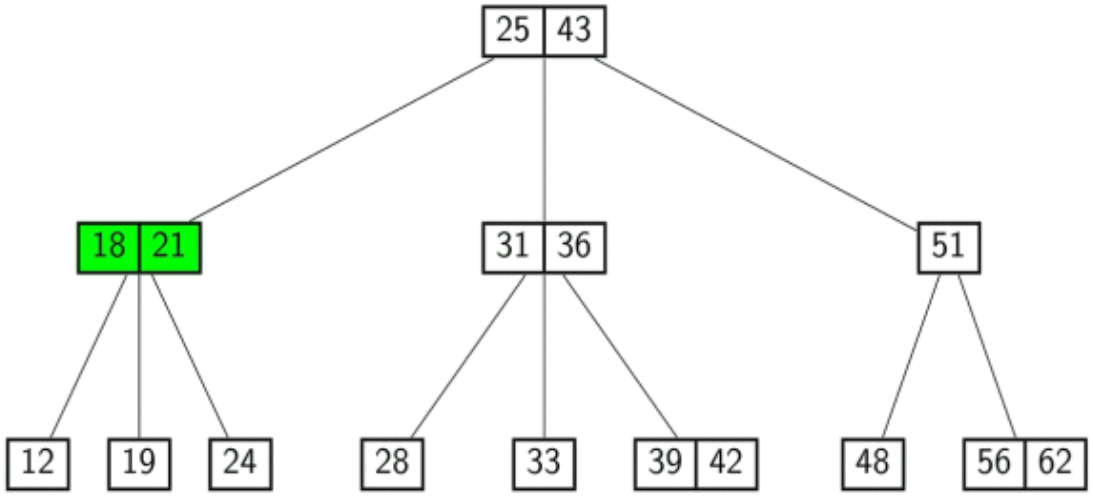
Example: *insert*(19)



Biedl,Maftuleac (SCS, UW) CS240 - Module 4 Winter 2015 23 / 1

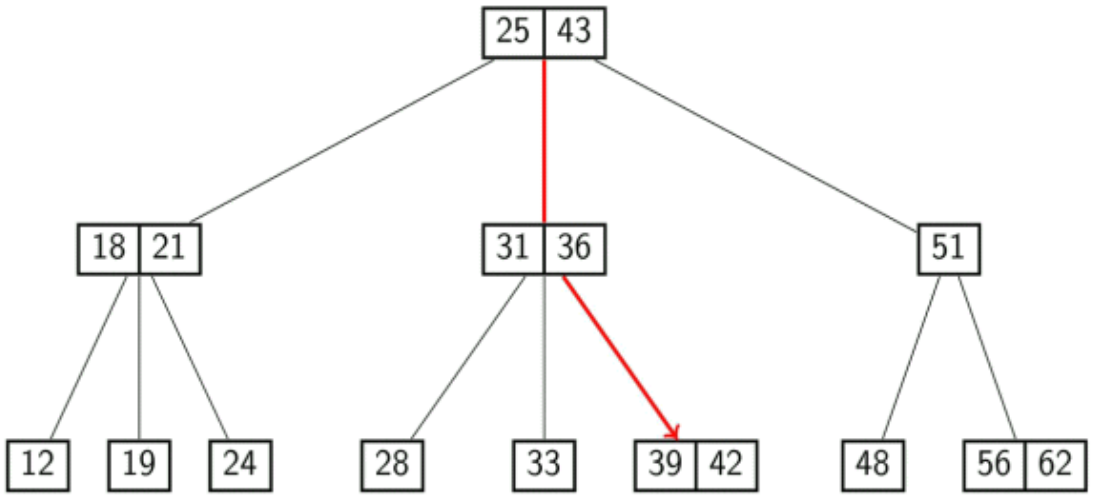
2-3 Tree Insertion

Example: *insert*(19)



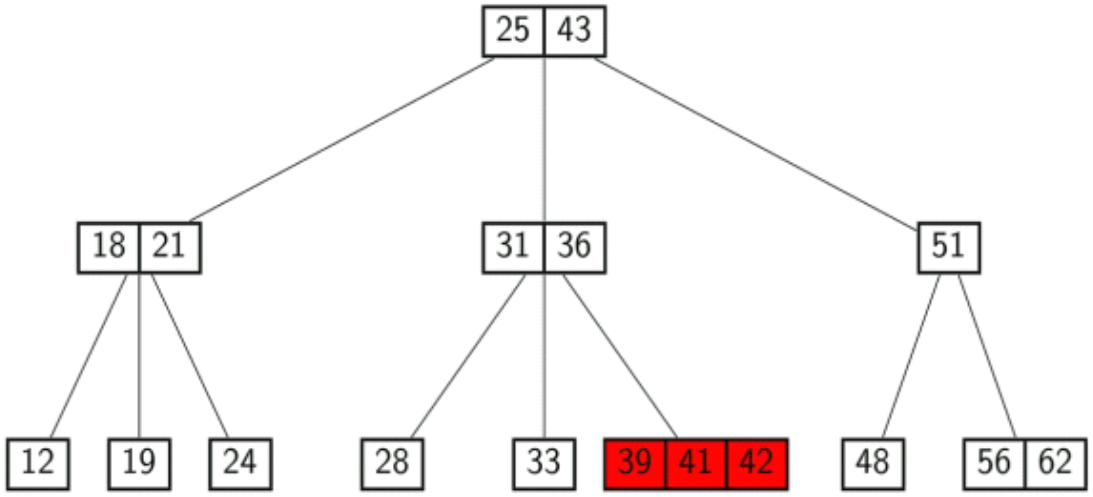
2-3 Tree Insertion

Example: *insert*(41)



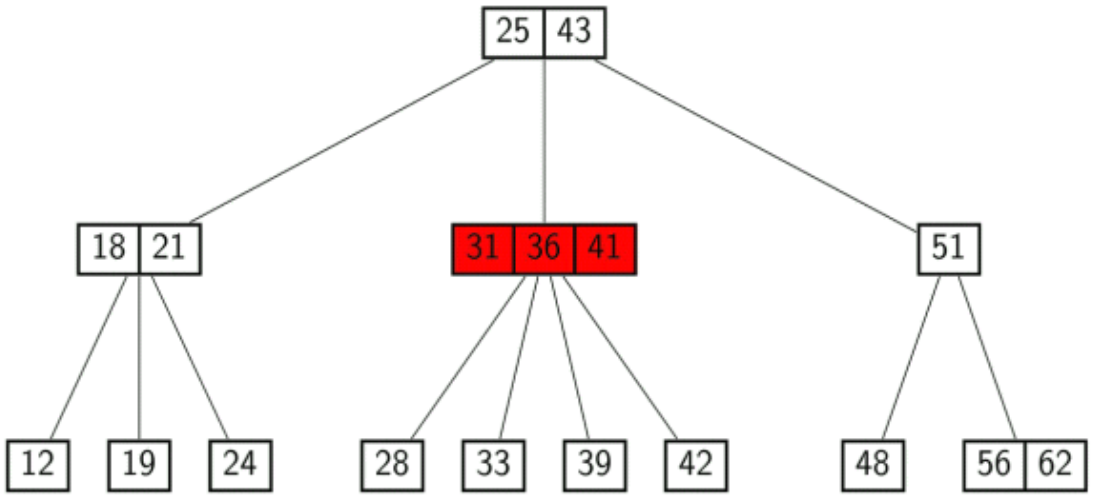
2-3 Tree Insertion

Example: *insert*(41)



2-3 Tree Insertion

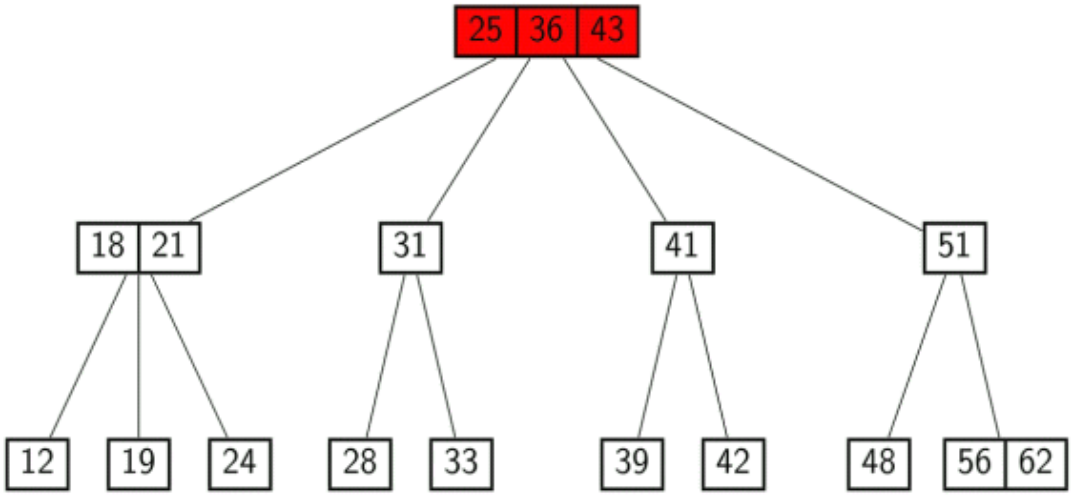
Example: *insert*(41)



Biedl,Maftuleac (SCS, UW) CS240 - Module 4 Winter 2015 23 / 1

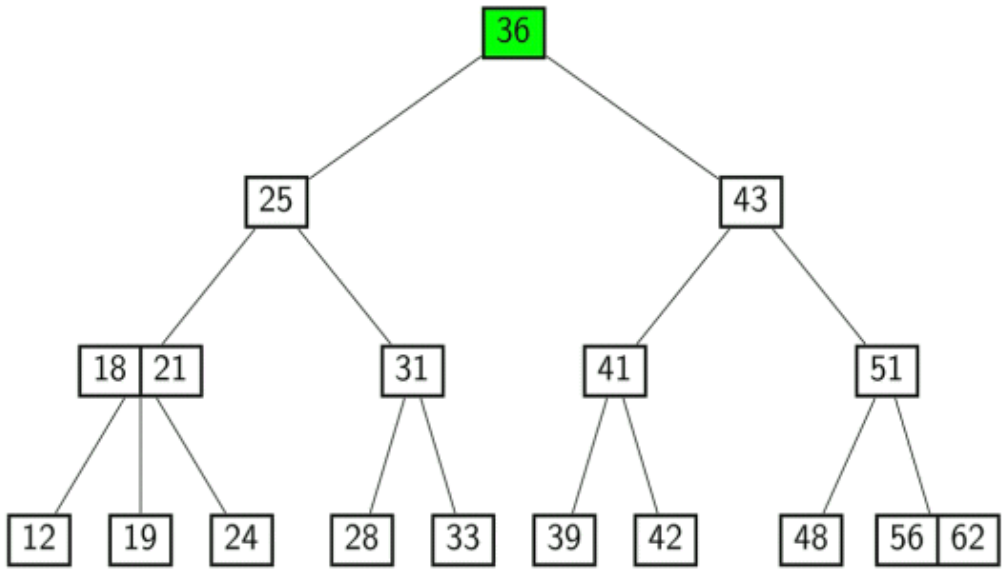
2-3 Tree Insertion

Example: *insert*(41)



2-3 Tree Insertion

Example: *insert*(41)



Deletion from a 2-3 Tree

As with BSTs and AVL trees, we first swap the KVP with its successor, so that we always delete from a leaf.

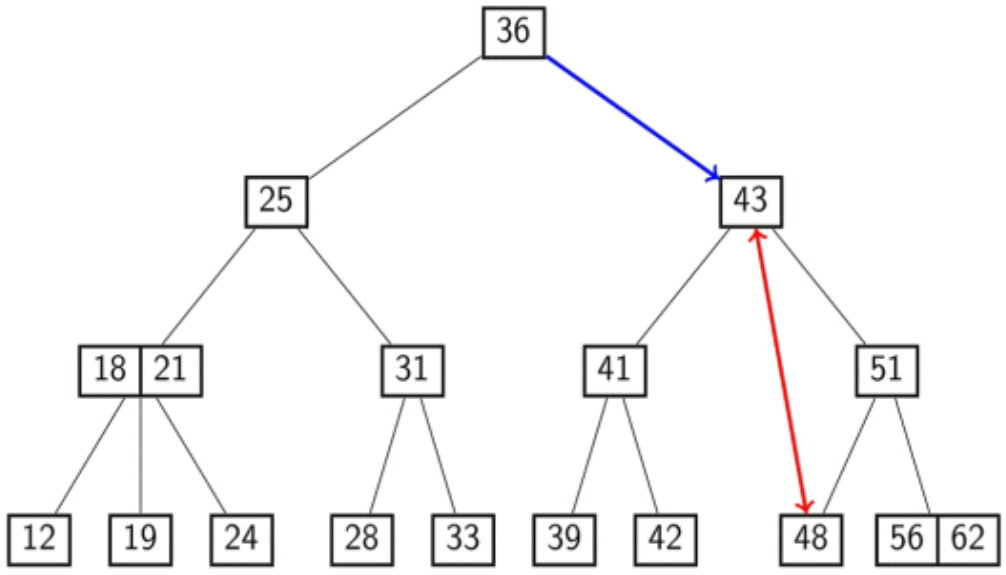
Say we're deleting KVP x from a node V:

- If V is a 2-node, just delete x.
- Elself V has a 2-node *immediate* sibling U, perform a *transfer*: Put the "intermediate" KVP in the parent between V and U into V, and replace it with the adjacent KVP from U.
- Otherwise, we *merge* V and a 1-node sibling U: Remove V and (recursively) delete the "intermediate" KVP from the parent, adding it to U.

Deletion/Search/Insert $\rightarrow O(h \log b)$

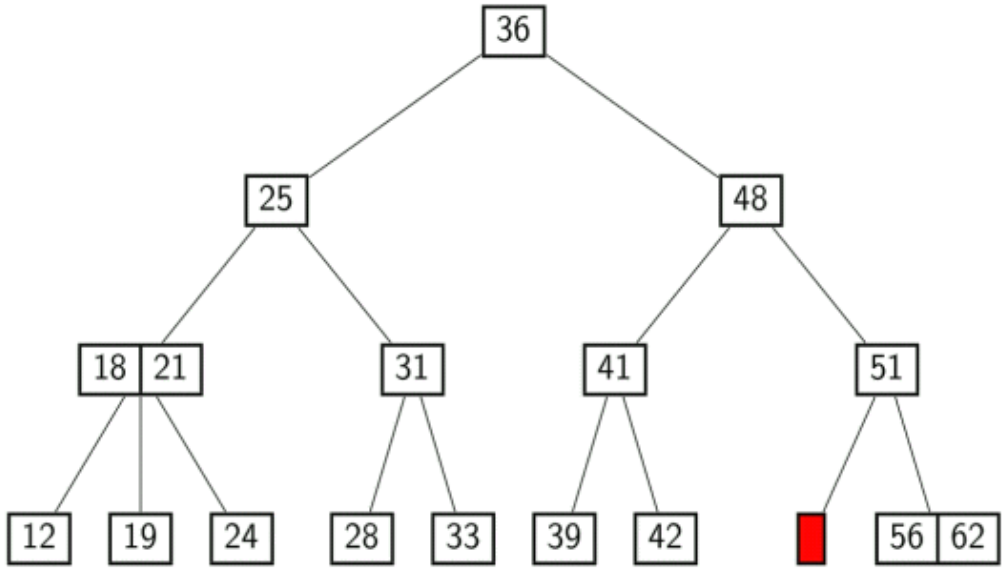
2-3 Tree Deletion

Example: *delete*(43)



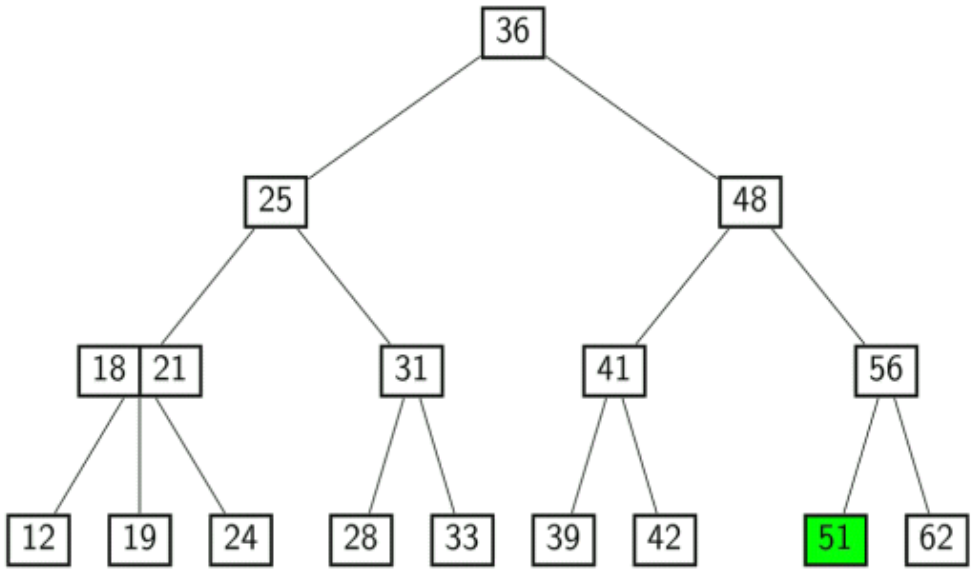
2-3 Tree Deletion

Example: *delete*(43)



2-3 Tree Deletion

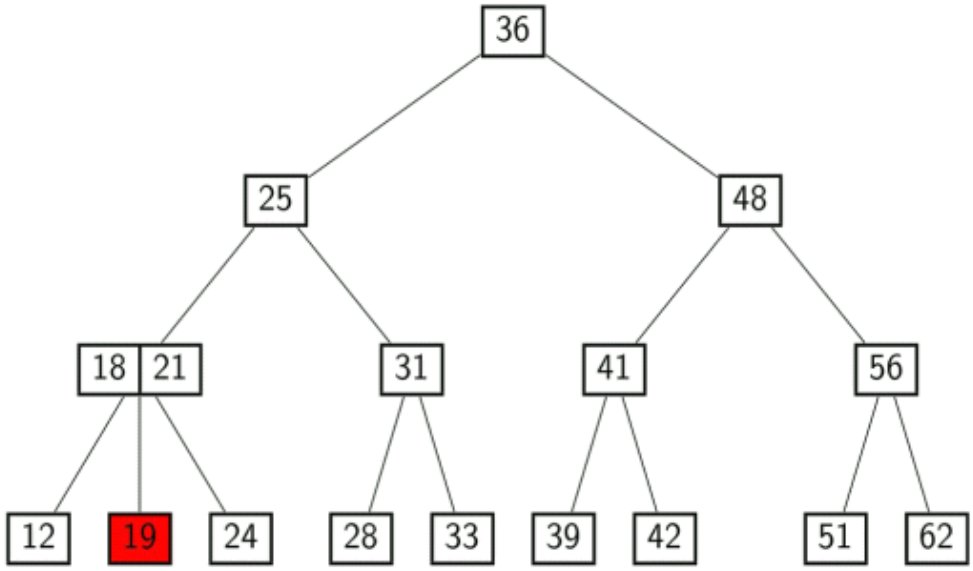
Example: *delete*(43)



Biedl,Maftuleac (SCS, UW) CS240 - Module 4 Winter 2015 25 / 1

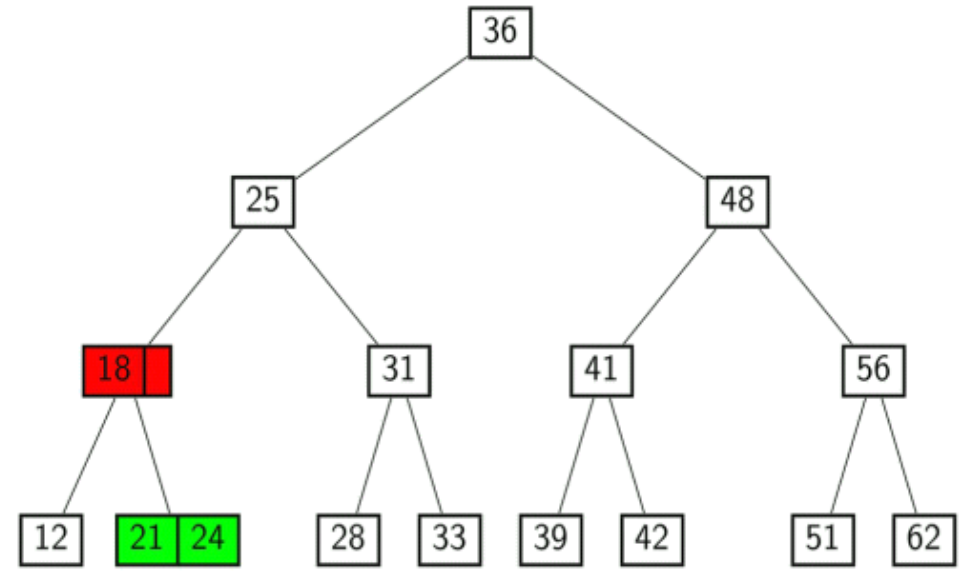
2-3 Tree Deletion

Example: *delete*(19)



2-3 Tree Deletion

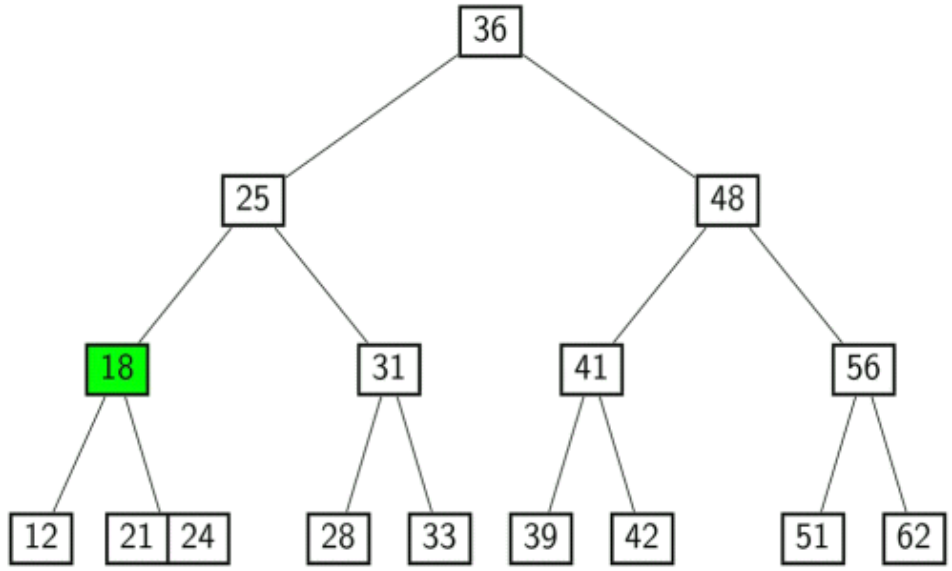
Example: *delete*(19)



Biedl,Maftuleac (SCS, UW) CS240 - Module 4 Winter 2015 25 / 1

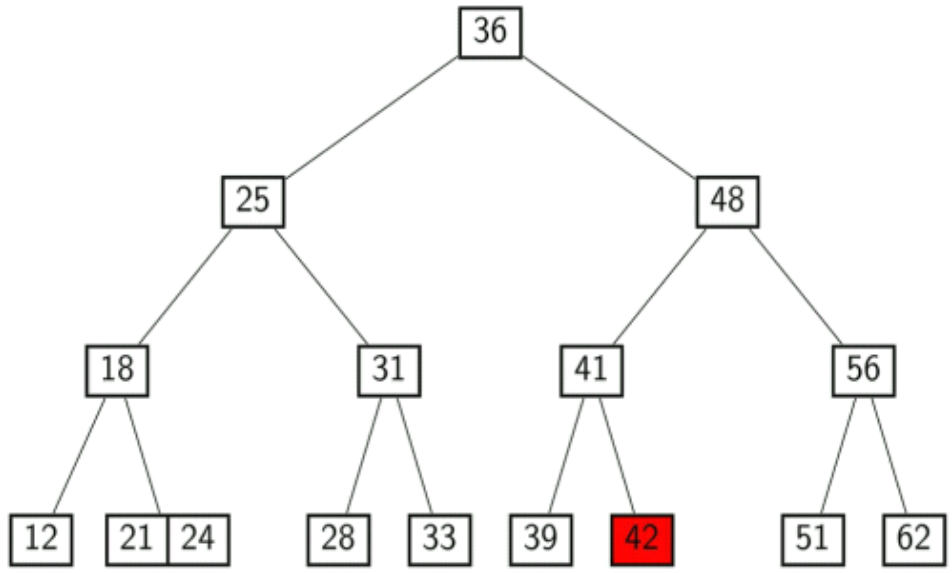
2-3 Tree Deletion

Example: *delete*(19)



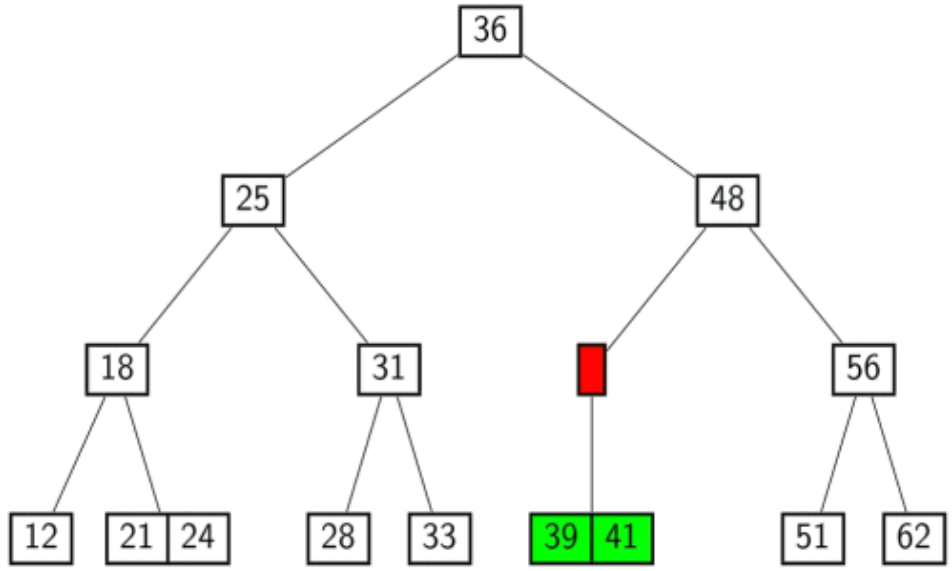
2-3 Tree Deletion

Example: *delete*(42)



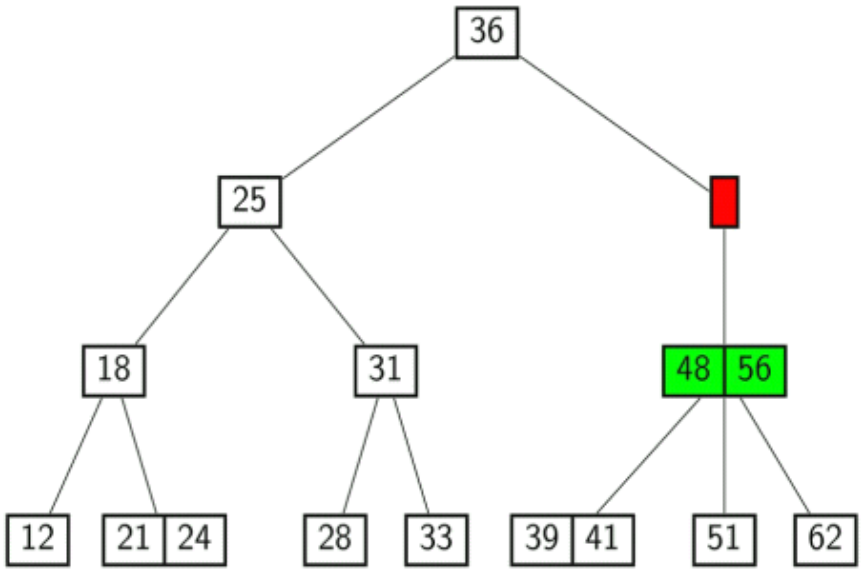
2-3 Tree Deletion

Example: *delete*(42)



2-3 Tree Deletion

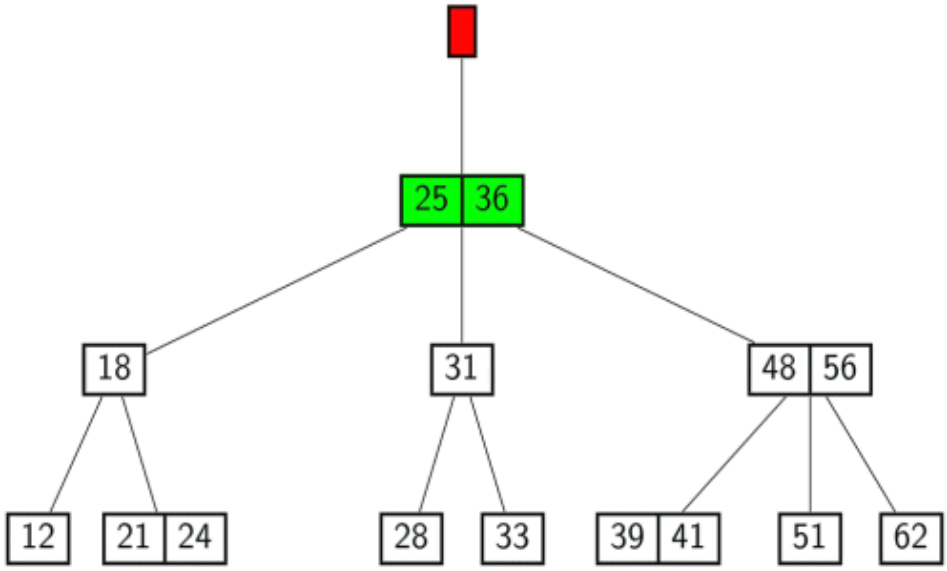
Example: *delete*(42)



Biedl,Maftuleac (SCS, UW) CS240 - Module 4 Winter 2015 25 / 1

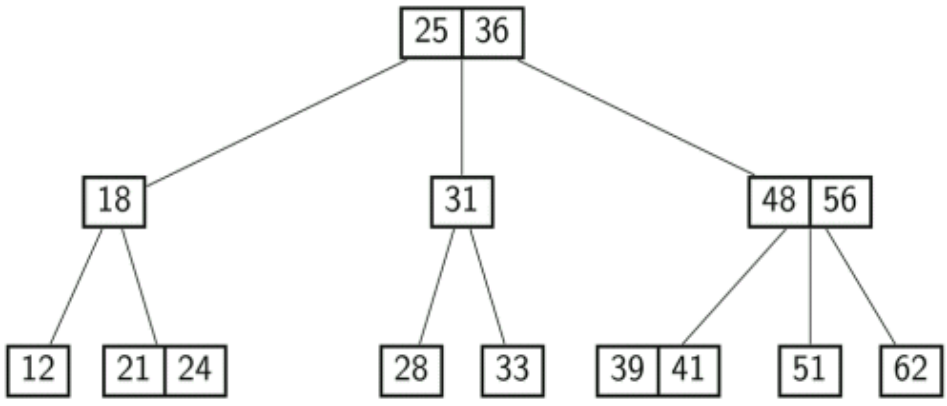
2-3 Tree Deletion

Example: *delete*(42)



2-3 Tree Deletion

Example: *delete*(42)



B-Trees

The 2-3 Tree is a specific type of B-tree:

A *B-tree of order M* is a search tree satisfying:

- Each node has between 2 and M links to (possibly empty) subtrees.
- If a node has k links, then it stores $k-1$ key-value pairs (KVPs).
- All nodes that store links to empty subtrees are on the same level.
- Each non-root node has links to at least $\lceil M/2 \rceil$ subtrees.
- The root has links to at least 2 subtrees.

Some people call this a B-tree of *min-size* $\lceil M/2 \rceil$, or a $(\lceil M/2 \rceil, M)$-tree.
A 2-3 tree has $M = 3$.

search, *insert*, *delete* work just like for 2-3 trees.

Height of a B-tree

What is the least number of KVPs in a height-h B-tree?

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

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

Therefore height of tree with n nodes is $\Theta\big((\log n)/(\log M)\big)$.

Analysis of B-tree operations

Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log M)$ search, insert, and delete.

Then *search*, *insert*, and *delete* work just like for 2-3 trees, and each require $\Theta(\textit{height})$ node operations.

Total cost is $O\left(\frac{\log n}{\log M} \cdot (\log M)\right) = O(\log n)$.

$h = \frac{\log n}{\log M}$ ← $b = M$ (order M B_tree)

Dictionaries in external memory

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

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

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

If M is small enough so an M-node fits into a single page, then a B-tree of order M only loads $\Theta\big((\log n)/(\log M)\big)$ pages.

This can result in a *huge* savings:
memory access is often the largest time cost in a computation.

B-tree variations

Max size M: Permitting one additional KVP in each node allows *insert* and *delete* to avoid *backtracking* via *pre-emptive splitting* and *pre-emptive merging*.

Red-black trees: Identical to a B-tree with minsize 1 and maxsize 3, but each 2-node or 3-node is represented by 2 or 3 binary nodes, and each node holds a "color" value of red or black.

B$^+$-trees: All KVPs are stored at the leaves (interior nodes just have keys), and the leaves are linked sequentially.

RAM Machine → preliminary operations take constant time

↳ if all data on RAM, then AVL & B-tree have the same complexity $O(\log n)$

↳ if data is huge ⇒ on disk ⇒ you need to have disk access

↳ if we count # of disk access

↳ AVL tree. $\Theta(\log(n))$ disk access in worst case

↳ B-tree, assuming M keys fit in a block

↳ $\Theta(h) \in \Theta\left(\frac{\log n}{\log m}\right)$ disk accesses per operation

Red-Black tree

↳ BST Realization of (2,4) B. tree

↳ search / insert / delete $\Theta(\log n)$

↳ less structured than BSTs

Search intensive → AVL is better

insert / delete → RB tree is better