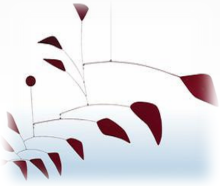


EECS 4101-5101
Advanced Data Structures

Shahin Kamali

Topic 2c 2-3 Trees and B Trees

York University

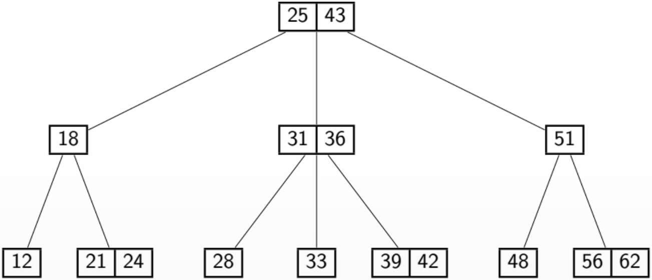
Picture is from the cover of the textbook CLRS.

Overview

- Introduction to 2-3 trees
- b-trees as an extension of 2-3 trees
- Dictionary Operations on 2-3 trees and b-trees

2-3 Trees

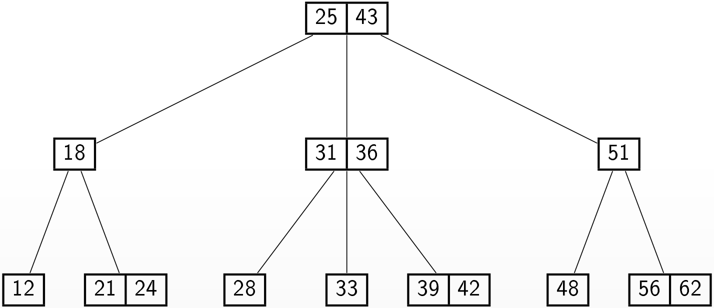
- A **ternary tree** is a tree in which each node has at most 3 children.
- A 2-3 Tree is a ternary tree like a BST with additional structual properties:
    - Every node either contains *one KVP* and *two children*, or *two KVPs* and *three children*.
    - All the leaves are at the same level (A leaf is a node with empty children.)



Search in a 2-3 tree

- 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.

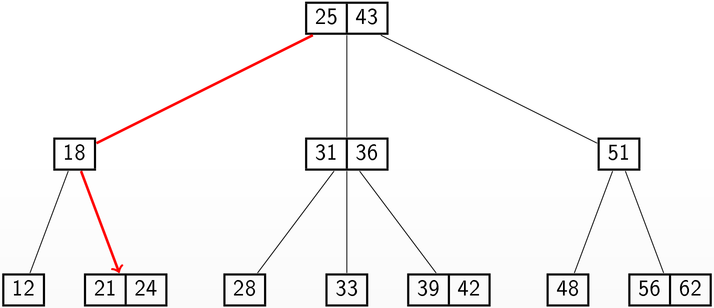
search(21)



Search in a 2-3 tree

- 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.

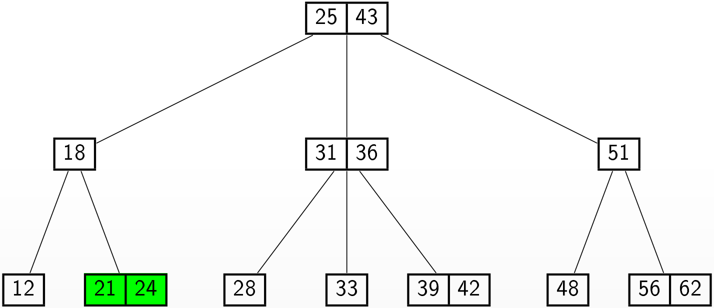
search(21)



Search in a 2-3 tree

- 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.

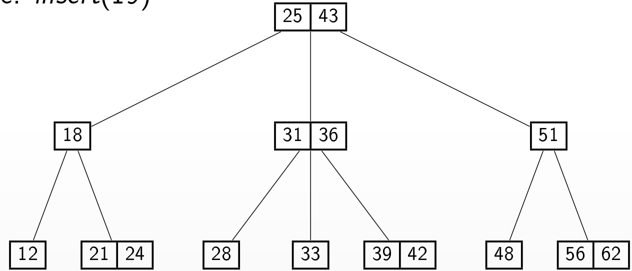
search(21)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

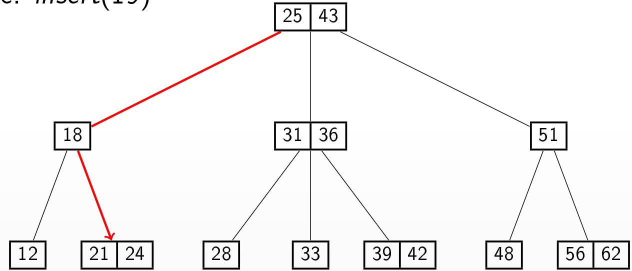
Example: *insert*(19)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

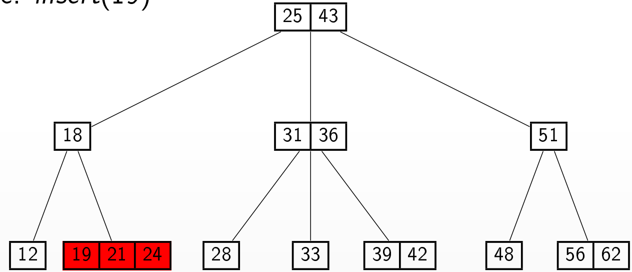
Example: *insert*(19)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

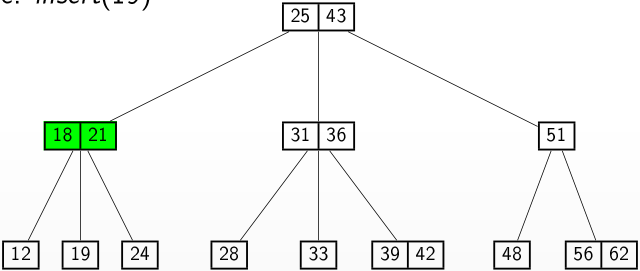
Example: *insert*(19)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

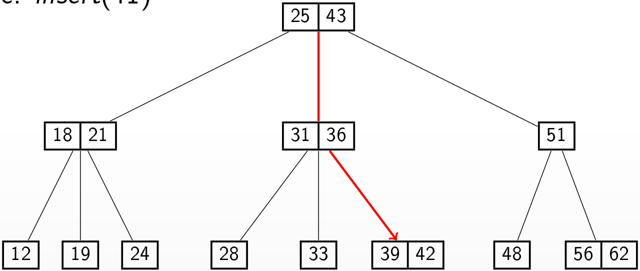
Example: *insert*(19)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

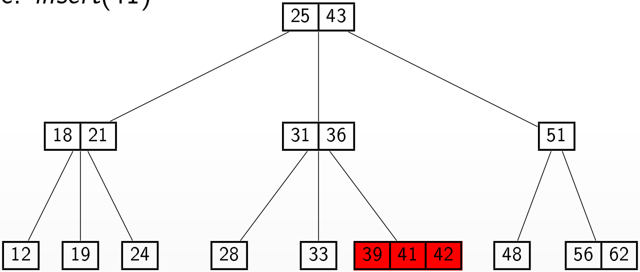
Example: *insert*(41)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

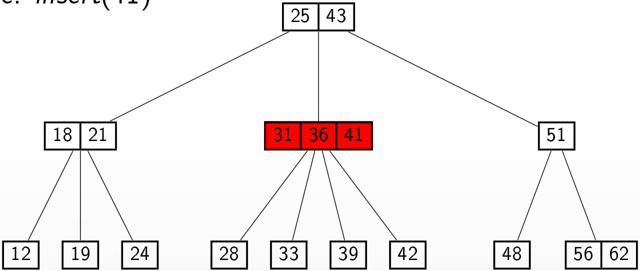
Example: *insert*(41)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

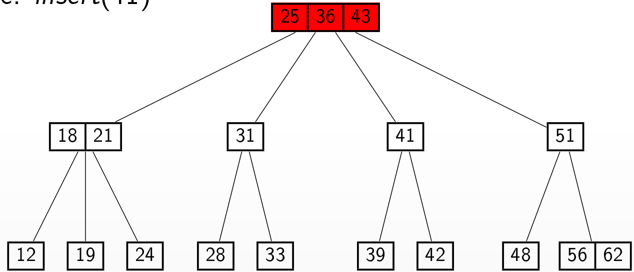
Example: *insert*(41)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

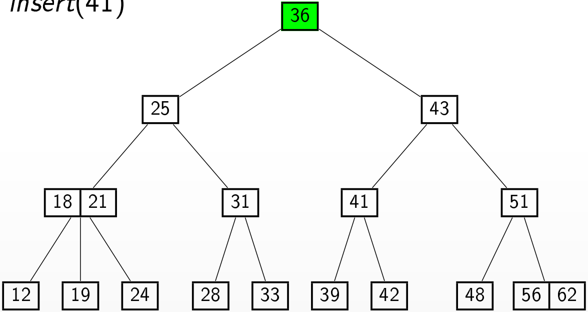
Example: *insert*(41)



Insertion in a 2-3 tree

- Inserting a new KVP to 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.

Example: *insert*(41)

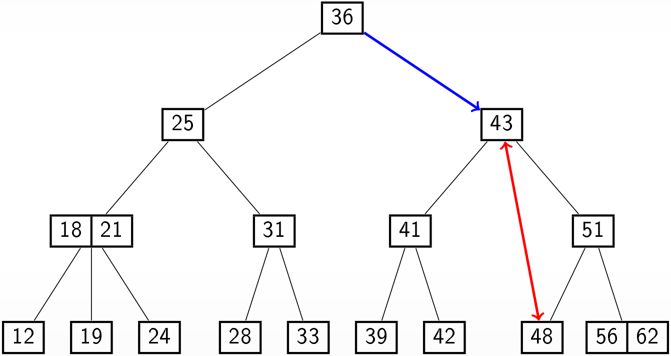


Deletion from a 2-3 Tree

- As with BSTs and AVL trees, we first swap the KVP with its successor → this way 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.
    - Else If 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.

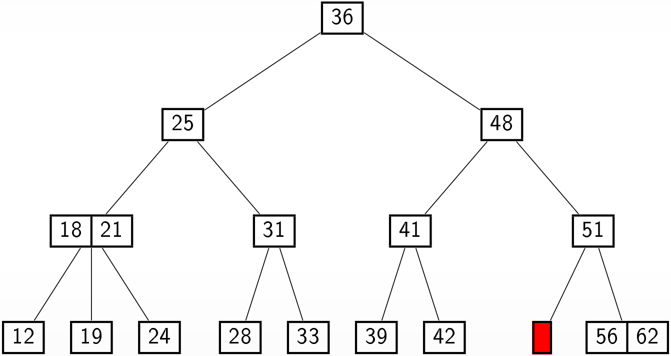
2-3 Tree Deletion

Example: *delete*(43)



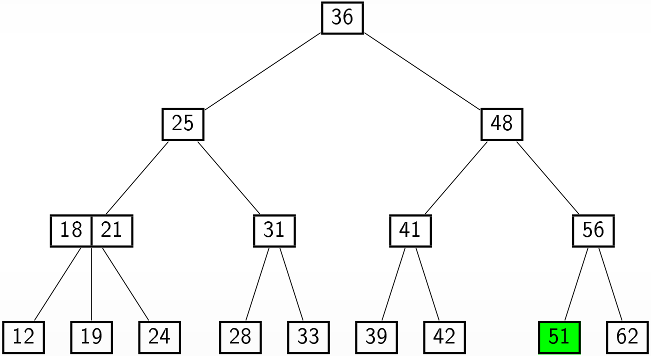
2-3 Tree Deletion

Example: *delete*(43)



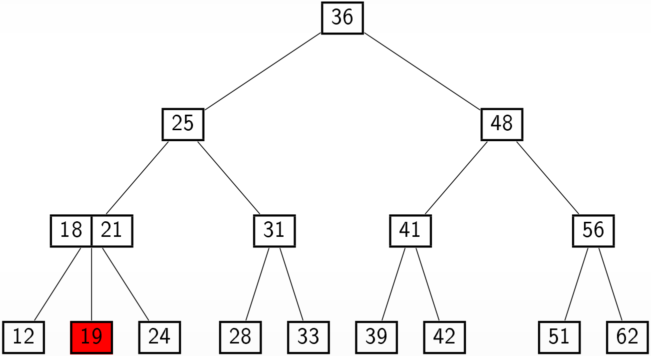
2-3 Tree Deletion

Example: *delete*(43)



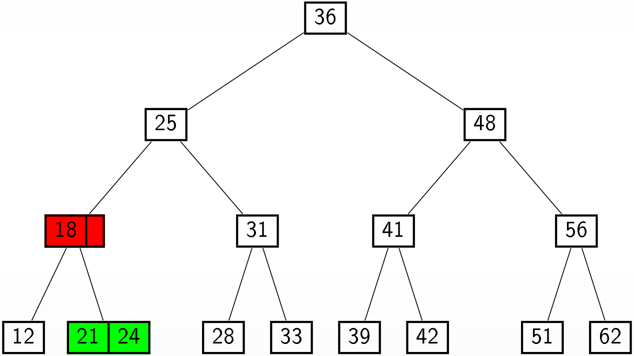
2-3 Tree Deletion

Example: *delete*(19)



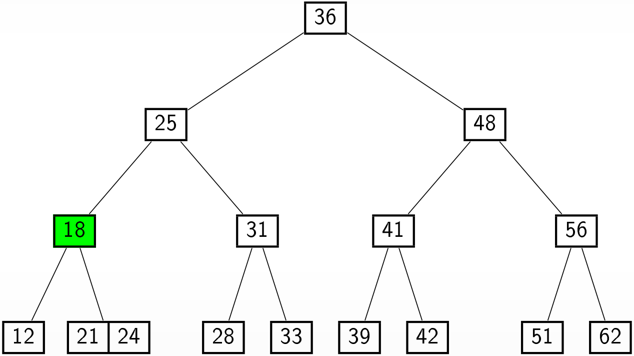
2-3 Tree Deletion

Example: *delete*(19)



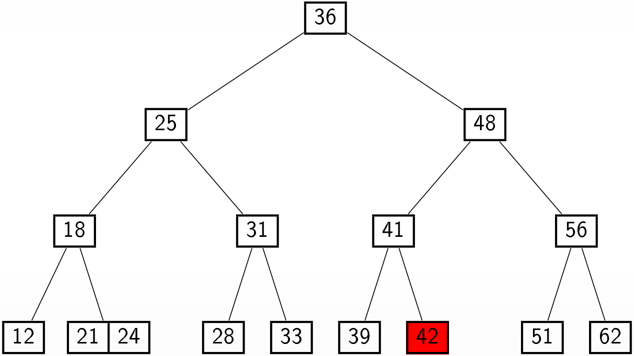
2-3 Tree Deletion

Example: *delete*(19)



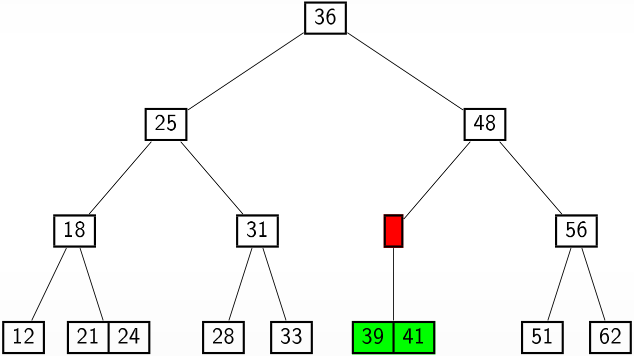
2-3 Tree Deletion

Example: *delete*(42)



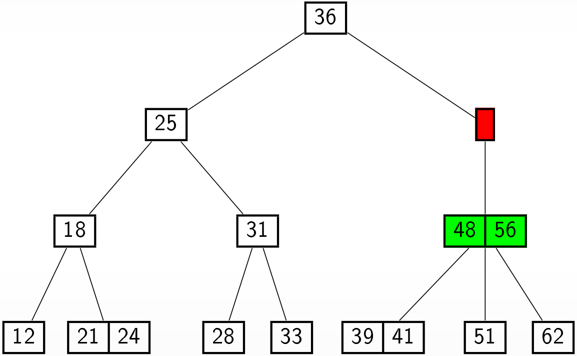
2-3 Tree Deletion

Example: *delete*(42)



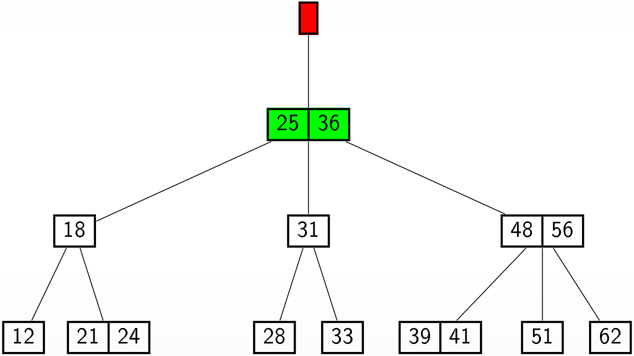
2-3 Tree Deletion

Example: *delete*(42)



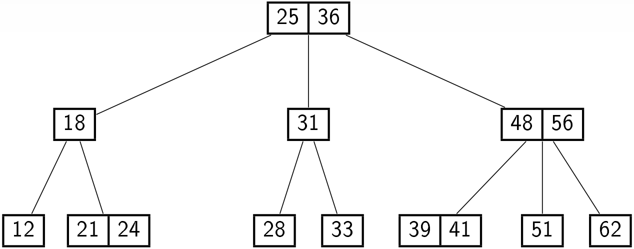
2-3 Tree Deletion

Example: *delete*(42)



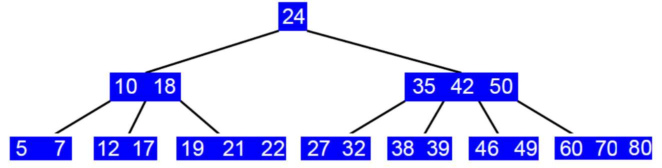
2-3 Tree Deletion

Example: *delete*(42)



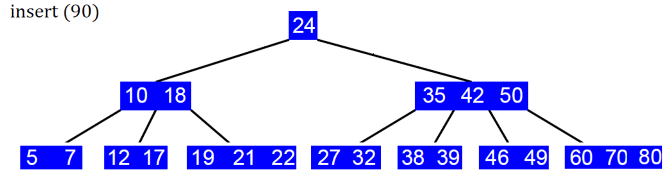
B-Trees

- A *B-tree of minsize d* is a search tree satisfying:
  - Each node contains at most $2d$ KVPs.
    Non-root nodes contain at least d KVPs (root can have 1 or more).
  - All the leaves are at the same level.
- Some people call this a B-tree of order $(2d + 1)$, or a $(d + 1, 2d + 1)$-tree.
  - The 2-3 Tree is a specific type of B-tree with $d = 1$.
  - Here is a tree with $d = 2$:



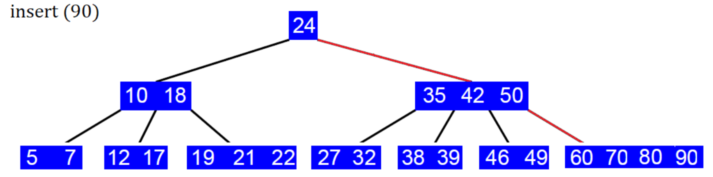
B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).



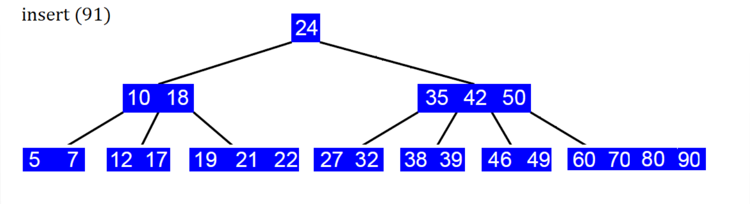
B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
    - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).



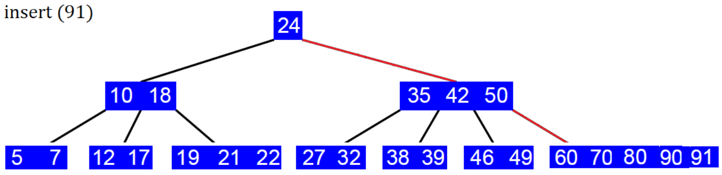
B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
    - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).



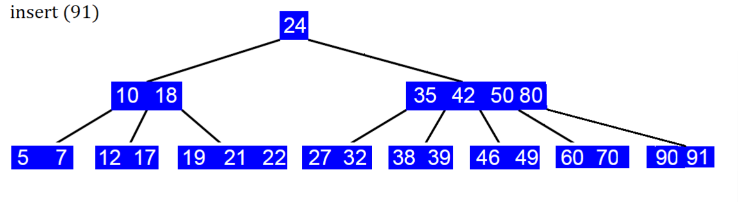
B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).



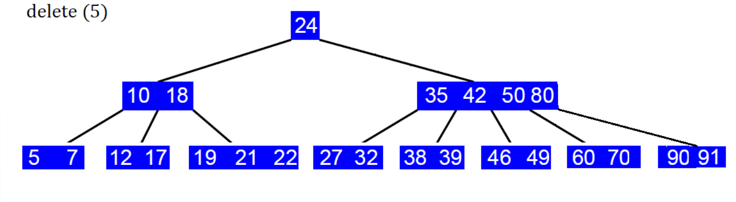
B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
    - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).



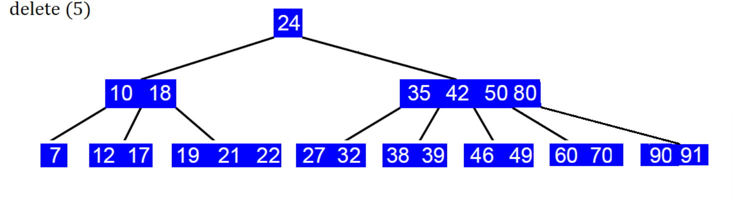
B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
    - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
    - For *delete*, take the following steps:
        - if there is no underflow after delete, do nothing.
        - else, check if any direct sibling has an extra key; if it does, borrow a key from the parent and let the parent borrow a key from the sibling (update the pointer after).
        - else, merge two nodes by creating a node containing the underflowed node (with $d-1$ keys), the key at parent (1 key), and direct sibling (d keys). The new key will have size $2d$.



B-Tree Operations

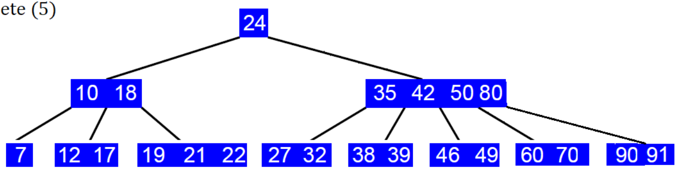
- *search*, *insert*, *delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, take the following steps:
    - if there is no underflow after delete, do nothing.
    - else, check if any direct sibling has an extra key; if it does, borrow a key from the parent and let the parent borrow a key from the sibling (update the pointer after).
    - else, merge two nodes by creating a node containing the underflowed node (with $d-1$ keys), the key at parent (1 key), and direct sibling (d keys). The new key will have size $2d$.



B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
    - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
    - For *delete*, take the following steps:
        - if there is no underflow after delete, do nothing.
        - else, check if any direct sibling has an extra key; if it does, borrow a key from the parent and let the parent borrow a key from the sibling (update the pointer after).
        - else, merge two nodes by creating a node containing the underflowed node (with $d-1$ keys), the key at parent (1 key), and direct sibling (d keys). The new key will have size $2d$.

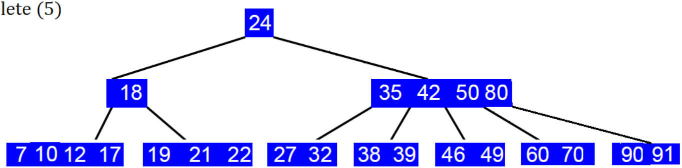
delete (5)



B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, take the following steps:
    - if there is no underflow after delete, do nothing.
    - else, check if any direct sibling has an extra key; if it does, borrow a key from the parent and let the parent borrow a key from the sibling (update the pointer after).
    - else, merge two nodes by creating a node containing the underflowed node (with $d-1$ keys), the key at parent (1 key), and direct sibling (d keys). The new key will have size $2d$.

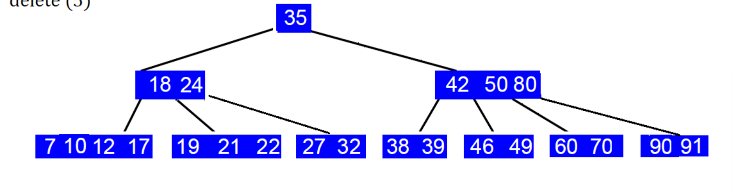
delete (5)

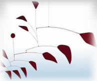


B-Tree Operations

- *search*, *insert*, *delete* work just like for 2-3 trees.
  - As before, *insert might* result in overflow, in which case we divide the node in two nodes and send parent upward (and repeat recursively).
  - For *delete*, take the following steps:
    - if there is no underflow after delete, do nothing.
    - else, check if any direct sibling has an extra key; if it does, borrow a key from the parent and let the parent borrow a key from the sibling (update the pointer after).
    - else, merge two nodes by creating a node containing the underflowed node (with $d-1$ keys), the key at parent (1 key), and direct sibling (d keys). The new key will have size $2d$.

delete (5)





Height of a B-tree

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

Level	#Nodes is $\geq$	Node size is $\geq$	KVPs is $\geq$
0	1	1	1
1	2	d	$2d$
2	$2(d+1)$	d	$2d(d+1)$
3	$2(d+1)^2$	d	$2d(d+1)^2$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
h	$2(d+1)^{h-1}$	d	$2d(d+1)^{h-1}$

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

$$\rightarrow \log(n+1) \geq 1 + h\log(d+1) \rightarrow h \leq \frac{\log(n+1)-1}{\log(d+1)} = O\left(\frac{\log n}{\log d}\right)$$

Analysis of B-tree operations

- Assume each node stores its KVPs and child-pointers in a dictionary that supports $O(\log d)$ search, insert, and delete.
- Then *search*, *insert*, and *delete* work just like for 2-3 trees, and each require $\Theta(\mathit{height})$ node operations.
- Total cost is $O\left(\frac{\log n}{\log d} \cdot (\log d)\right) = O(\log n)$.

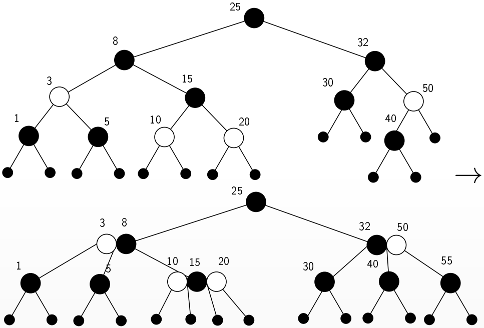
Dictionaries in external memory

- 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 for a single insert/delete/search operation.
  - If d is small enough so a $2d$-node fits into a single page,
    then a B-tree of minsize d only loads $\Theta\big((\log n)/(\log d)\big)$ pages.
  - This can result in a *huge* savings:
    memory access is often the largest time cost in a computation.
  - This was the main reason for the introduction of B-trees by Bayer and McCreight in 1970.

B-trees vs Red-Black Trees

Red-black trees: Identical to a B-tree with minsize 1 and maxsize 3

- Given a red-black tree, merge each black node with its red children; maintain one black node at each node of the B-tree. Why is the result a B-tree?



B-tree variations

Max size $2d+1$: Permitting one additional KVP in each node allows *insert* and *delete* to avoid *backtracking* via *pre-emptive splitting* and *pre-emptive merging*.

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