

22C:244

B-trees

Dr. Ramon Lawrence
University of Iowa
ramon-lawrence@uiowa.edu
<http://www.cs.uiowa.edu/~rlawrenc>

22C:244 - Dr. Ramon Lawrence

B-Trees and Indexing Overview

We have seen previously how multi-level indexes can improve the performance of searches.

One of the challenges in creating multi-level indexes is maintaining the index in the presence of inserts and deletes.

We will briefly discuss a historical indexing method call ISAM, and then learn about B+-trees which is the most common form of index used in database systems today.

Both ISAM and B+-trees are multi-level indexes, but B+-trees are more capable of handling modifications to the data file.

Page 2

Index-Sequential Access Method (ISAM)

ISAM was historically used in IBM mainframes. It is a multi-level index with the following properties:

- ◆ The data file is ordered on the index search key.
- ◆ The index is built once by examining the data file.
- ◆ The index itself looks like a B-tree with interior nodes containing search keys and pointers.

The problem with ISAM is that the index is created once and is **not** updated as the data file is updated.

- ◆ Periodically the index is rebuilt.
- ◆ Insertion is a major problem as the system must use overflow blocks that are not indexed.
 - ⇒ Performance suffers if there are a lot of overflow blocks.

Page 3

B-trees Introduction

B-trees, and more specifically, B+-trees are used as dynamic, multi-level indexes.

- ◆ We start by describing the B-tree data structure itself.

A **B-tree** is a search tree where each node has $\geq n$ data values and $\leq 2n$, where we chose n for our particular tree.

- ◆ Each key in a node is stored in an array.
 - ⇒ $\text{key}[0]$ is the first key, $\text{key}[1]$ is the second key, ..., $\text{key}[2n-1]$ is the $2n^{\text{th}}$ key
- ◆ There is also an array of pointers to children nodes:
 - ⇒ $\text{child}[0], \text{child}[1], \text{child}[2], \dots, \text{child}[2n]$
- ◆ A node may not contain all key values (and the corresponding children pointers).
 - ⇒ # of children = # of keys + 1
 - ⇒ If a node has k data values, the node has $k+1$ children.

A B-tree is **balanced** as every leaf has the same depth. Page 4

B-trees

Node Properties

Each node has the property:

- ◆ $\text{key}[0] < \text{key}[1] < \text{key}[2] < \dots < \text{key}[2n-1]$ (key array is sorted)
- ◆ A node may not contain all key values (and the corresponding children pointers).
 - $\Rightarrow \# \text{ of children} = \# \text{ of keys} + 1$
- ◆ For any $\text{key}[i]$:
 - $\Rightarrow 1) \text{key}[i] > \text{all entries in subtree pointed to by child}[i]$
 - $\Rightarrow 2) \text{key}[i] \leq \text{all entries in subtree pointed to by child}[i+1]$
- ◆ Recursive definition: each subtree pointed to by $\text{child}[i]$ is also a B-tree.
- ◆ A node must contain at least n keys and at most $2n$ keys where n is the order of the tree.

B-trees

Order Debate

There is an interesting debate on how to define an **order** of a B-tree. The original definition was the one given:

- ◆ The **order n** is the minimum # of keys in a node. The maximum number is $2n$.

However, many authors want to have a B-tree where the maximum # of keys in a node is odd.

- ◆ This is not possible by the above definition.

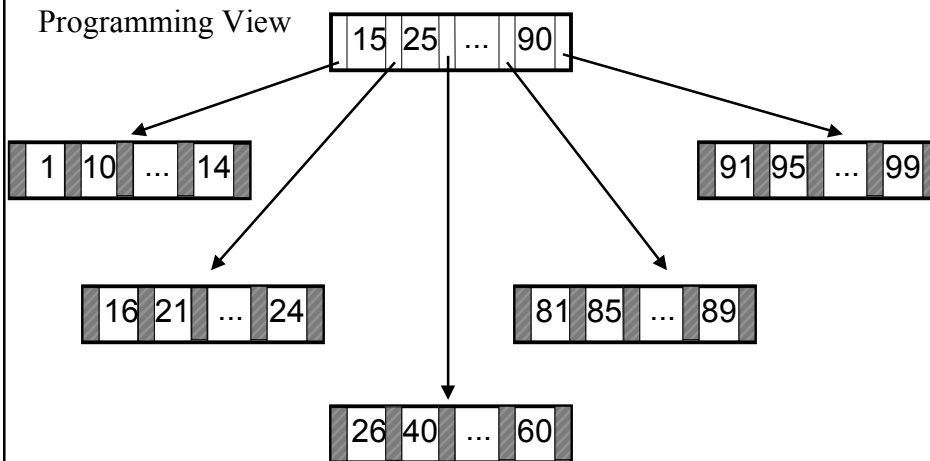
Consequently, these authors define order as the maximum # of keys in a node (instead of the minimum).

- ◆ Further, some use maximum # of pointers instead of keys.

Bottom line: B-trees with an odd maximum # of keys will be avoided in the class.

- ◆ The minimum # of nodes for an odd maximum n will be $\lfloor n/2 \rfloor$.

B-trees Example



Page 7

2-3 Trees Introduction

A **2-3 tree** is a B-tree where each node has either **1** or **2** data values and **2** or **3** children pointers.

- ◆ It is a special case of a B-tree.

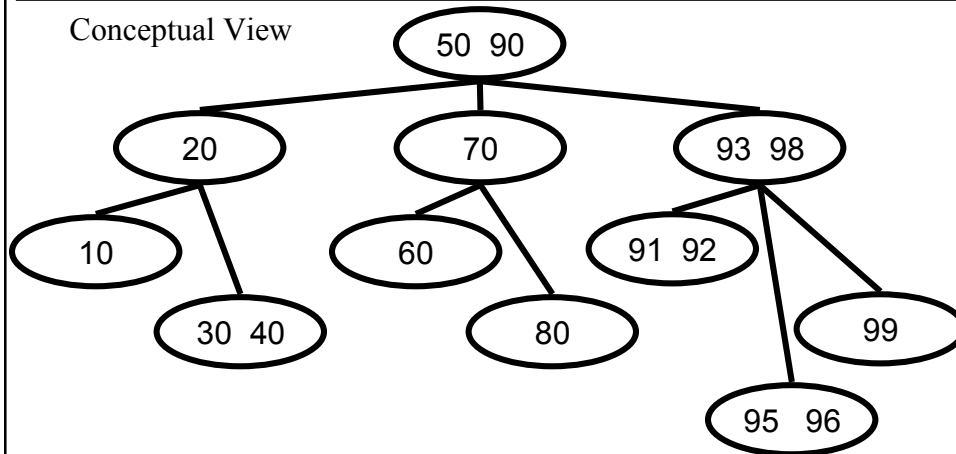
Fact:

- ◆ A 2-3 tree of height h always has at least as many nodes as a full binary tree of height h .
 ⇒ That is, a 2-3 tree will always have at least $2^h - 1$ nodes.

Page 8

2-3 Search Tree Example

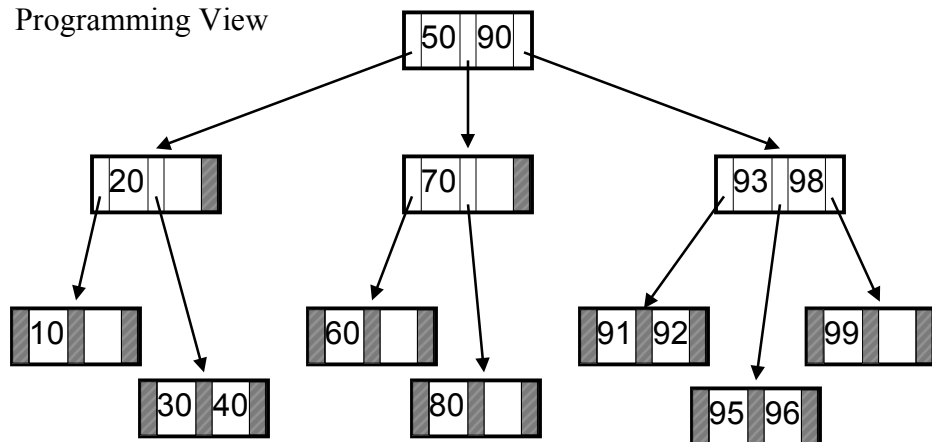
Conceptual View



Page 9

2-3 Tree Example

Programming View



Page 10

Searching a 2-3 Tree

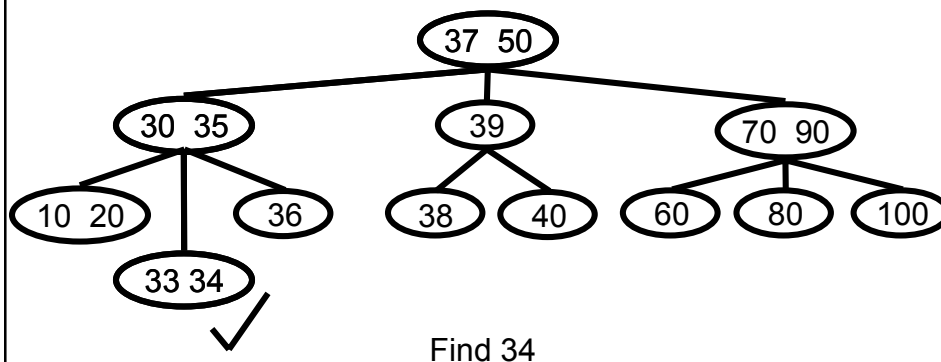
Searching a 2-3 tree is similar to searching a binary search tree.

Algorithm:

- ◆ Start at the root which begins as the curNode.
- ◆ If curNode contains the search key we are done, and have found the search key we were looking for.
- ◆ A 2-node contains one key:
 - ⇒ If search key < key[0], go left (child[0]) otherwise go right (child[1])
- ◆ A 3-node contains two key values:
 - ⇒ If search key < key[0], go left with first child pointer (child[0])
 - ⇒ else if search key < key[1] go down middle child pointer (child[1])
 - ⇒ else (search key >= key[1]) go right with last child pointer (child[2])
- ◆ If we encounter a NULL pointer, then we are done and the search failed.

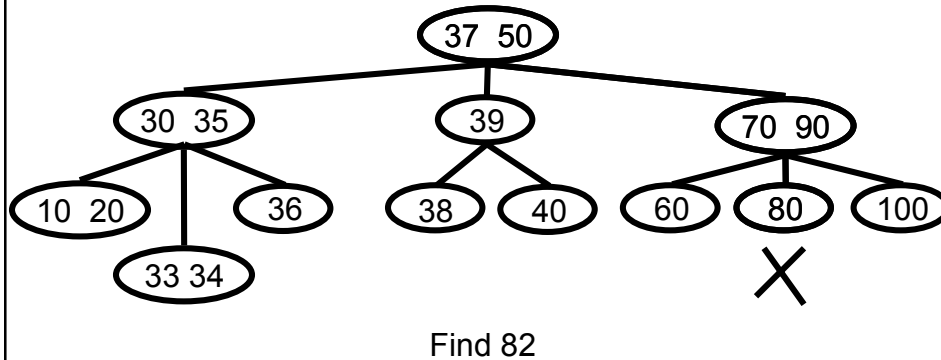
Page 11

Searching a 2-3 Tree Example #1



Page 12

Searching a 2-3 Tree Example #2



Insertion into a 2-3 Tree

Algorithm:

- ◆ Find the leaf node where the new key belongs.
- ◆ This insertion node will contain either a single key or two keys.
- ◆ If the node contains 1 key, insert the new key in the node (in the correct sorted order).
- ◆ If the node contains 2 keys:
 - ⇒ Insert the node in the correct sorted order.
 - ⇒ The node now contains 3 keys (overflow).
 - ⇒ Take the middle key and promote it to its parent node. (split node)
 - ⇒ If the parent node now has more than 3 keys, repeat the procedure by promoting the middle node to its parent node.
- ◆ This promotion procedure continues until:
 - ⇒ Some ancestor has only one node, so overflow does not occur.
 - ⇒ All ancestors are “full” in which case the current root node is split into two nodes and the tree “grows” by one level.

Insertion into a 2-3 Tree Splitting Algorithm

Splitting Algorithm:

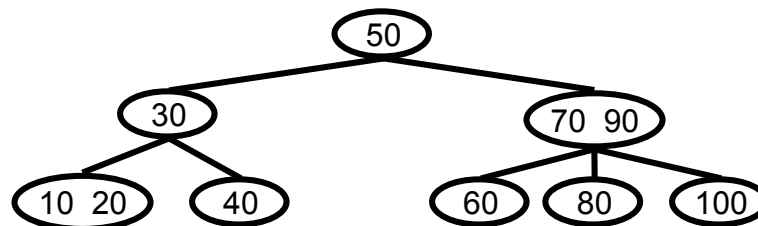
- ◆ Given a node with overflow (more than 2 keys in this case), we split the node into two nodes each having a single key.
- ◆ The middle value (in this case $\text{key}[1]$) is passed up to the parent of the node.
 - ⇒ This, of course, requires *parent* pointers in the 2-3 tree.
- ◆ This process continues until we find a node with sufficient room to accommodate the node that is being percolated up.
- ◆ If we reach the root and find it has 2 keys, then we split it and create a new root consisting of the “middle” node.

The splitting process can be done in logarithmic time since we split at most one node per level of the tree and the depth of the tree is logarithmic in the number of nodes in the tree.

- ◆ Thus, 2-3 trees provide an efficient height balanced tree.

Page 15

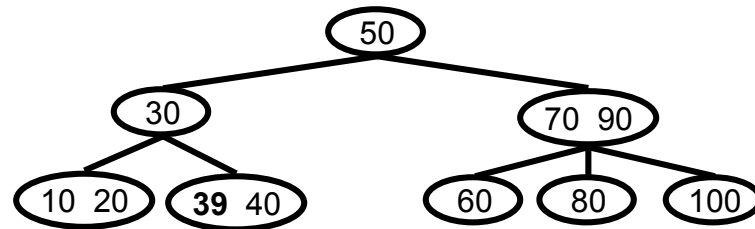
Insertion Examples



Insert 39

Page 16

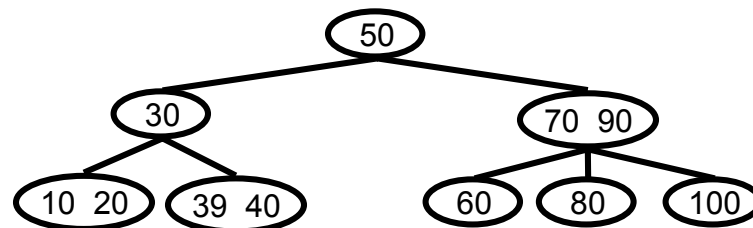
Insertion Examples



Done!

Page 17

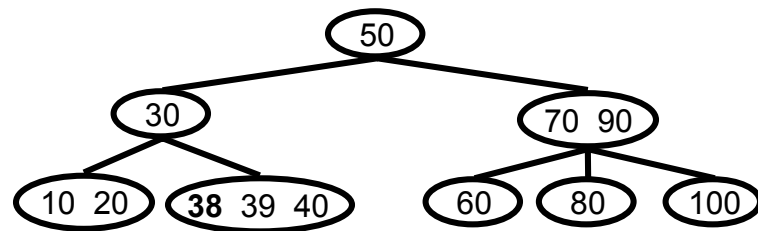
Insertion Examples



Insert 38

Page 18

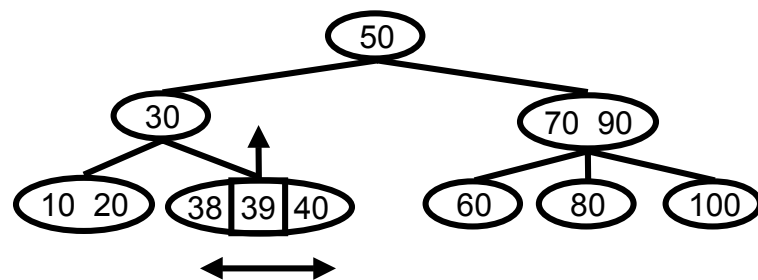
Insertion Examples



Insert 38

Page 19

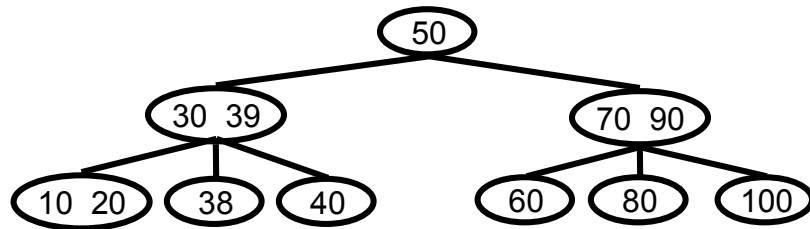
Insertion Examples



Push up, split apart

Page 20

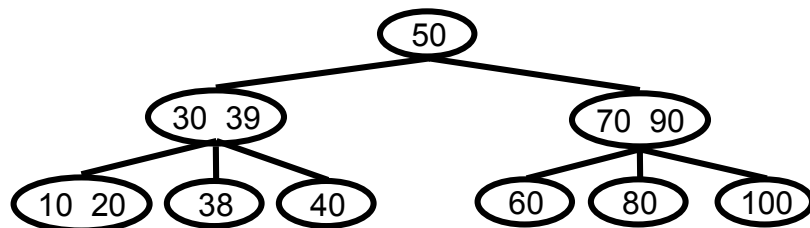
Insertion Examples



Done!

Page 21

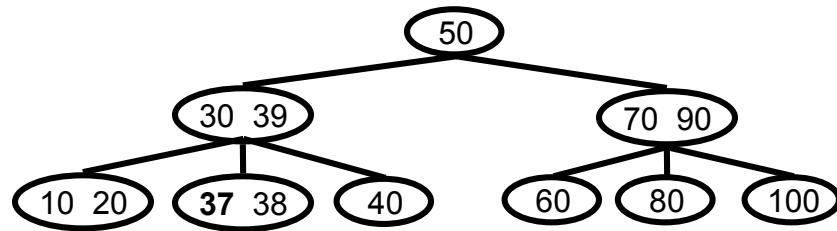
Insertion Examples



Insert 37

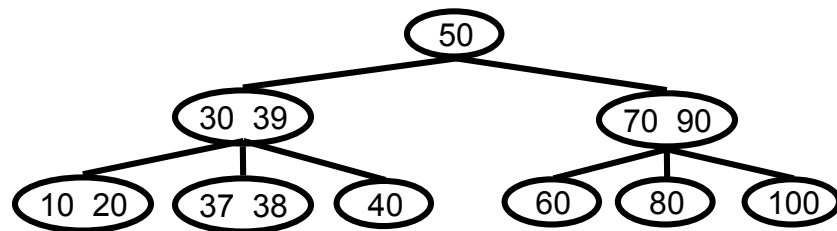
Page 22

Insertion Examples



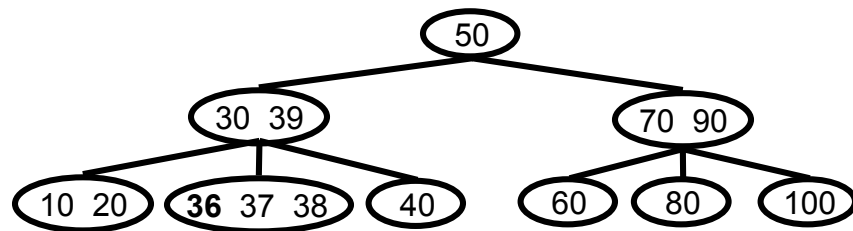
Done!

Insertion Examples



Insert 36

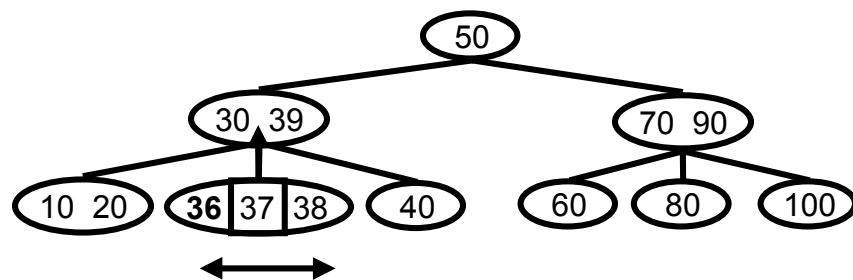
Insertion Examples



Insert 36

Page 25

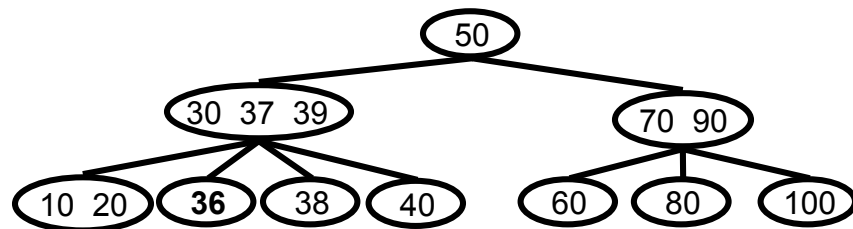
Insertion Examples



Push up, split apart

Page 26

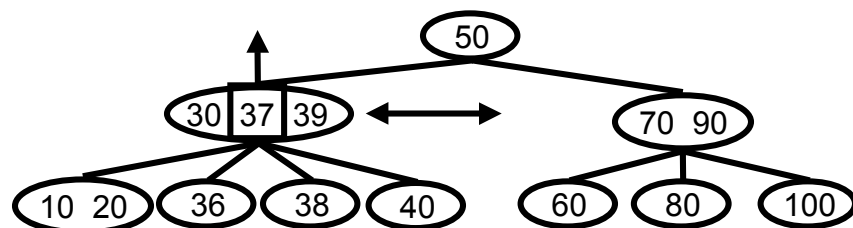
Insertion Examples



Need to go further up the tree to resolve overcrowding

Page 27

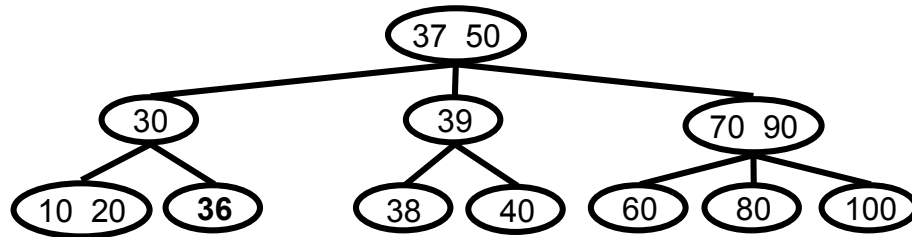
Insertion Examples



Push up, split apart

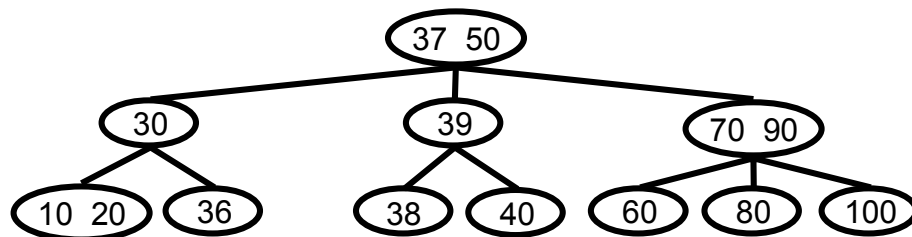
Page 28

Insertion Examples



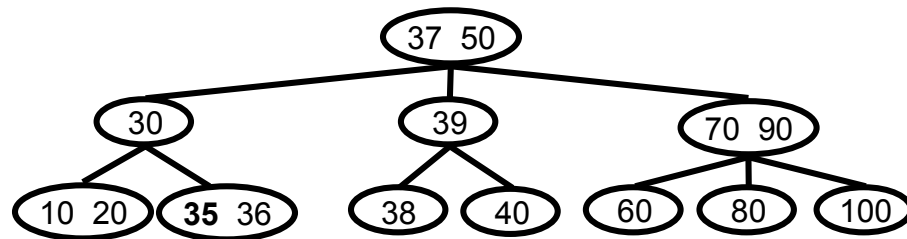
Done!

Insertion Examples



Insert 35

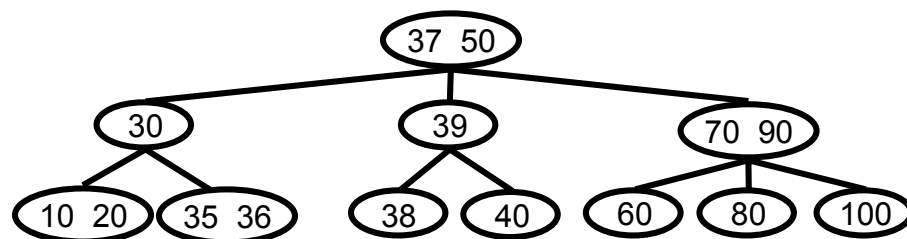
Insertion Examples



Insert 35

Page 31

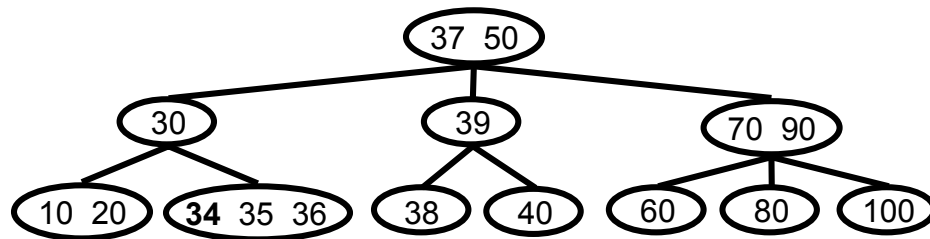
Insertion Examples



Insert 34

Page 32

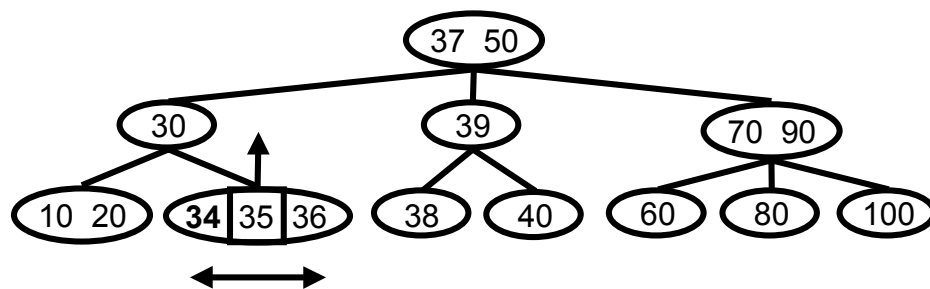
Insertion Examples



Insert 34

Page 33

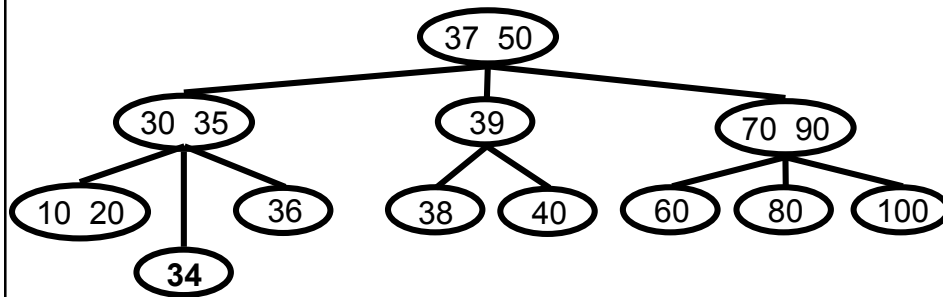
Insertion Examples



Push up, split apart

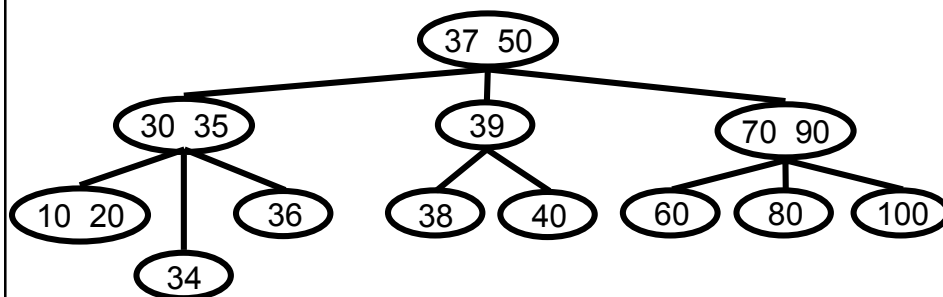
Page 34

Insertion Examples



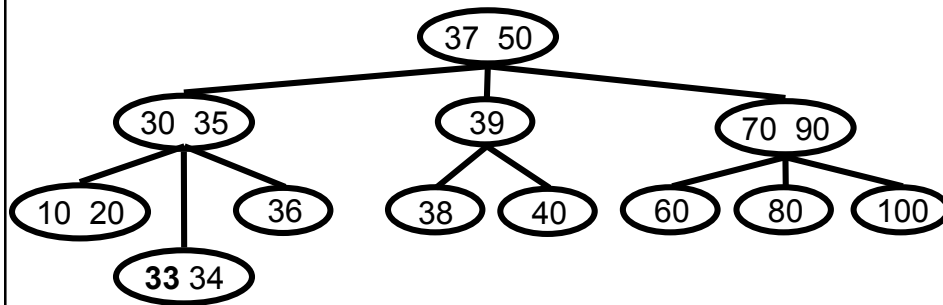
Done!

Insertion Examples



Insert 33

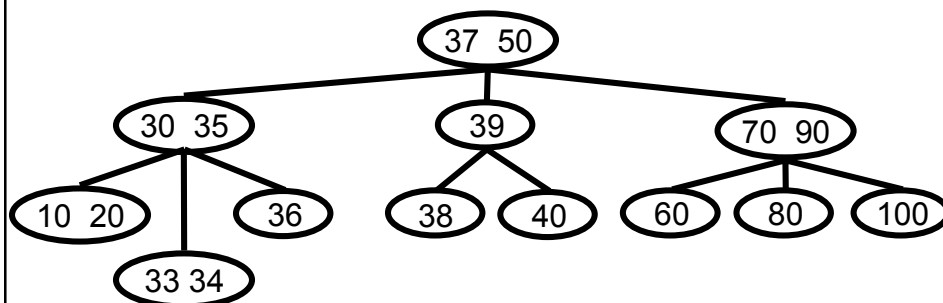
Insertion Examples



Done!

Page 37

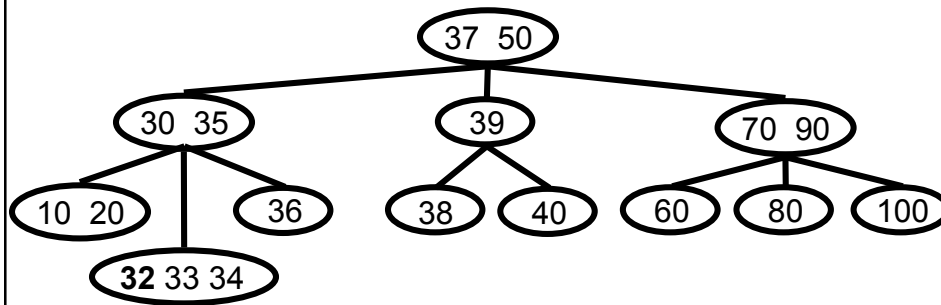
Insertion Examples



Insert 32

Page 38

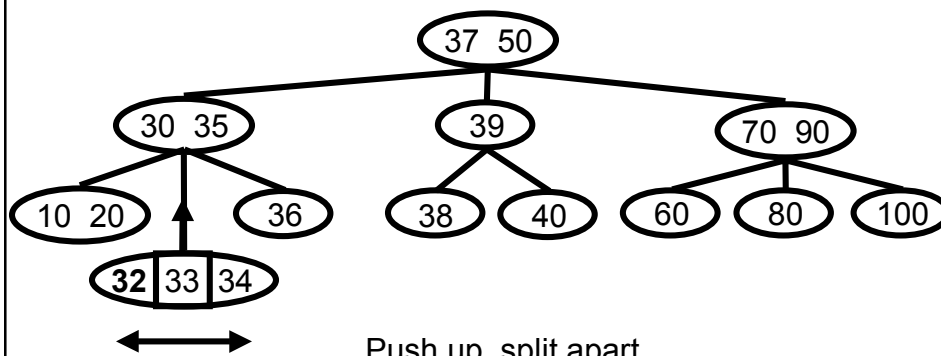
Insertion Examples



Insert 32

Page 39

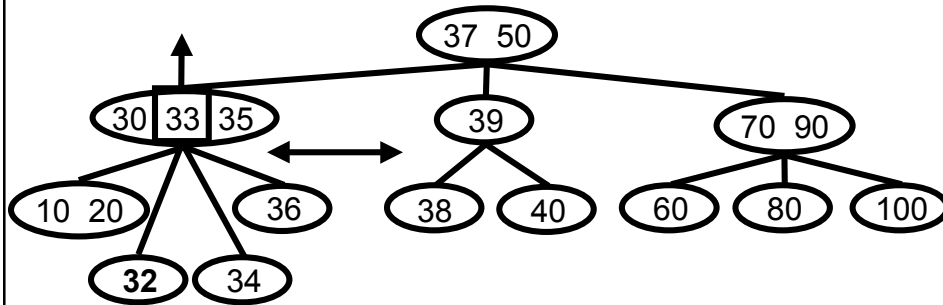
Insertion Examples



Push up, split apart

Page 40

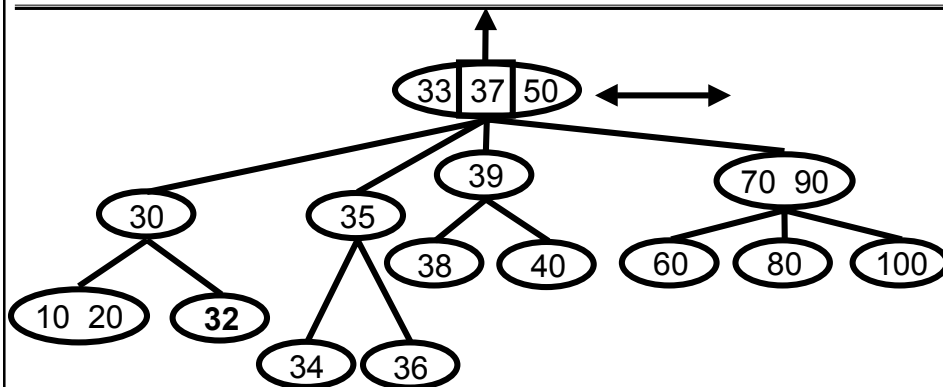
Insertion Examples



Push up, split apart

Page 41

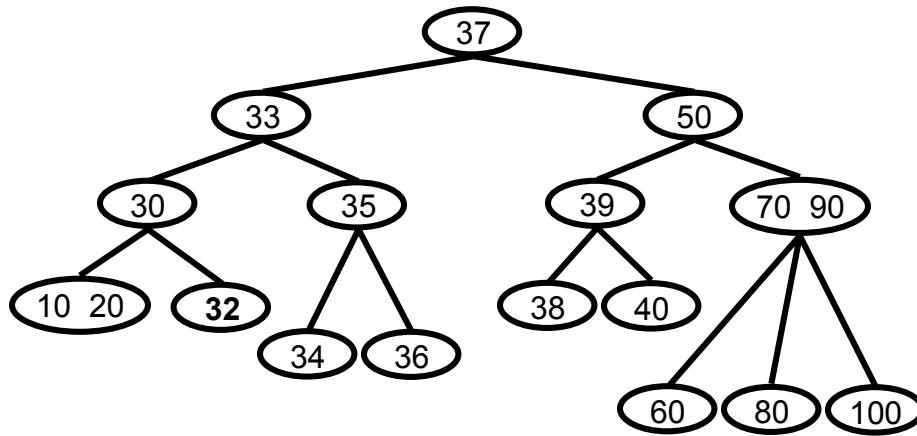
Insertion Examples



Push up, split apart

Page 42

Insertion Examples



A new level is born!

Page 43

Insertion Special Cases

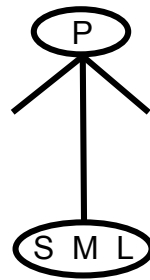
There are 3 special cases for insertion related to splitting:

- ◆ 1) Splitting a leaf node
 - ⇒ Promote middle key to parent and create two new nodes containing half the keys.
 - ⇒ Do not have child pointers to worry about.
- ◆ 2) Splitting an interior node
 - ⇒ Promote middle key to parent and create two new nodes containing half the keys.
 - ⇒ Make sure child pointers are copied over as well as keys.
- ◆ 3) Splitting the root node
 - ⇒ Similar to splitting an interior node, but now the tree will grow by one level and will have a new root node (must update root pointer).

Note that cases 1 and 2 are **ONLY** possible if a leaf node has been previously split. Case 3 is only possible if all ancestors of the leaf node had to be split.

Page 44

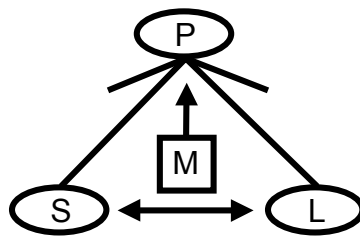
Special Case: Splitting a Leaf Node



Leaf node overflow

Page 45

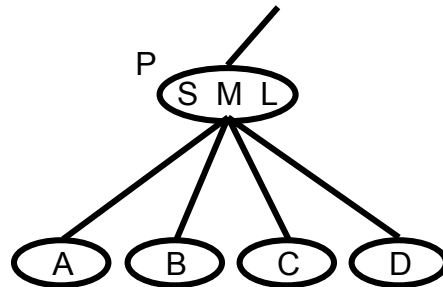
Special Case: Splitting a Leaf Node (2)



Splitting a leaf node

Page 46

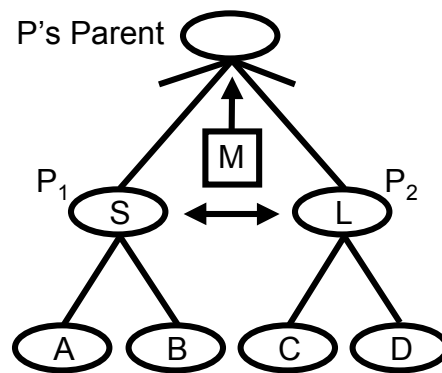
Special Case: Splitting an Interior Node



Interior node overflow
Splitting an internal node

Page 47

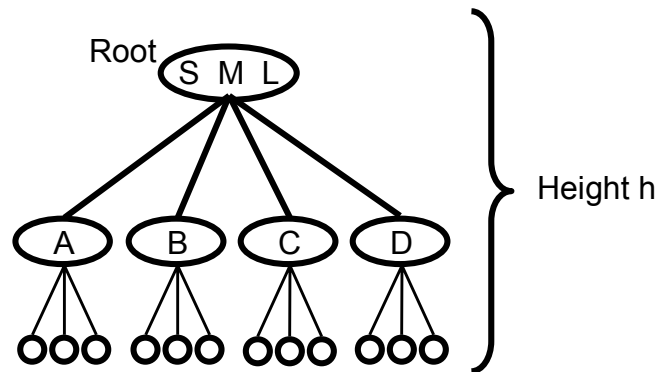
Special Case: Splitting an Interior Node (2)



Splitting an internal node

Page 48

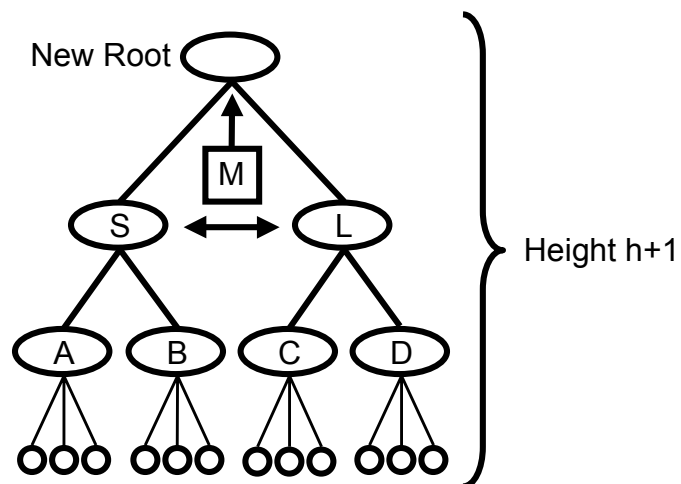
Special Case: Splitting the Root Node



Splitting the root node

Page 49

Special Case: Splitting the Root Node (2)



Page 50

Deletion From a 2-3 Tree

Algorithm:

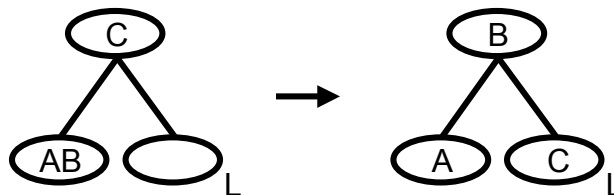
- ◆ To delete a key **K**, first locate the node **N** containing **K**.
 - ⇒ If **K** is not found, then deletion algorithm terminates.
- ◆ If **N** is an interior node, find **K**'s in-order successor and swap it with **K**. As a result, deletion always begins at a leaf node **L**.
- ◆ If leaf node **L** contains a value in addition to **K**, delete **K** from **L**, and we're done. (no underflow)
 - ⇒ For B-trees, underflow occurs if # of nodes < minimum.
- ◆ If underflow occurs (node has less than required # of keys), we merge it with its neighboring nodes.
 - ⇒ Check siblings of leaf. If sibling has two values, redistribute them.
 - ⇒ Otherwise, merge **L** with an adjacent sibling and bring down a value from **L**'s parent.
 - ⇒ If **L**'s parent has underflow, recursively apply merge procedure.
 - ⇒ If underflow occurs to the root, the tree may shrink a level.

Page 51

Deletion Re-distributing values in Leaf Nodes

If deleting **K** from **L** causes **L** to be empty:

- ◆ Check siblings of now empty leaf.
- ◆ If sibling has two values, redistribute the values.



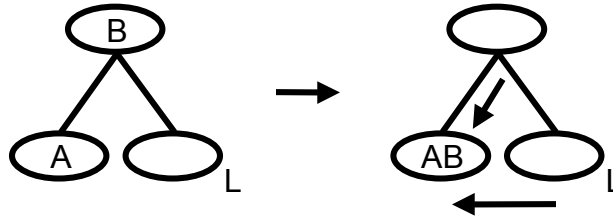
Page 52

Deletion

Merging Leaf Nodes

Merging leaf nodes:

- ◆ If no sibling node has extra keys to spare, merge L with an adjacent sibling and bring down a value from L's parent.



- ◆ The merging of L may cause the parent to be left without a value and only one child. If so, recursively apply deletion procedure to the parent.

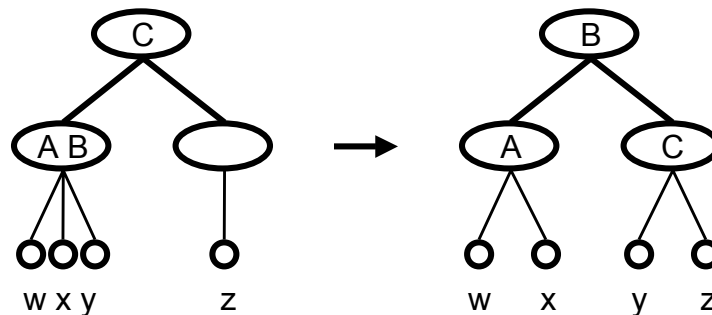
Page 53

Deletion

Re-distributing values in Interior Nodes

Re-distributing values in interior nodes:

- ◆ If the node has a sibling with two values, redistribute the values.

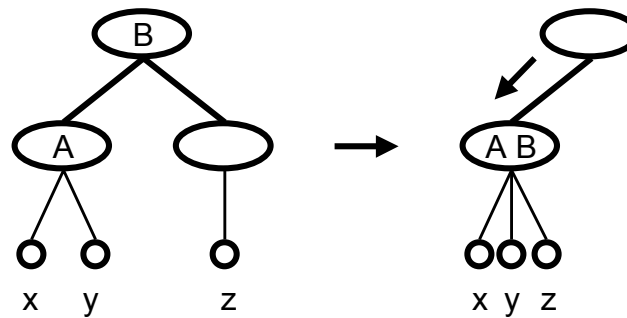


Page 54

Deletion Merging Interior Nodes

Merging interior nodes:

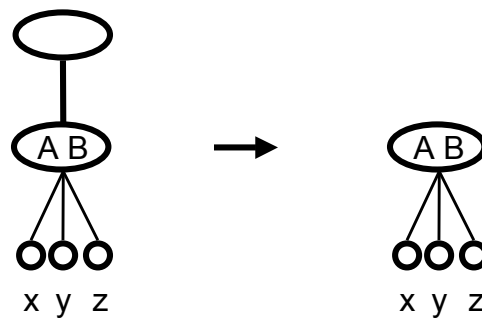
- ◆ If the node has no sibling with two values, merge the node with a sibling, and let the sibling adopt the node's child.



Page 55

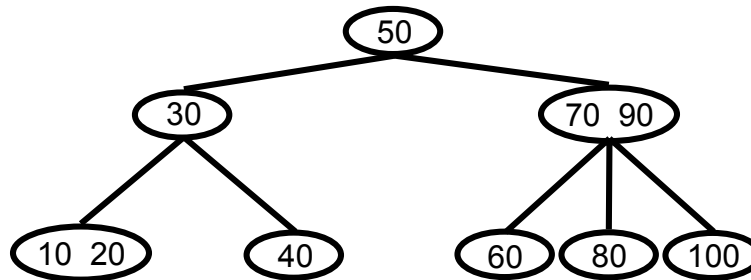
Deletion Merging on the Root Node

If the merging continues so that the root of the tree is without a value (and has only one child), delete the root. Height will now be $h-1$.



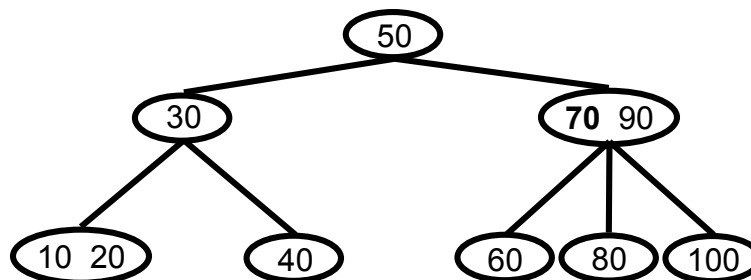
Page 56

Deletion Examples



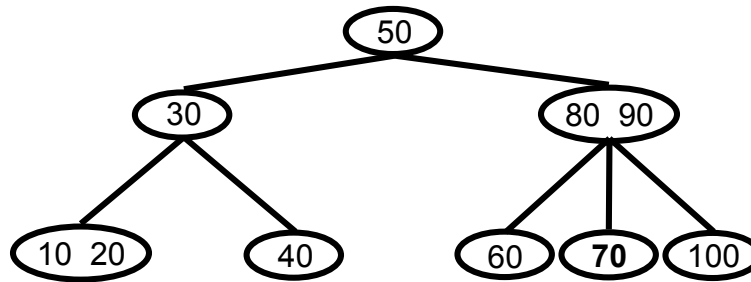
Original tree

Deletion Examples



Delete 70

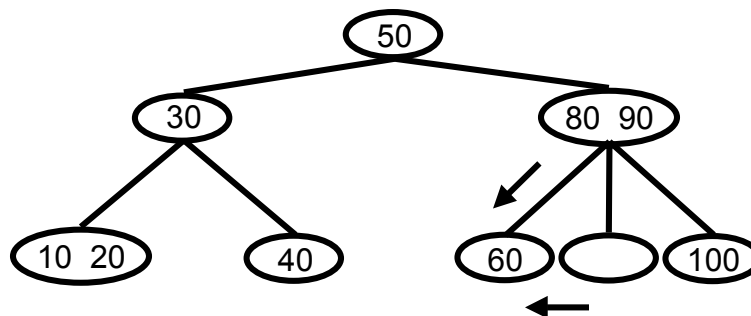
Deletion Examples



Swap with in-order successor

Page 59

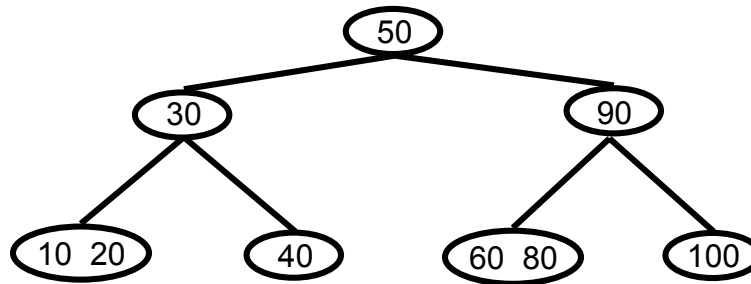
Deletion Examples



Merge and pull down

Page 60

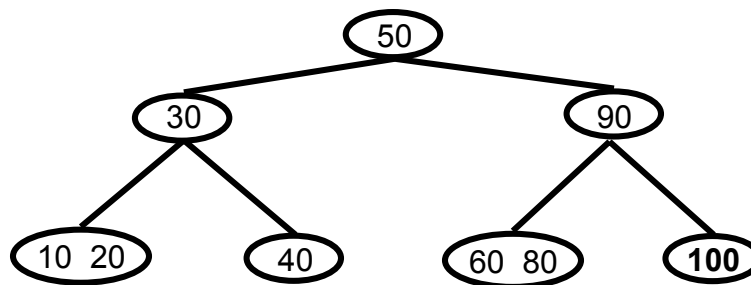
Deletion Examples



Done!

Page 61

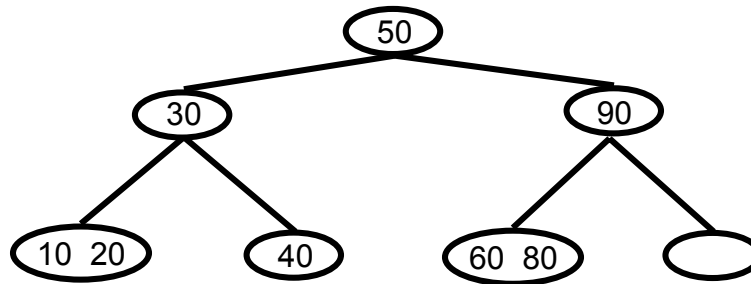
Deletion Examples



Delete 100

Page 62

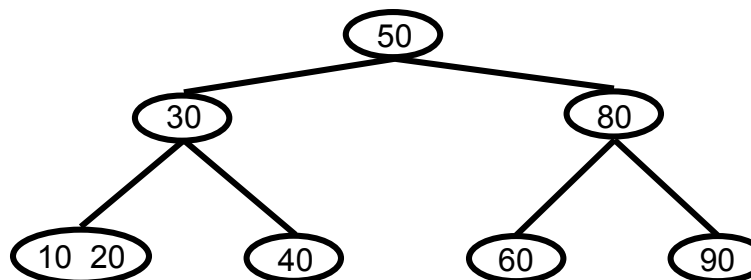
Deletion Examples



Redistribute

Page 63

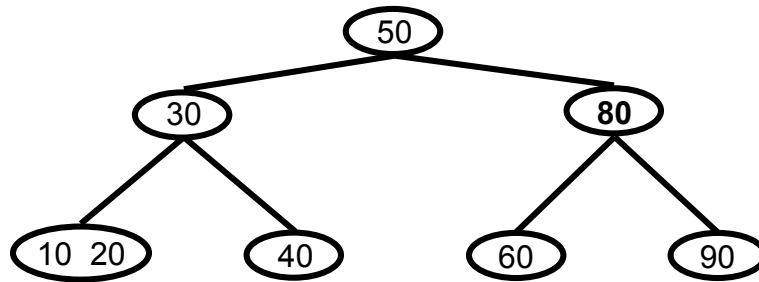
Deletion Examples



Done!

Page 64

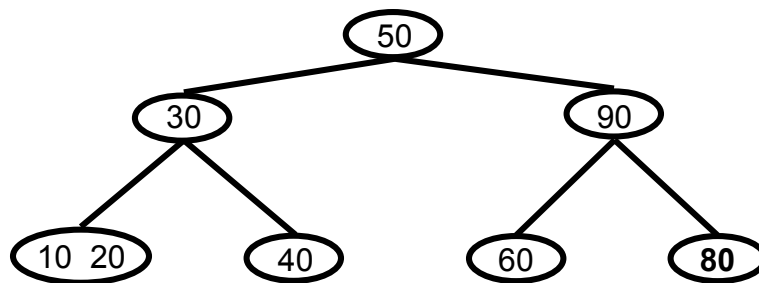
Deletion Examples



Delete 80

Page 65

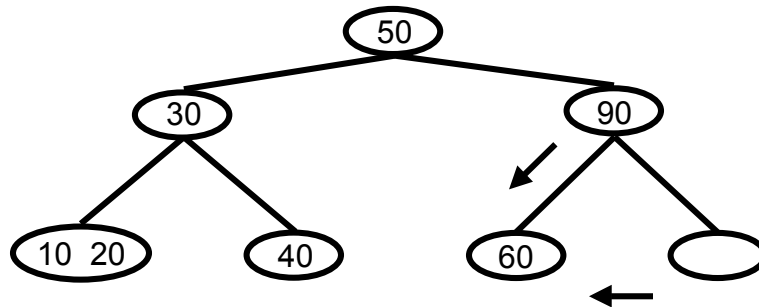
Deletion Examples



Swap with in-order successor

Page 66

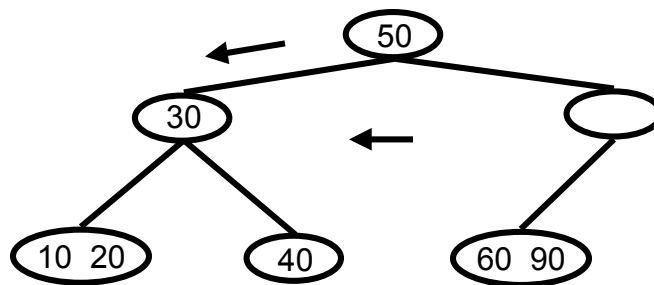
Deletion Examples



Merge and pull down

Page 67

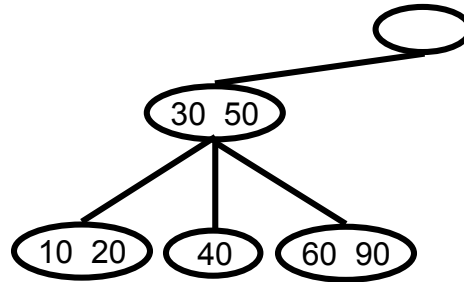
Deletion Examples



Merge and pull down

Page 68

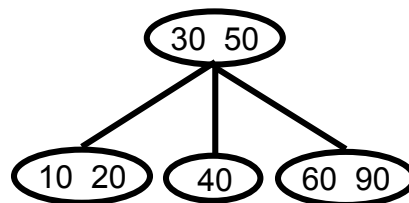
Deletion Examples



Merge and pull down

Page 69

Deletion Examples



Done

Page 70

B-trees as External Data Structures

Now that we understand how a B-tree works as a data structure, we will investigate how it can be used for an index.

A regular B-tree can be used as an index by:

- ◆ Each node in the B-tree stores not only keys, but also a record pointer for each key to the actual data being stored.
 - ⇒ Could also potentially store the record in the B-tree node itself.
- ◆ To find the data you want, search the B-tree using the key, and then use the pointer to retrieve the data.
 - ⇒ No additional disk access is required if the record is stored in the node.

Given this description, it is natural to wonder how we might calculate the best B-tree **order**.

- ◆ Depends on disk block and record size.
- ◆ We want a node to occupy an entire block.

Page 71



Calculating the Size of a B-tree Node

Given a block of 4096 bytes, calculate the order of a B-tree if the key size is 4 bytes, and all pointers are 8 bytes.

Answer:

Assuming no header information is kept in blocks:

$$\text{node size} = \text{keySize} * \text{numKeys} + \text{dataPtrSize} * \text{numKeys} + \text{childPtrSize} * (\text{numKeys} + 1)$$

$$\text{size of one node} = 4 * k + 8 * k + 8 * (k + 1) \leq 4096$$

$$k = 204 \text{ keys}$$

Maximum order is 102.

Page 72

B-tree Question

Given a block of 4096 bytes, calculate the order of a B-tree if the key size is 4 bytes, internal B-tree pointers are 8 bytes, and we store the record itself in the B-tree node instead of a pointer. The record size is 100 bytes.

Advantages of B-trees

The advantages of a B-tree over ISAM or other multi-level indexing schemes are:

- ◆ 1) B-trees automatically create or destroy index levels as the data file changes.
- ◆ 2) B-trees automatically manage record allocation to blocks, so no overflow blocks are needed.
- ◆ 3) A B-tree is always balanced, so the search time is the same for any search key and is logarithmic.

For these reasons, B-trees and B+-trees are the index scheme of choice for commercial databases.

B+-trees

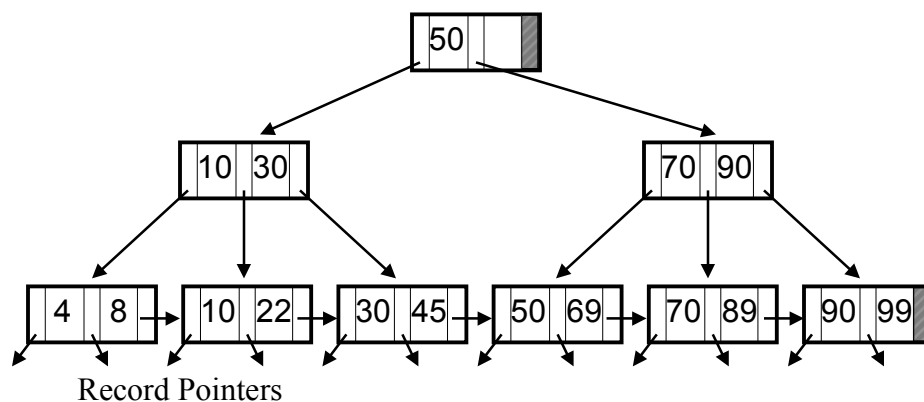
A **B+-tree** is a multi-level index structure like a B-tree except that *all* data is stored at the leaf nodes of the resulting tree instead of (possibly) within the tree itself.

- ◆ Each leaf node contains a pointer to the next leaf node which makes it easy to chain together and maintain the data records in “sequential” order for sequential processing.

Thus, a B+-tree has two distinct node types:

- ◆ 1) **interior nodes** - store pointers to other interior nodes or leaf nodes.
- ◆ 2) **leaf nodes** - store keys and pointers to the data records (or the data records themselves).

B+-tree Example



Operations on B+-trees

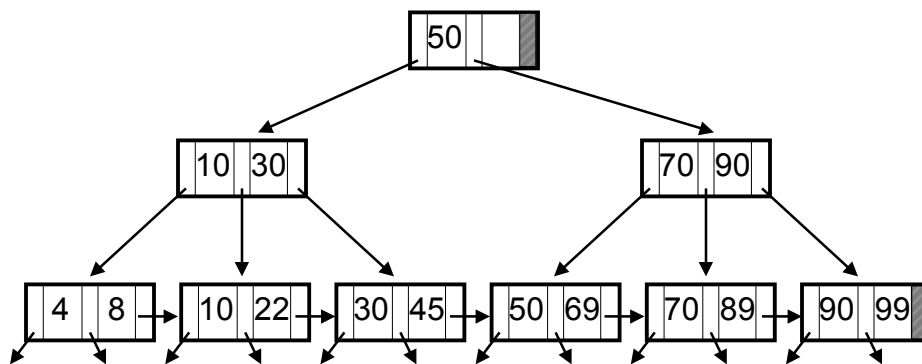
The general algorithms for inserting and deleting from a B+-tree are similar to B-trees except for one important difference:

- ◆ All key values stay in leaves.

When we must merge nodes for deletion or add nodes during splitting, the key values removed/promoted to the parent nodes are copies.

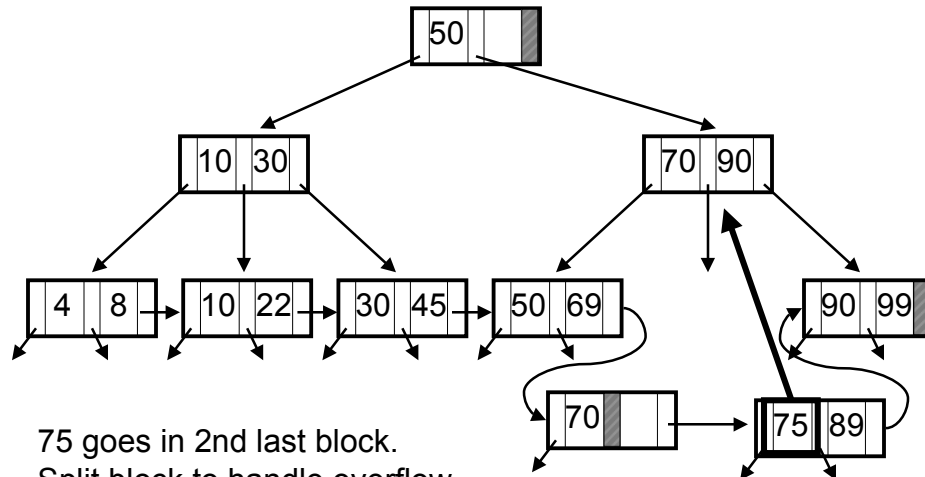
- ◆ That is, all non-leaf levels do not store actual data, they are simply a hierarchy of multi-level index to the data.

B+-tree Insert Example



Insert 75

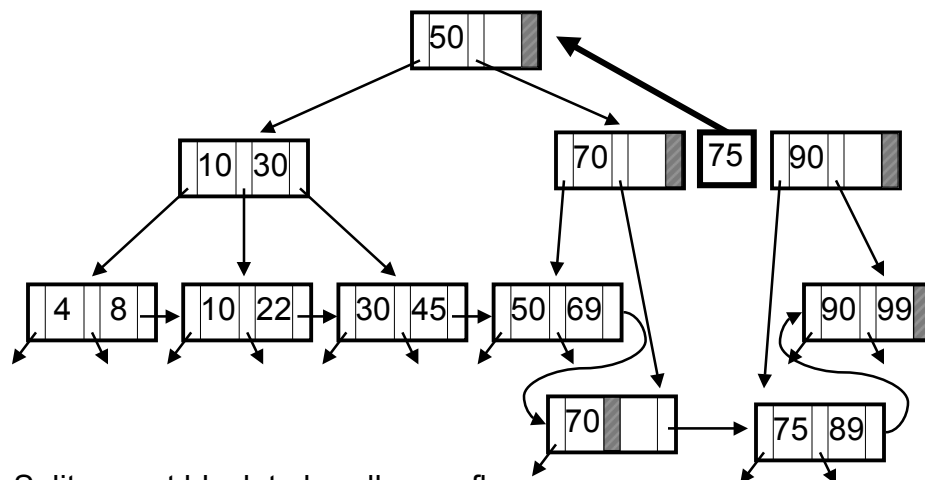
B+-tree Insert Example (2)



75 goes in 2nd last block.
Split block to handle overflow.
Promote 75. Note that 75 stays in a leaf!

Page 79

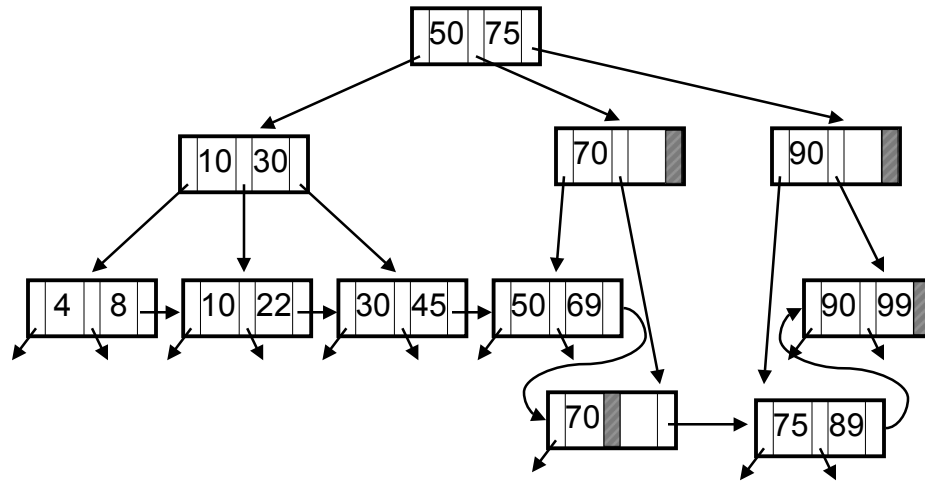
B+-tree Insert Example (3)



Split parent block to handle overflow.
Promote 75. Note that 75 does not stay!

Page 80

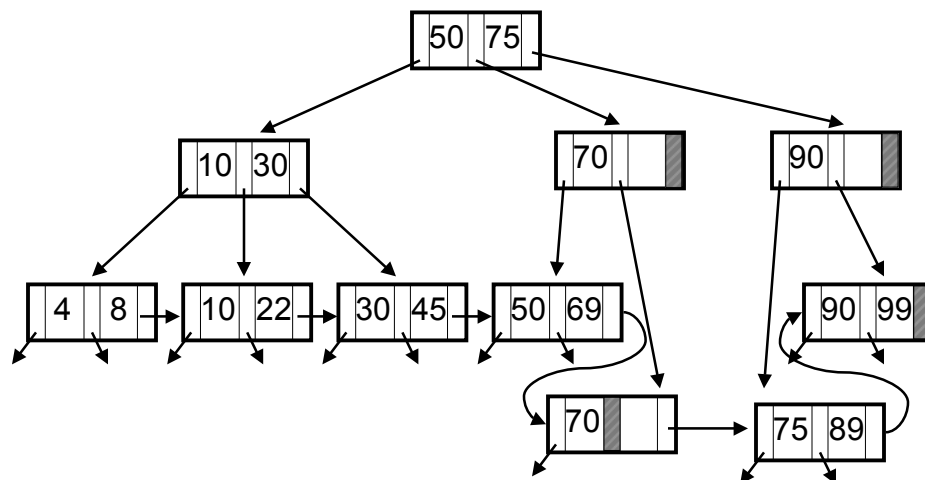
B+-tree Insert Example (4)



Insertion done!

Page 81

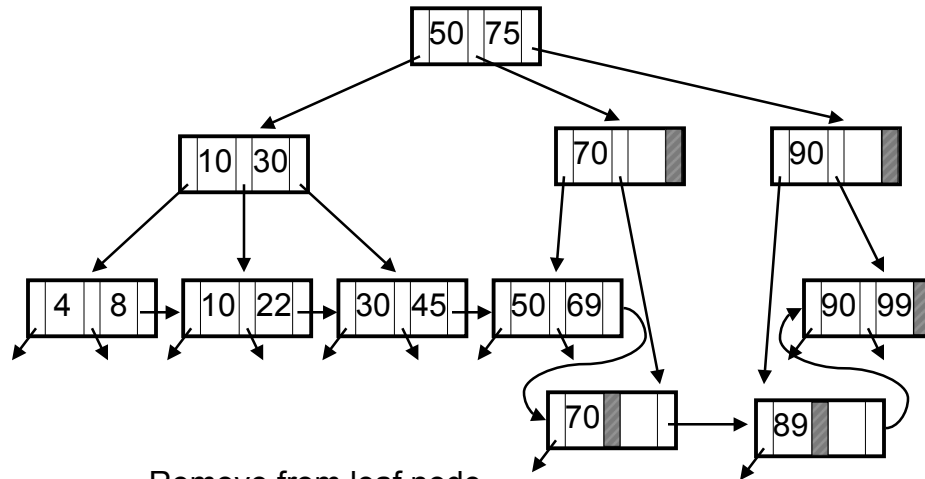
B+-tree Delete Example



Delete 75.

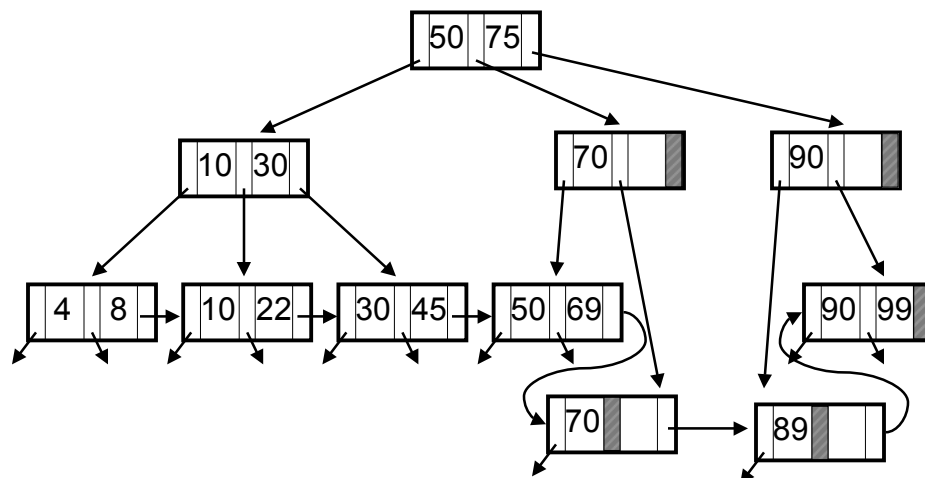
Page 82

B+-tree Delete Example (2)



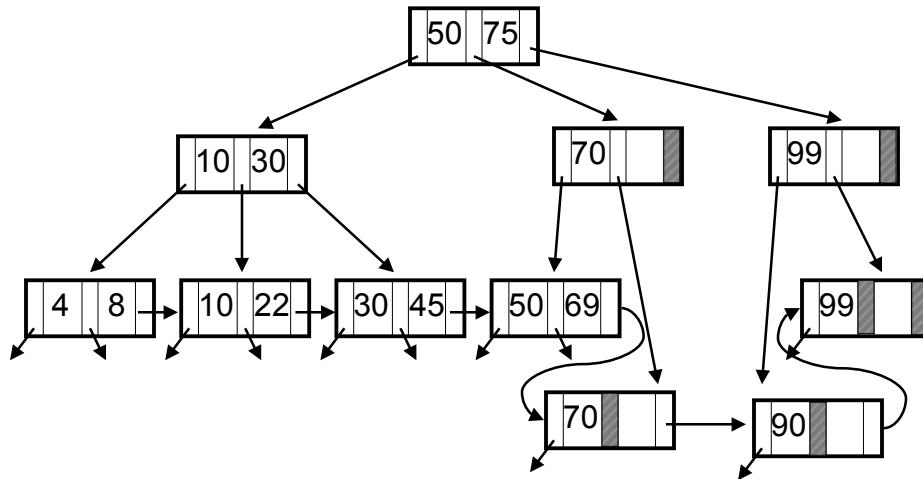
Page 83

B+-Tree Delete Example 2



Page 84

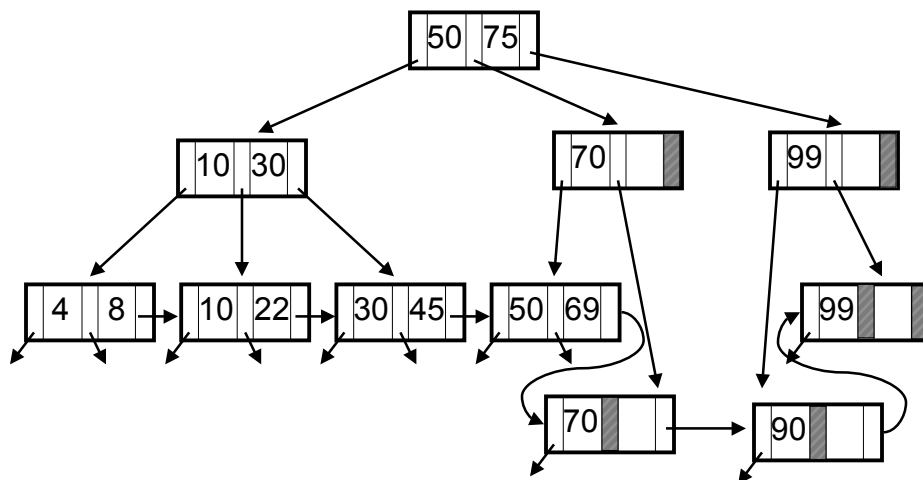
B+-Tree Delete Example 2 (2)



Redistribute keys 90 and 99.

Page 85

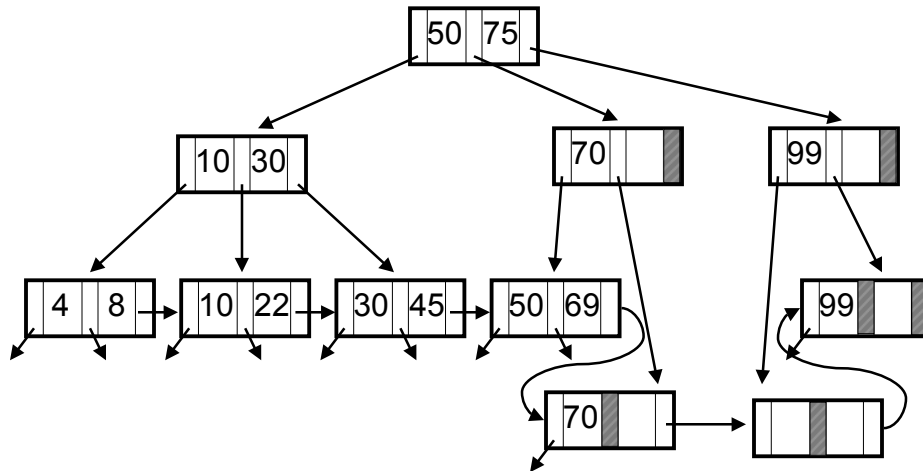
B+-Tree Delete Example 3



Delete 90.

Page 86

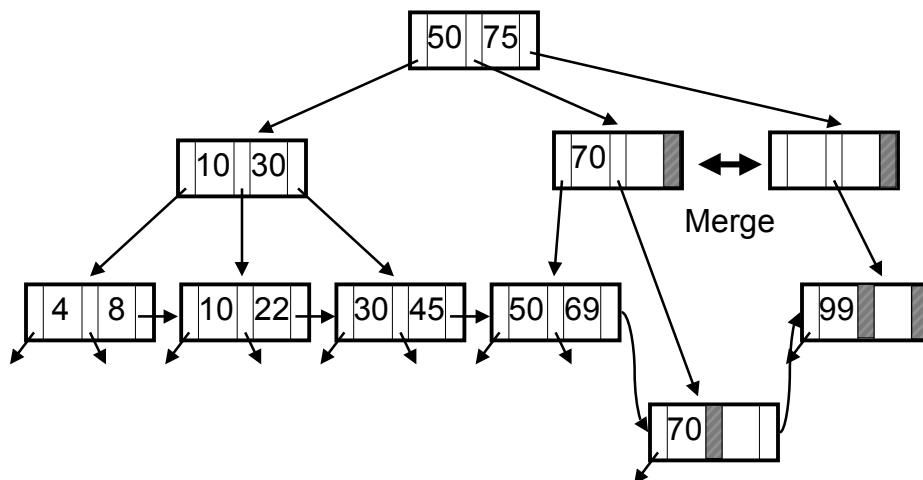
B+-Tree Delete Example 3 (2)



Empty leaf node. Merge with sibling.

Page 87

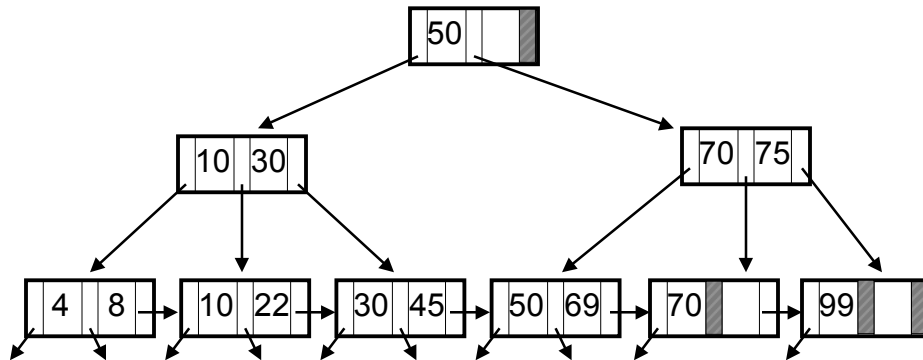
B+-Tree Delete Example 3 (2)



Empty interior node. Merge with sibling.

Page 88

B+-Tree Delete Example 3 (3)



Bring down 75 from parent node. Done.

Page 89



B+-tree Practice Question

For a B+-tree of **order 2 (max. keys=4)**, insert the following keys in order:

- ◆ 10, 20, 30, 40, 50, 60, 70, 80, 90
- ◆ Assuming keys increasing by 10, what is the first key added that causes the B+-tree to grow to height 3?

Show the tree after deleting the following keys:

- ◆ a) 70
- ◆ b) 90
- ◆ c) 10
- ◆ Assume you start with the tree after inserting 90 above.

Page 90

B+-tree Challenge Exercise

For a B+-tree with **maximum keys=3**, insert the following keys in order:

◆ 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Show the tree after deleting the following keys:

◆ a) 70

◆ b) 90

◆ c) 10

Try the deletes when the minimum # of keys is 1 and when the minimum # of keys is 2.

Page 91

Observations about B+-trees

Since the inter-node connections are done by pointers, there is no assumption that in the B+-tree, the “logically” close blocks are “physically” close.

◆ The non-leaf levels of the B+-tree form a hierarchy of sparse indices.

The B+-tree contains a relatively small number of levels (logarithmic in the size of the main file), thus searches and modifications can be conducted efficiently.

⇒ Further, in practice, coalescing/deleting is not implemented as it is too hard and not worth it!

Example:

◆ If a B+-tree node can store 300 key-pointer pairs at maximum, and on average is 69% full, then 208 (207+1) pointers/block.

◆ Level 3 B+-tree can index 208^3 records = 8,998,912 records!

Page 92

B+-trees Discussion

By isolating the data records in the leaves, we also introduce slight additional implementation complexity because the leaf and interior nodes have different structures.

- ◆ Interior nodes contain only pointers to additional index nodes or leaf nodes while leaf nodes contain pointers to data records.

This additional complexity is outweighed by the advantages of B+-trees which include:

- ◆ Better sequential access ability.
- ◆ Greater overall storage capacity for a given block size since the interior nodes can hold more pointers each of which requires less space.
- ◆ Uniform data access times.

Page 93

B-trees Summary

A **B-tree** is a search tree where each node has $\geq n$ data values and $\leq 2n$, where we chose n for our particular tree.

- ◆ A 2-3 tree is a special case of a B-tree.
- ◆ Common operations: search, insert, delete
 - ⇒ Insertion may cause node overflow that is handled by promotion.
 - ⇒ Deletion may cause node underflow that is handled by mergers.
- ◆ Handling special cases for insertion and deletion make the code for implementing B-trees very complex.
- ◆ Note difference between B+-tree and B-tree for insert/delete!

B+-trees are commonly used as an indexing structure because they can be updated in logarithmic time, manage record pointer allocation on blocks, can be searched in logarithmic time, and support sequential access.

- ◆ They are also better than regular B-trees.

Page 94

Major Objectives

The "One Things":

- ◆ Insert and delete from a B-tree and a B+-tree.
- ◆ Calculate the maximum order of a B-tree.

Major Theme:

- ◆ B-trees are the standard multi-level indexing method because they are space-efficient and handle insertions/deletions automatically.

Other objectives:

- ◆ Calculate query access times using B-trees indexes.
- ◆ List disadvantages of ISAM compared to B-trees.
- ◆ Compare/contrast B-trees and B+-trees.