B-trees

COSC 404 – Database System Implementation

B-Trees and Indexing
Overview

We have seen how multi-level indexes can improve search performance.

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

We will learn B+-trees which are the most common form of index used in database systems today.

B-trees
Introduction

A ***B-tree*** is a search tree where each node has >= ***n*** data values and <= ***2n***, where we chose ***n*** for our particular tree.

- Each key in a node is stored in a sorted array.
  - key[0] is the first key, key[1] is the second key,…,key[2*n*-1] is the $2n^{th}$ key
  - key[0] < key[1] < key[2] < … < key[2*n*-1]
- There is also an array of pointers to children nodes:
  - child[0], child[1], child[2], …, child[2*n*]
  - Recursive definition: Each subtree pointed to by child[i] is also a B-tree.
- For any key[i]:
  - 1) key[i] > all entries in subtree pointed to by child[i]
  - 2) key[i] <= all entries in subtree pointed to by child[i+1]
- A node may not contain all key values.
  - # of children = # of keys +1

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

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, may 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, can 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 ***n/2*** $\lfloor \rfloor$

B-Trees Performance

Question: A B-tree has a maximum of 10 keys per node. What is the maximum number of children for a given node?

A) 0

B) 1

C) 10

D) 11

E) 20

2-3 Trees
Introduction

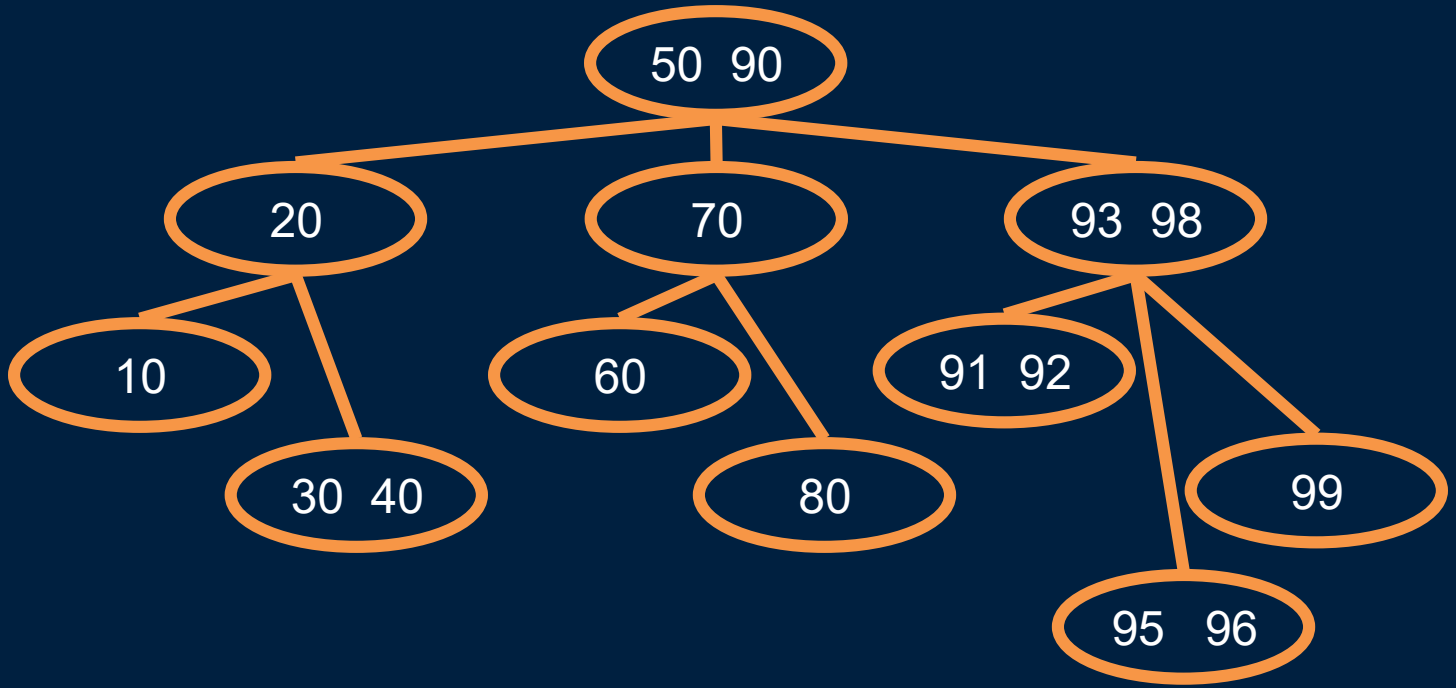
A ***2-3 tree*** is a B-tree where each node has either ***1*** or ***2*** data values and ***2*** or ***3*** children pointers.

- 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.

2-3 Search Tree Example



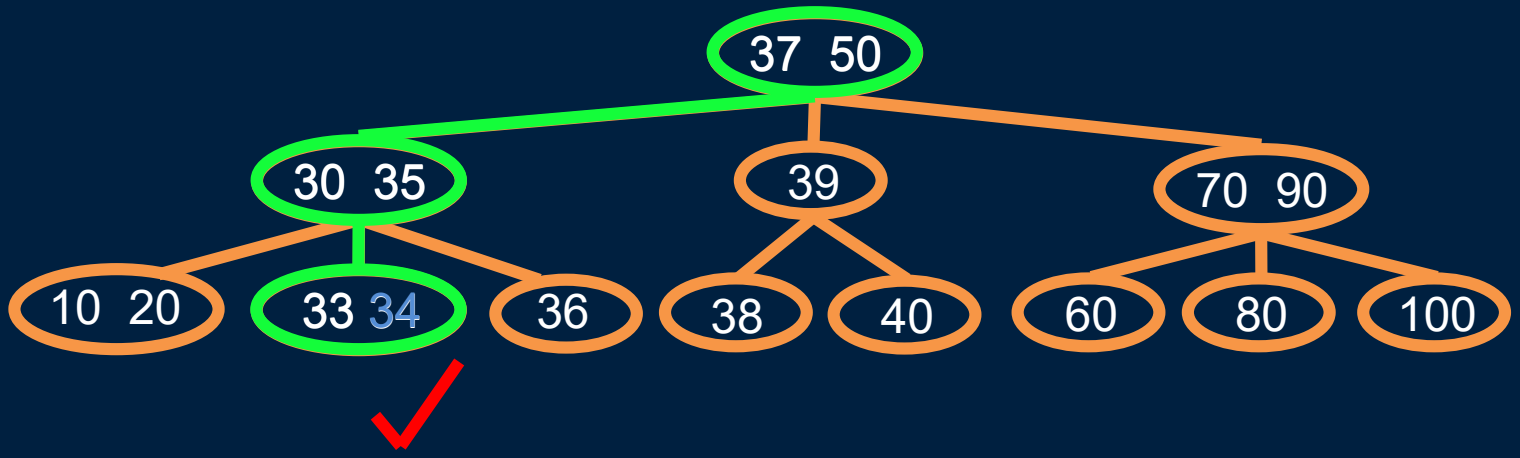
Searching a 2-3 Tree

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

Algorithm:

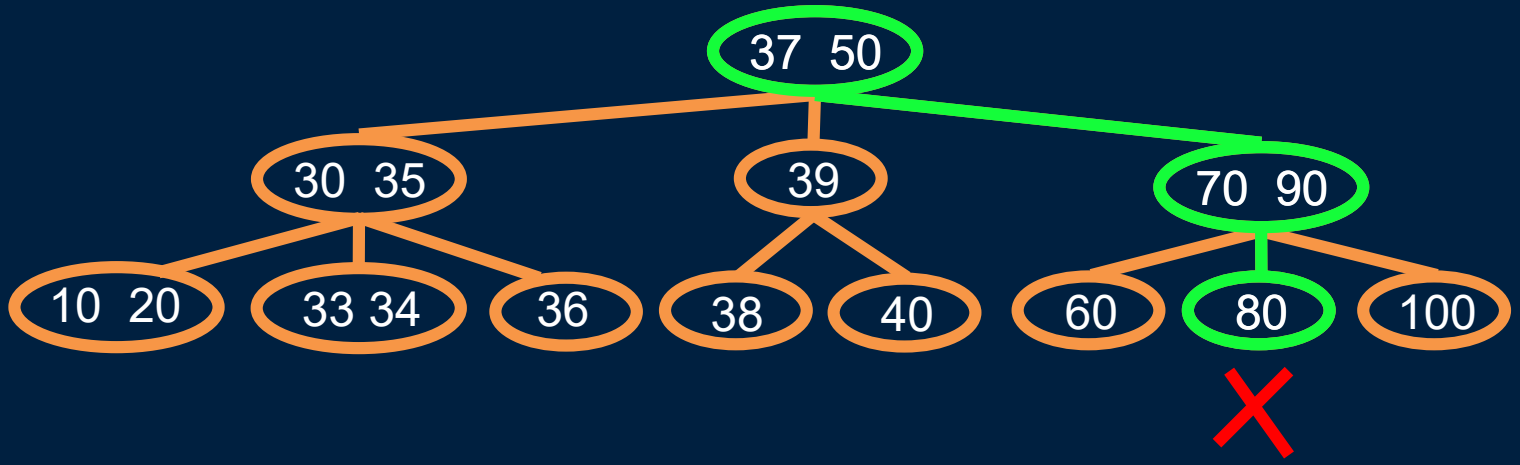
- Start at the root which begins as the curNode.
- If curNode contains the search key we are done, and have found the search key we were looking for.
- A 2-node contains one key:
  - If search key < key[0], go left (child[0]) otherwise go right (child[1])
- A 3-node contains two key values:
  - If search 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.

Searching a 2-3 Tree
Example #1



Find 34

Searching a 2-3 Tree
Example #2



Find 82

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

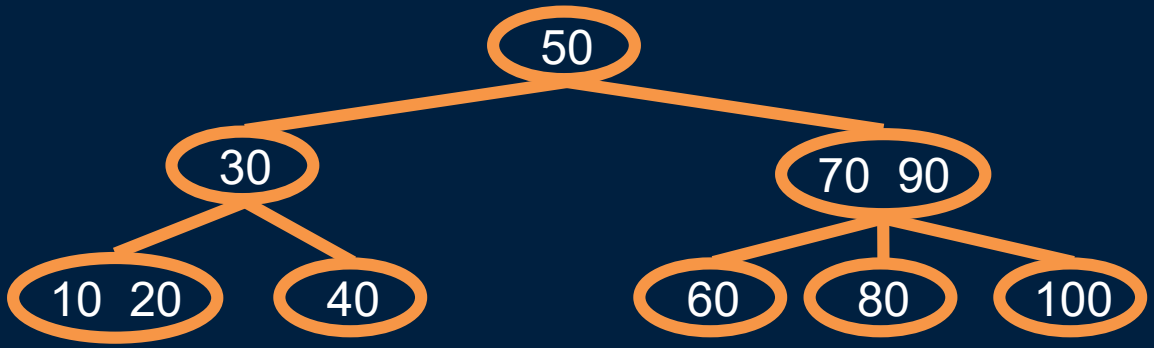
Splitting Algorithm:

- 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 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.

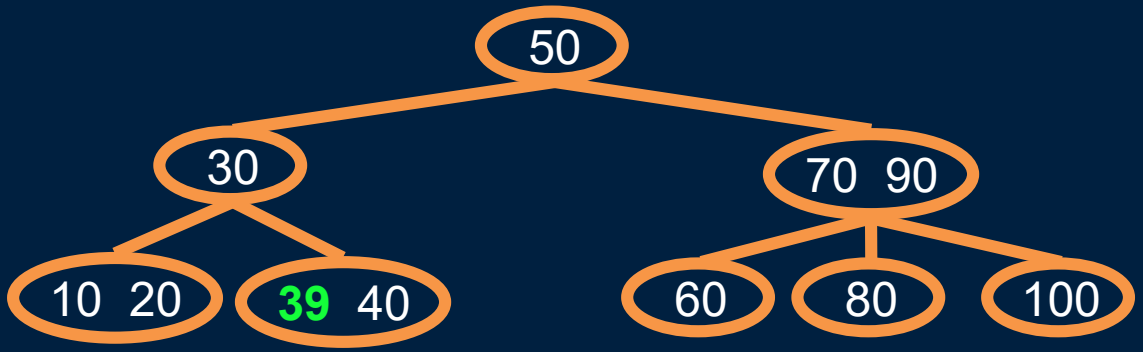
- Thus, 2-3 trees provide an efficient height balanced tree.

Insertion Examples



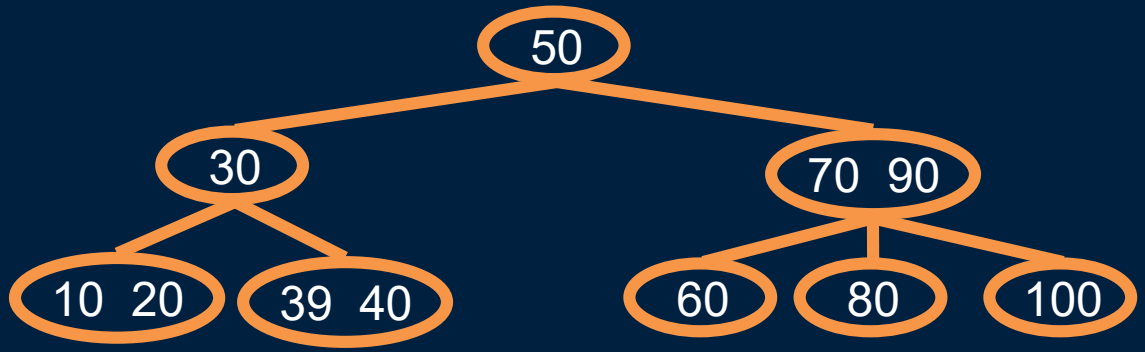
Insert 39

Insertion Examples



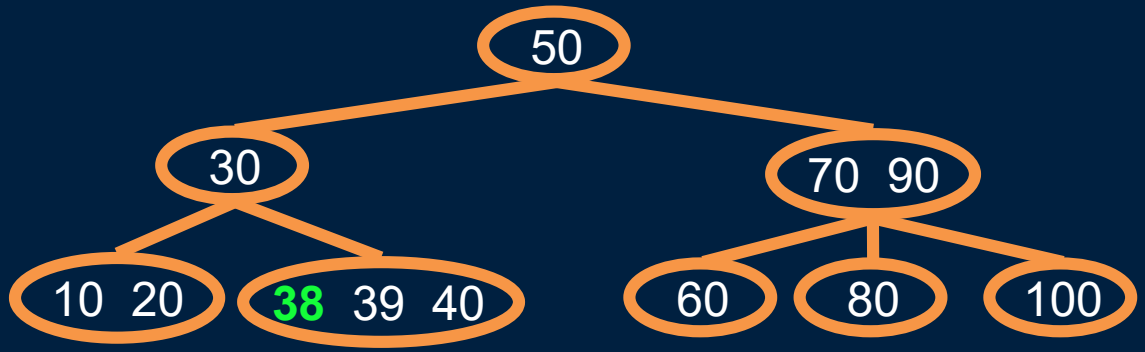
Done!

Insertion Examples



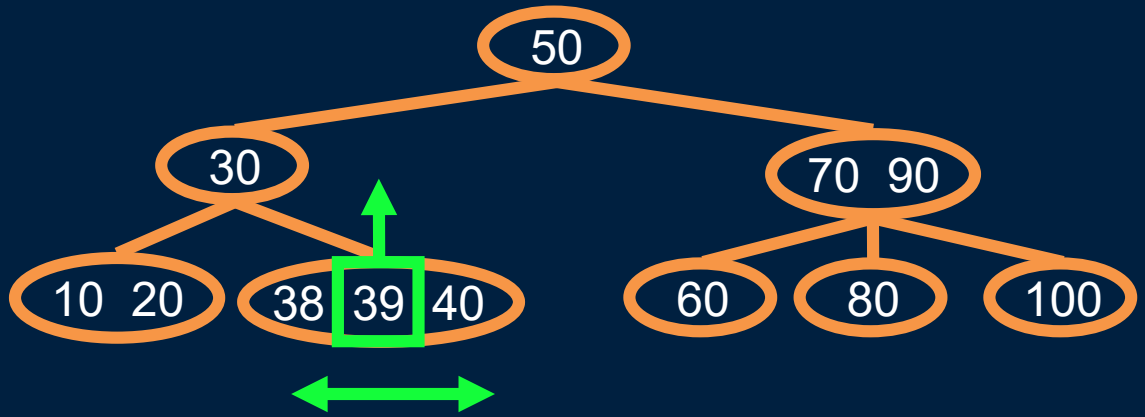
Insert 38

Insertion Examples



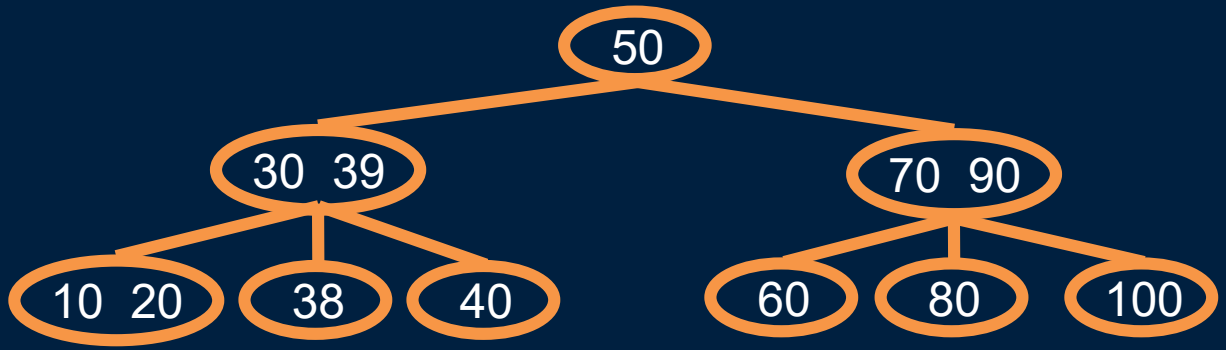
Insert 38

Insertion Examples



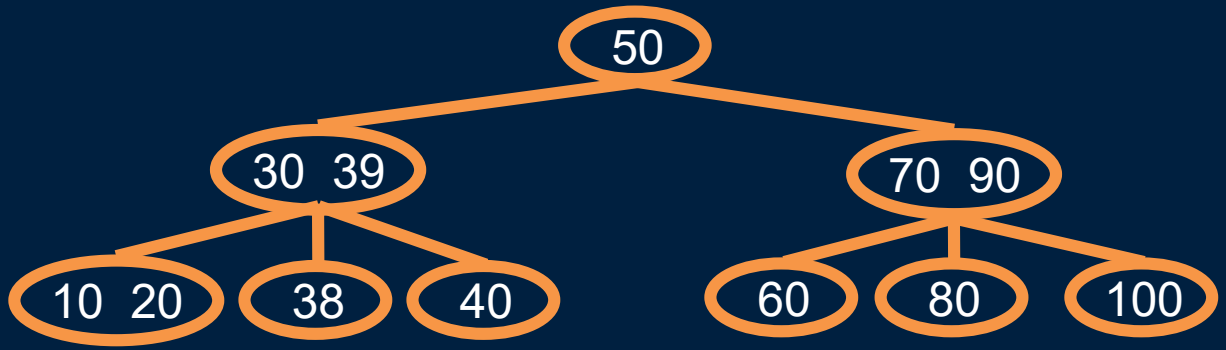
Push up, split apart

Insertion Examples



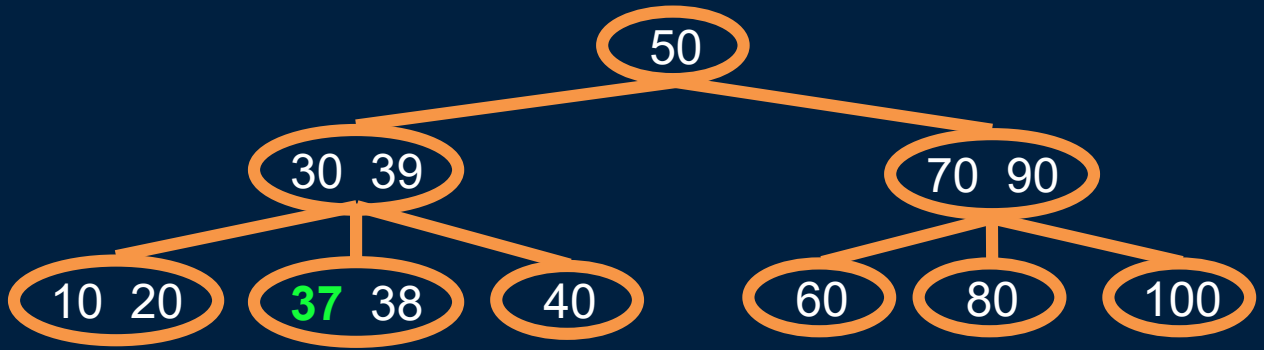
Done!

Insertion Examples



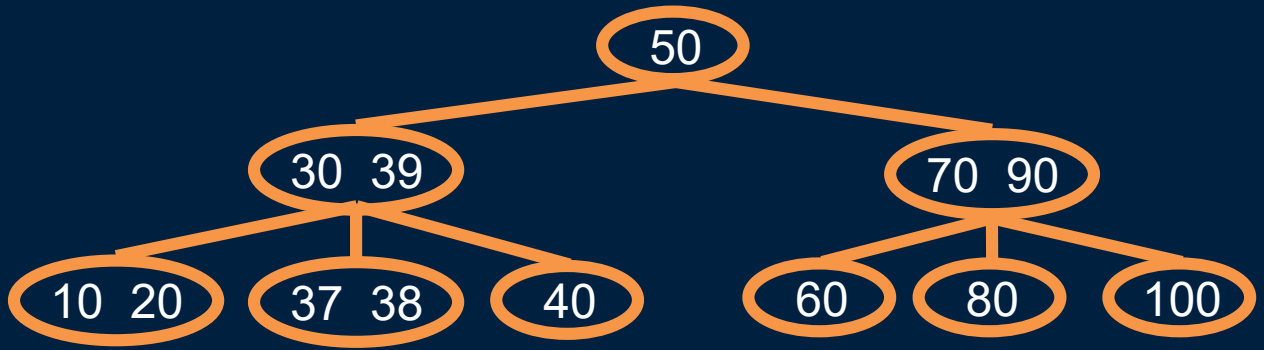
Insert 37

Insertion Examples



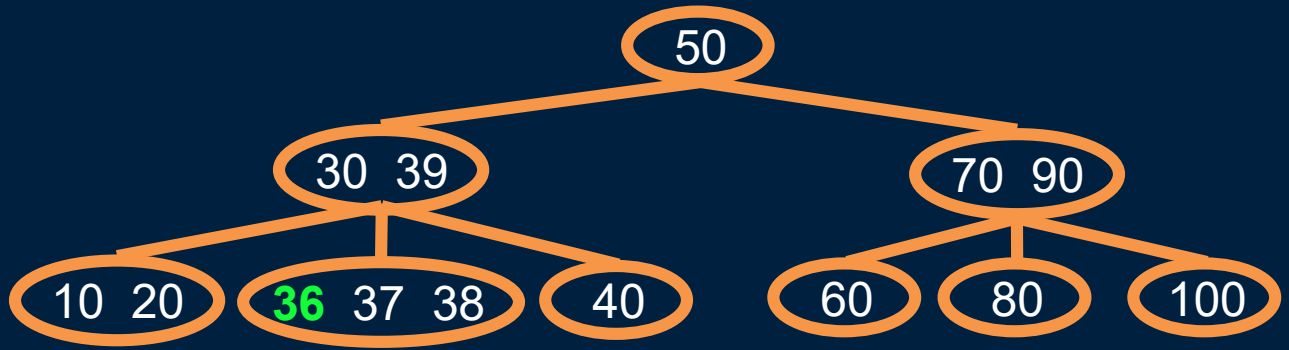
Done!

Insertion Examples



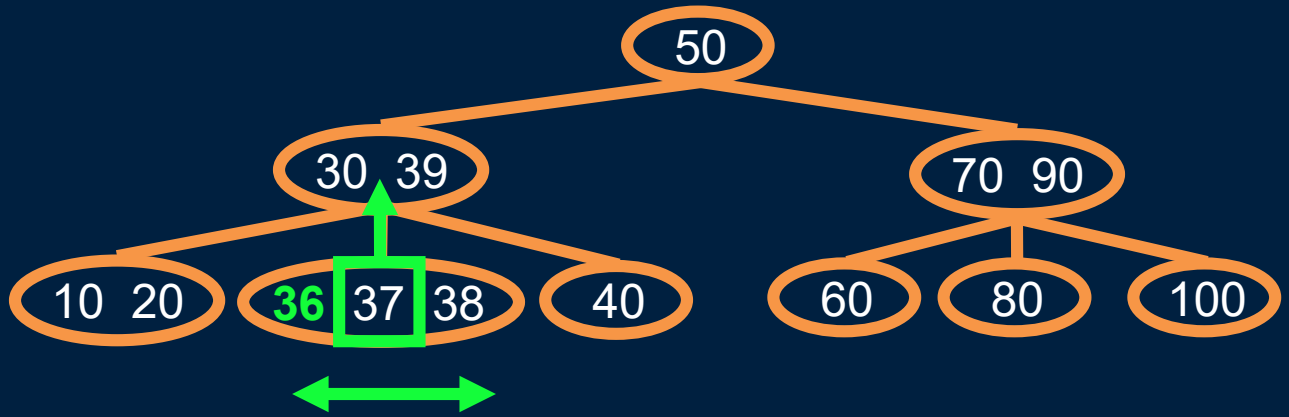
Insert 36

Insertion Examples



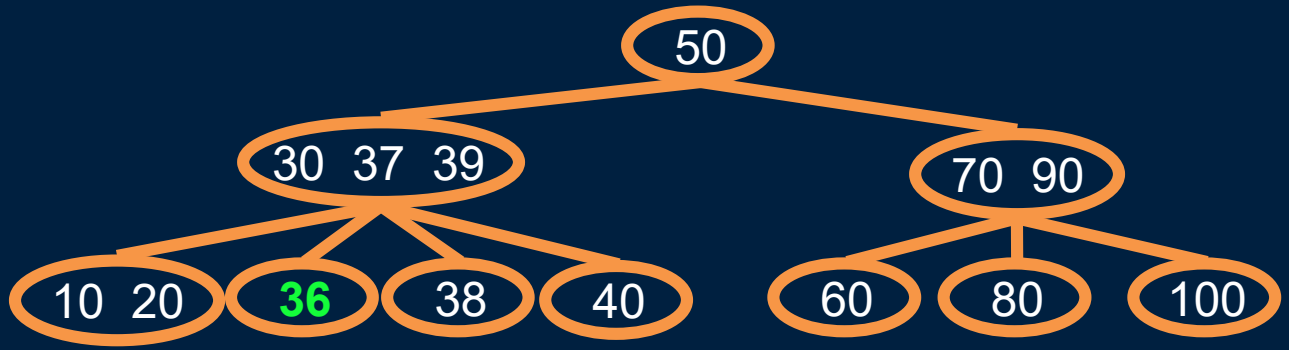
Insert 36

Insertion Examples



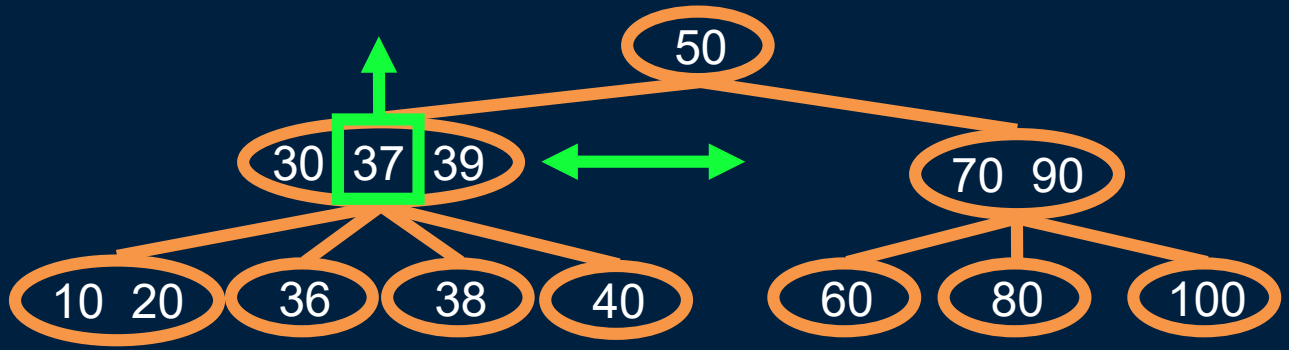
Push up, split apart

Insertion Examples



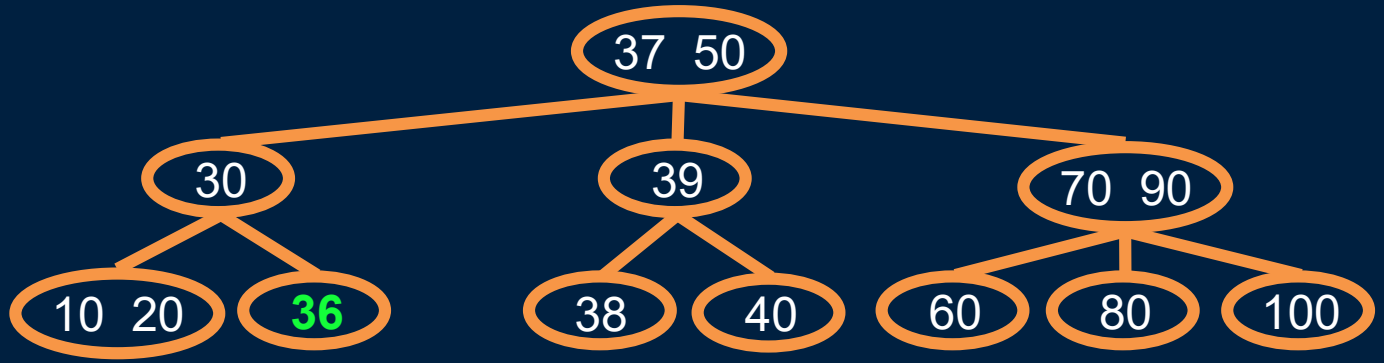
Need to go further up the tree to resolve overcrowding

Insertion Examples



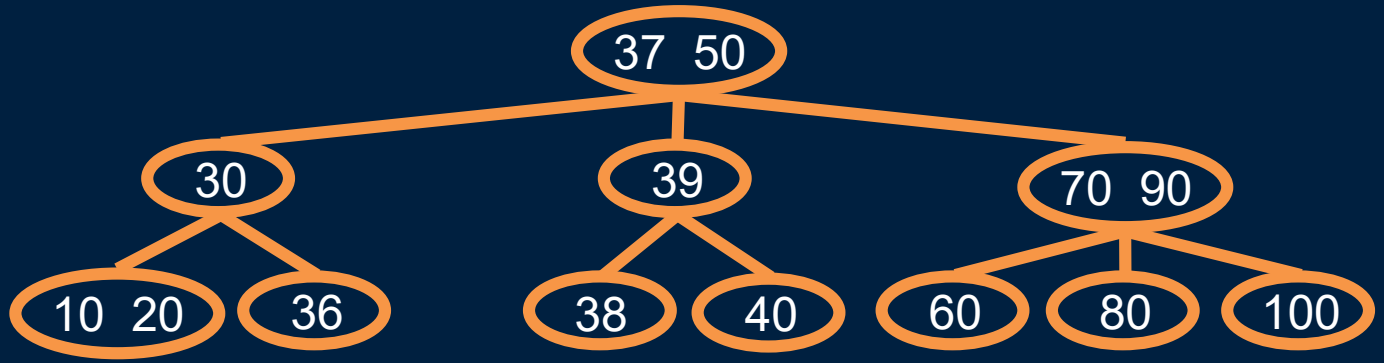
Push up, split apart

Insertion Examples



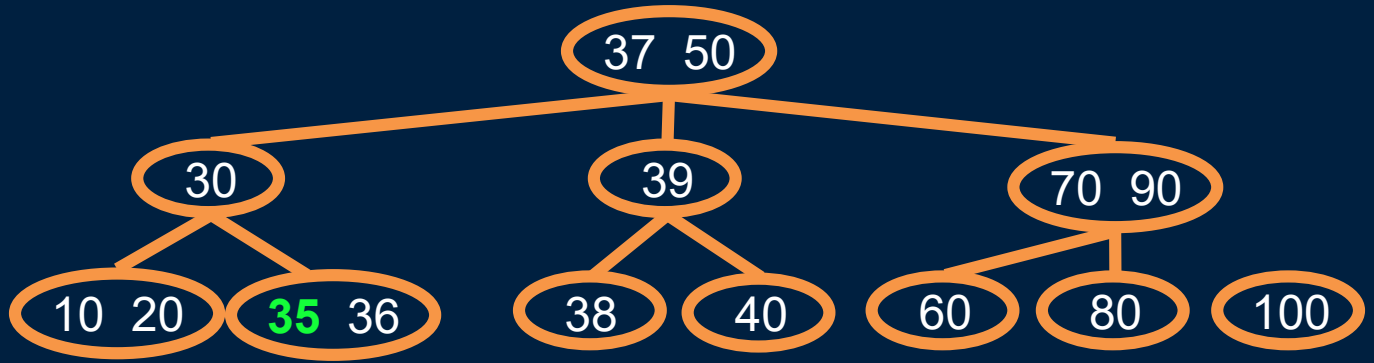
Done!

Insertion Examples



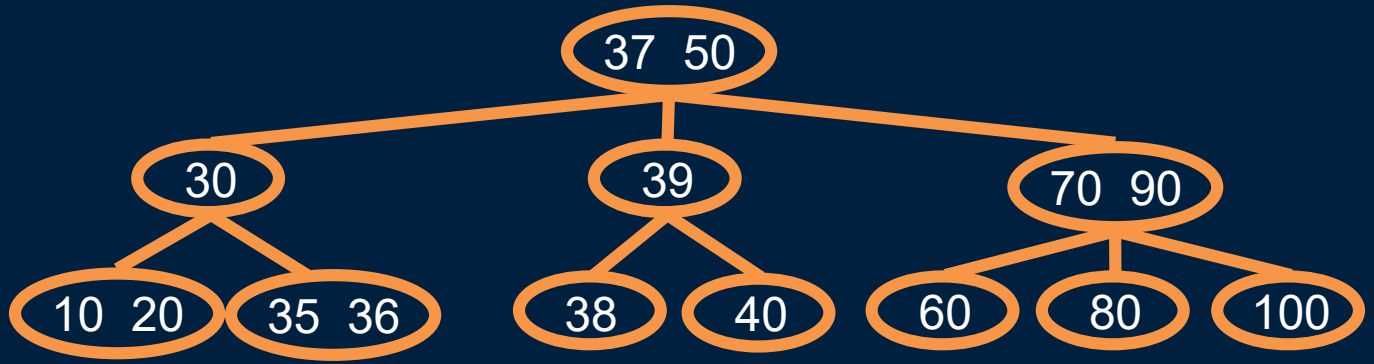
Insert 35

Insertion Examples



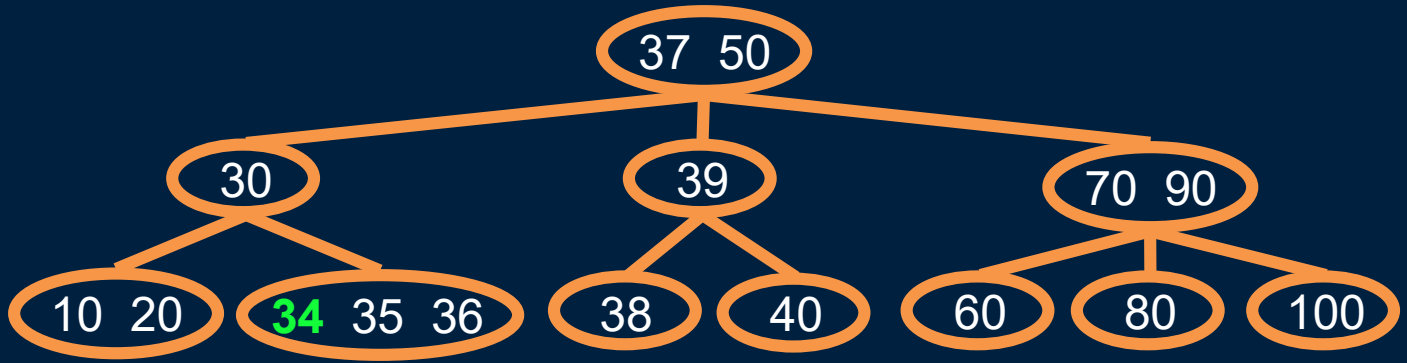
Insert 35

Insertion Examples



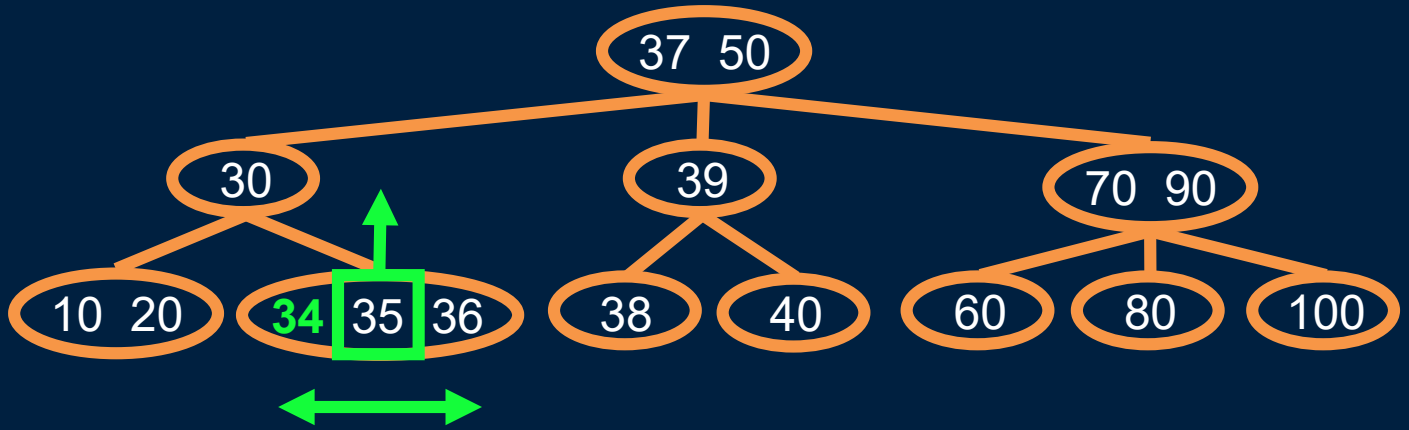
Insert 34

Insertion Examples



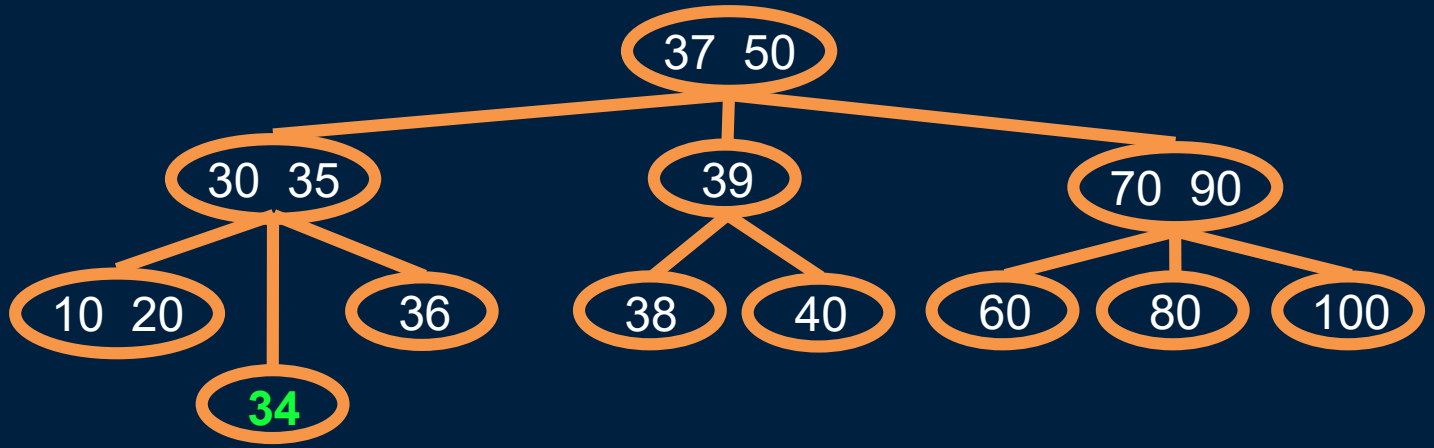
Insert 34

Insertion Examples



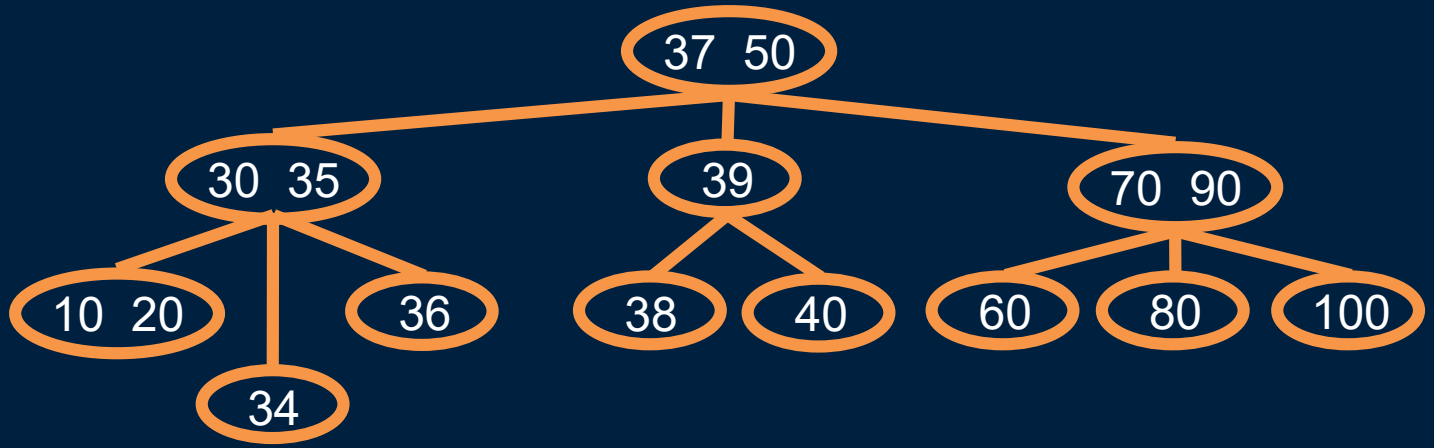
Push up, split apart

Insertion Examples



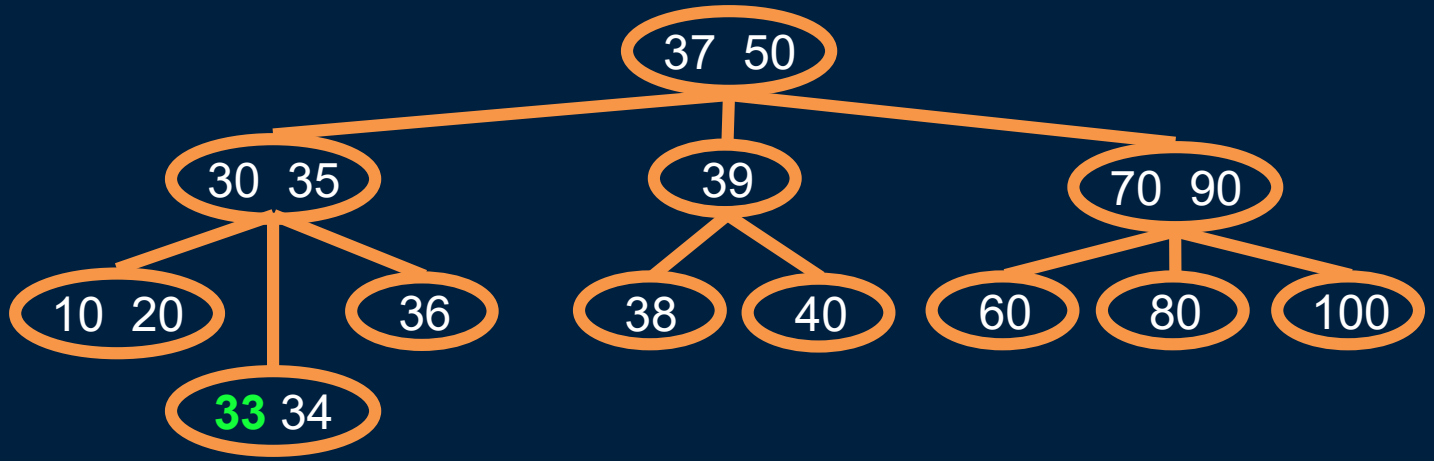
Done!

Insertion Examples



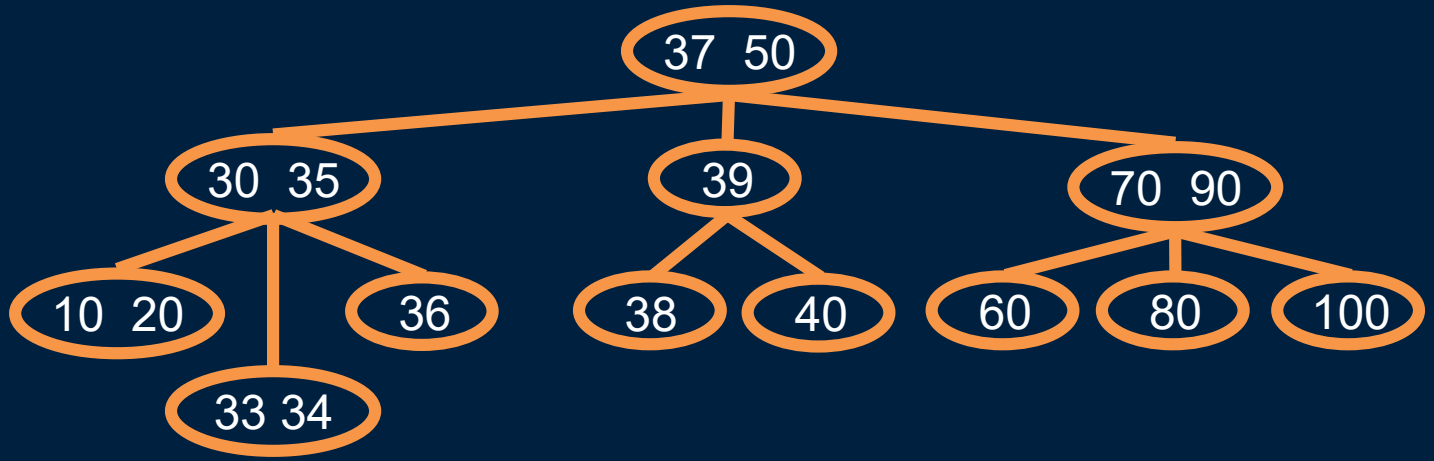
Insert 33

Insertion Examples



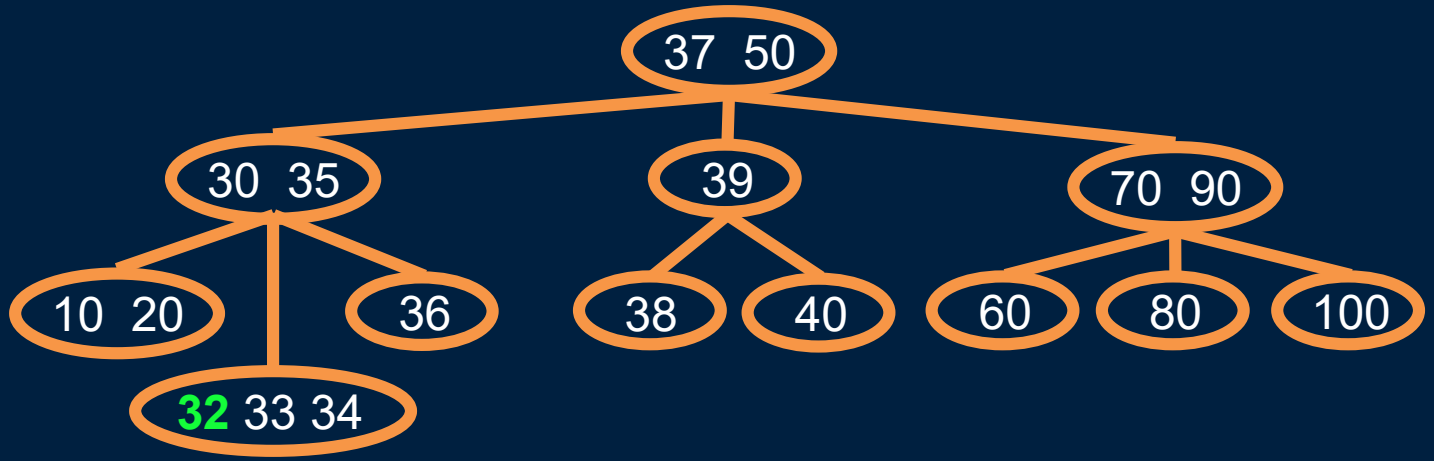
Done!

Insertion Examples



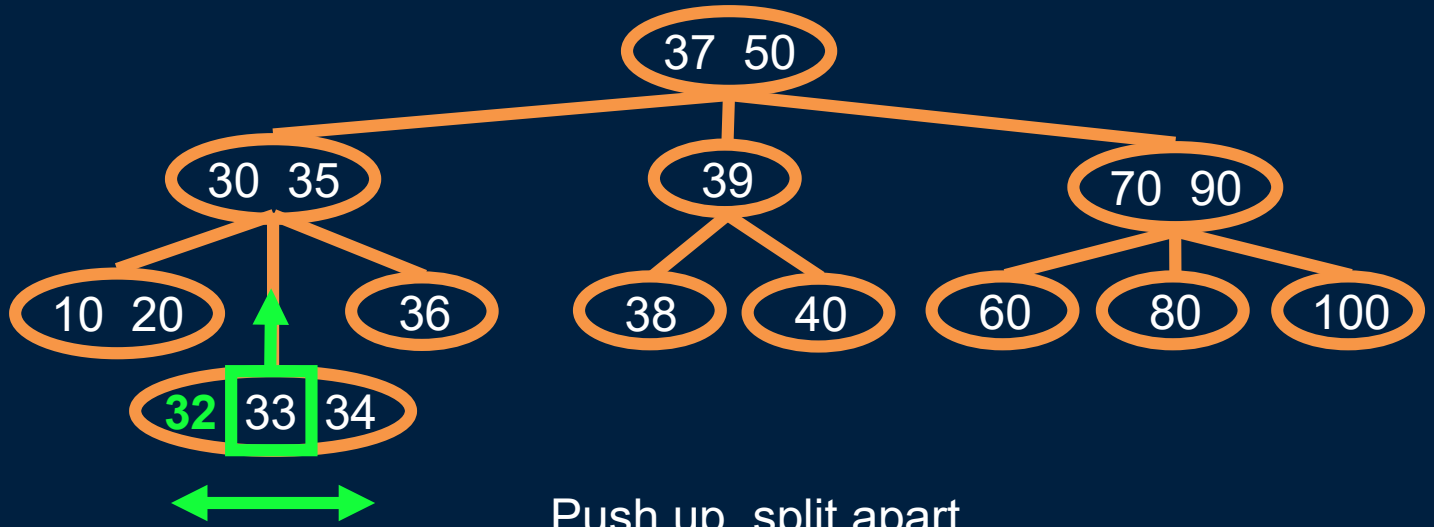
Insert 32

Insertion Examples



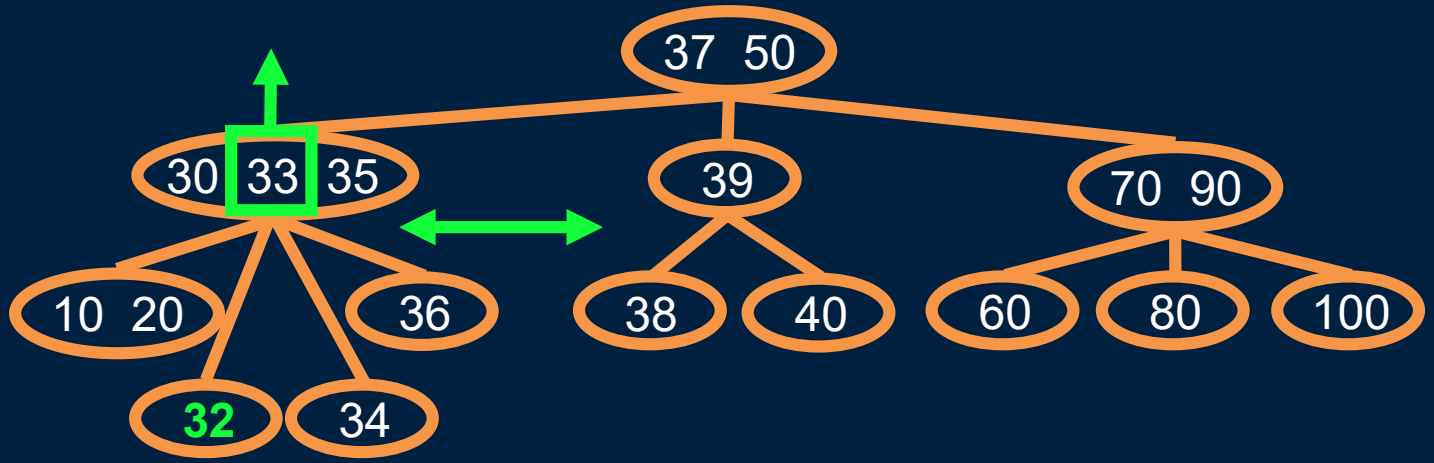
Insert 32

Insertion Examples



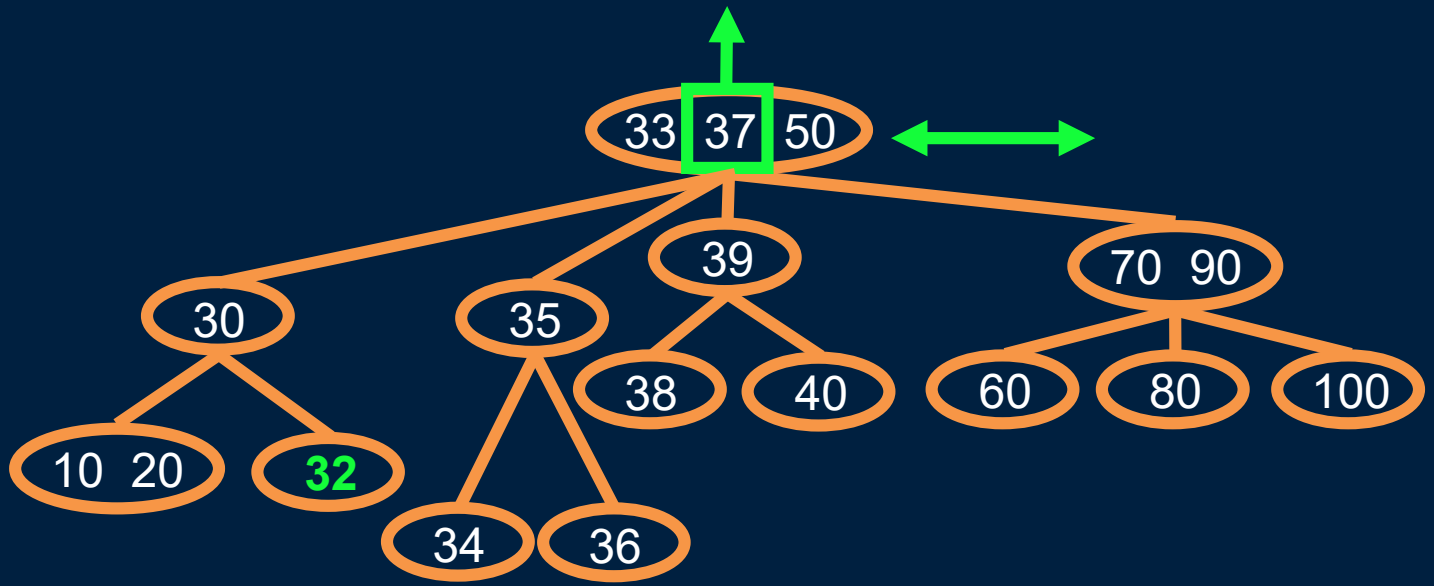
Push up, split apart

Insertion Examples



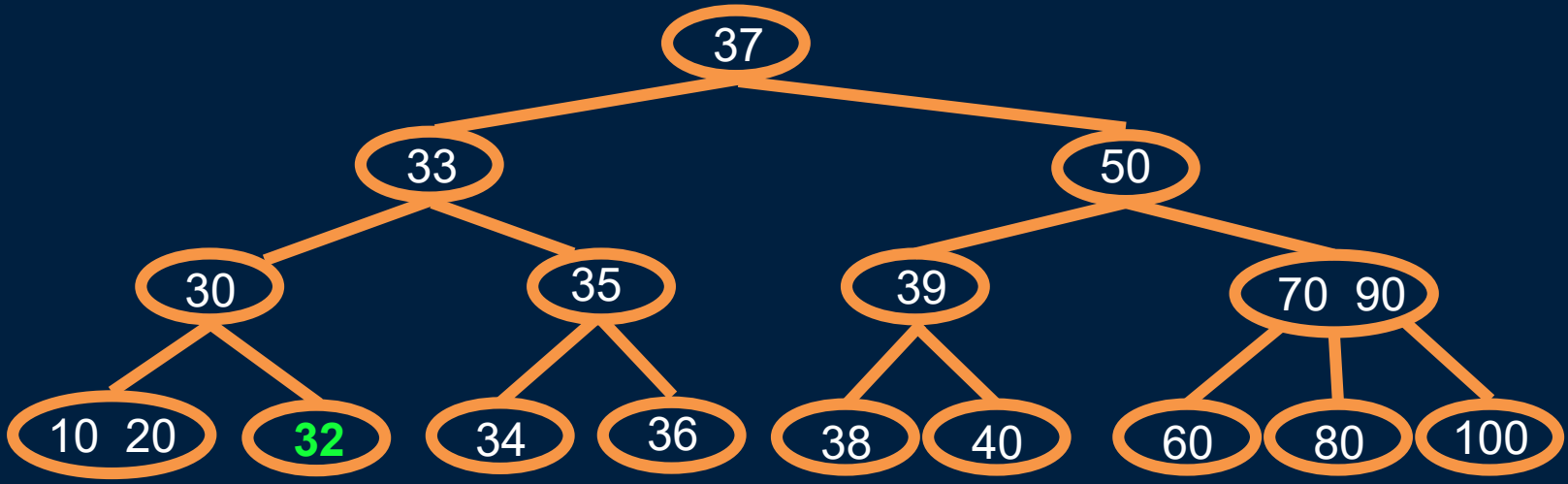
Push up, split apart

Insertion Examples



Push up, split apart

Insertion Examples



A new level is born!

Insertion Special Cases

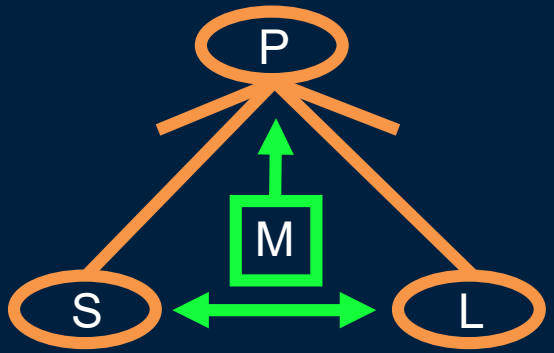
There are 3 cases of splitting for insertion:

- 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).
- Case 2 is **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.

Special Case: Splitting a Leaf Node

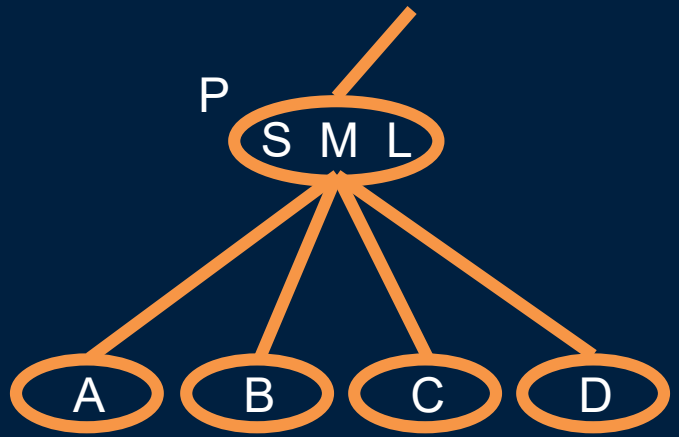
Leaf node overflow

Special Case: Splitting a Leaf Node (2)



Splitting a leaf node

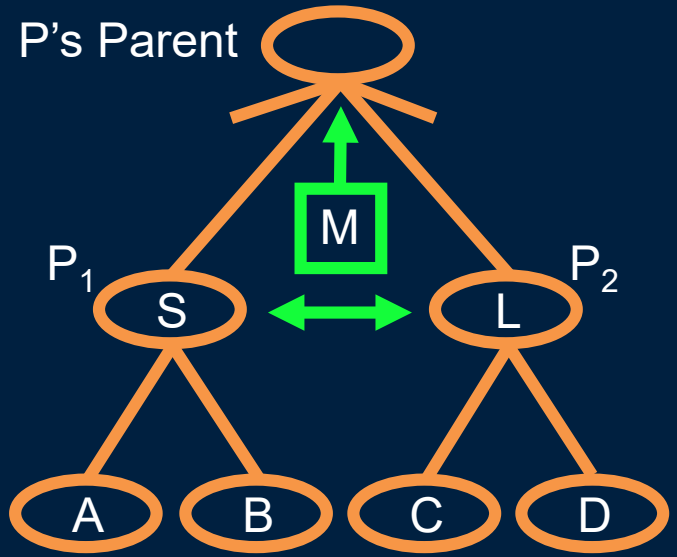
Special Case: Splitting an Interior Node



Interior node overflow

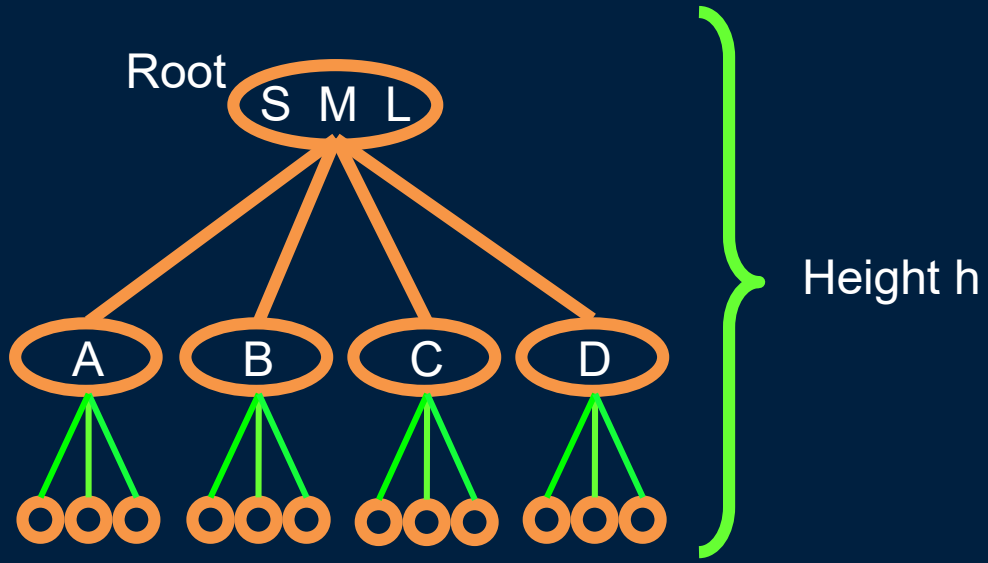
Splitting an internal node

Special Case: Splitting an Interior Node (2)



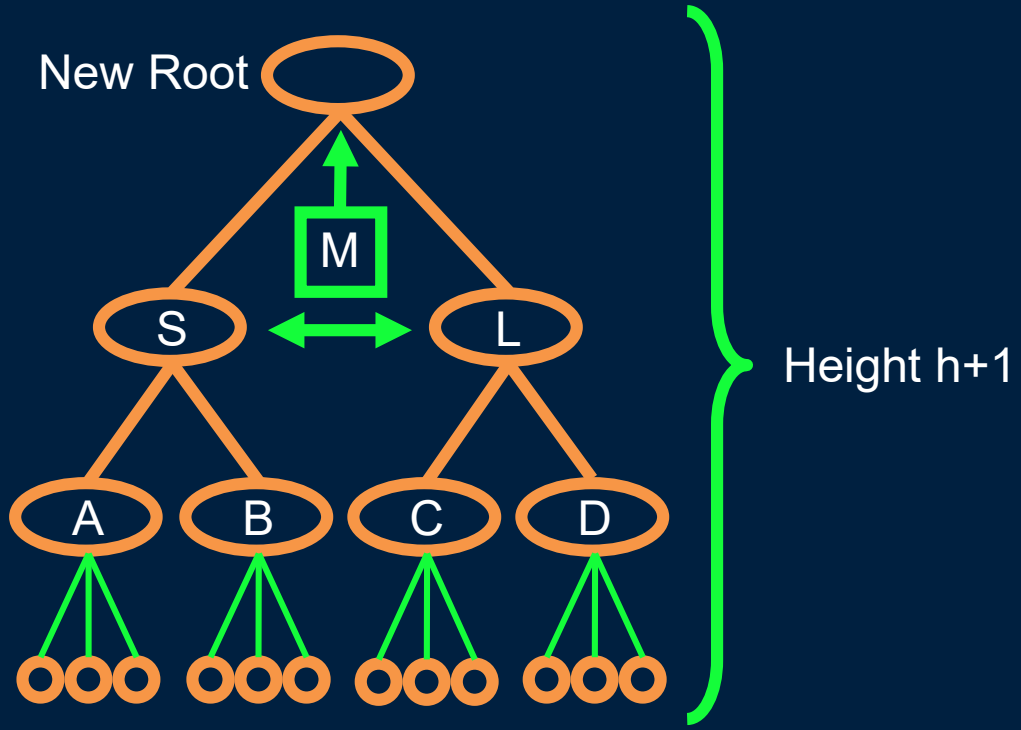
Splitting an internal node

Special Case: Splitting the Root Node



Splitting the root node

Special Case: Splitting the Root Node (2)



B-tree Insertion Practice Question

For a B-tree of ***order 1 (max. keys=2)***, insert the following keys in order:

- 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150

Deletion From a 2-3 Tree

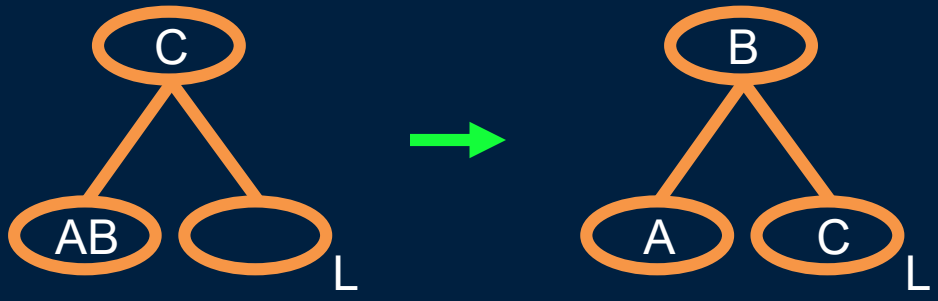
Algorithm:

- 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.

Deletion
Re-distributing values in Leaf Nodes

If deleting K from L causes L to be empty:

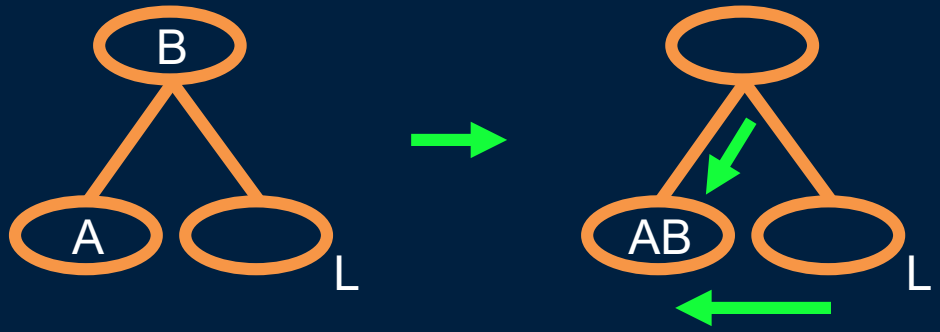
- Check siblings of now empty leaf.
- If sibling has two values, redistribute the values.



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.

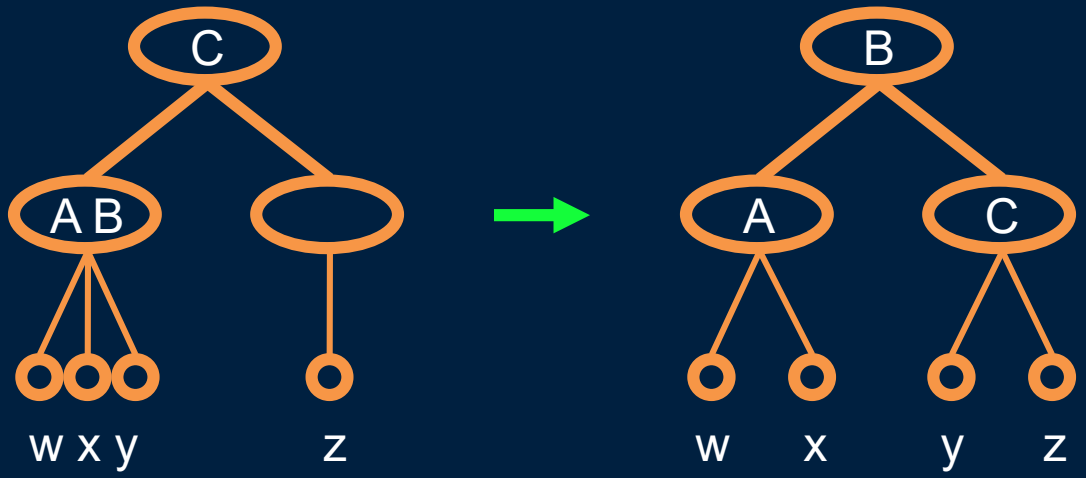


- The merging of L may cause the parent to be left without a value and only one child. If so, recursively apply deletion procedure to the parent.

Deletion
Re-distributing values in Interior Nodes

Re-distributing values in interior nodes:

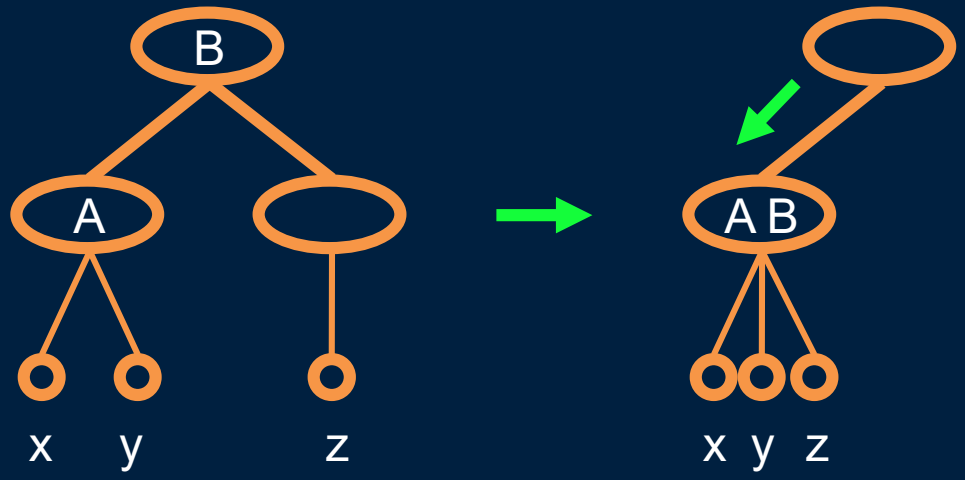
- If the node has a sibling with two values, redistribute the values.



Deletion
Merging Interior Nodes

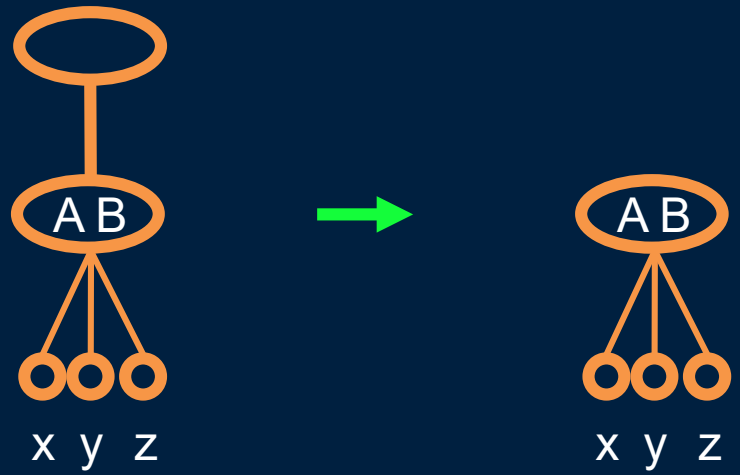
Merging interior nodes:

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

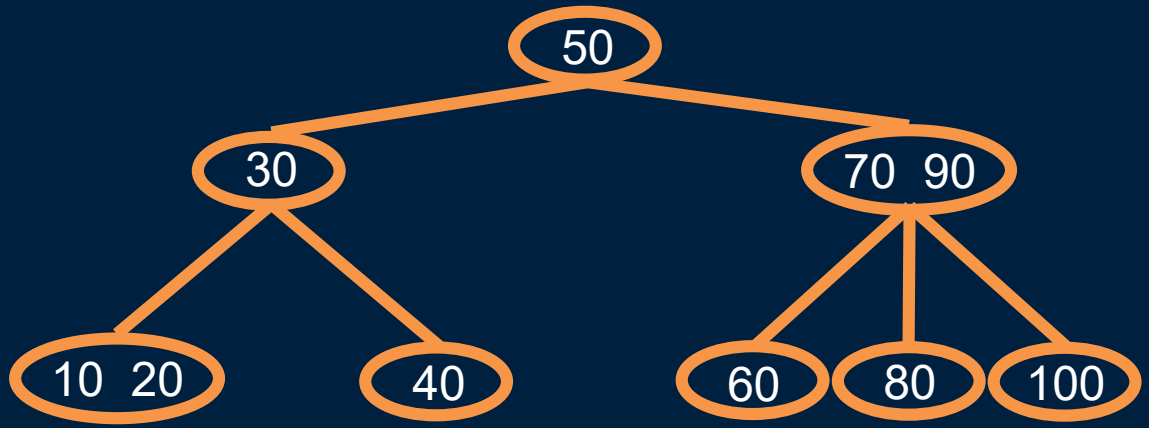


Deletion
Merging on the Root Node

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.

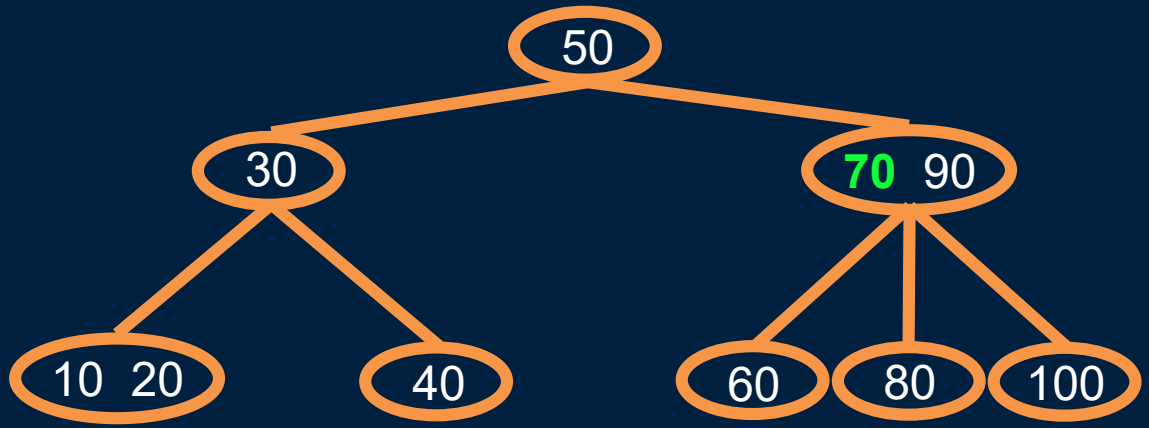


Deletion Examples



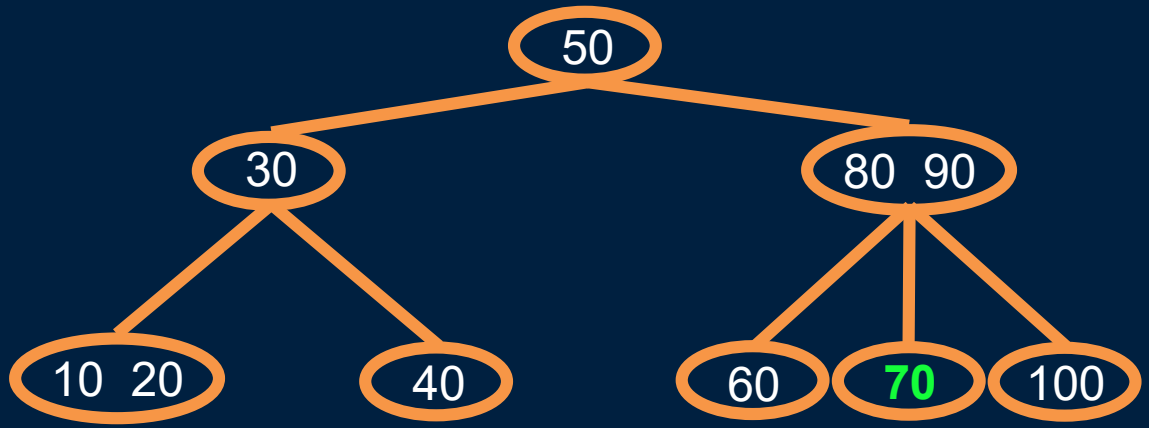
Original tree

Deletion Examples



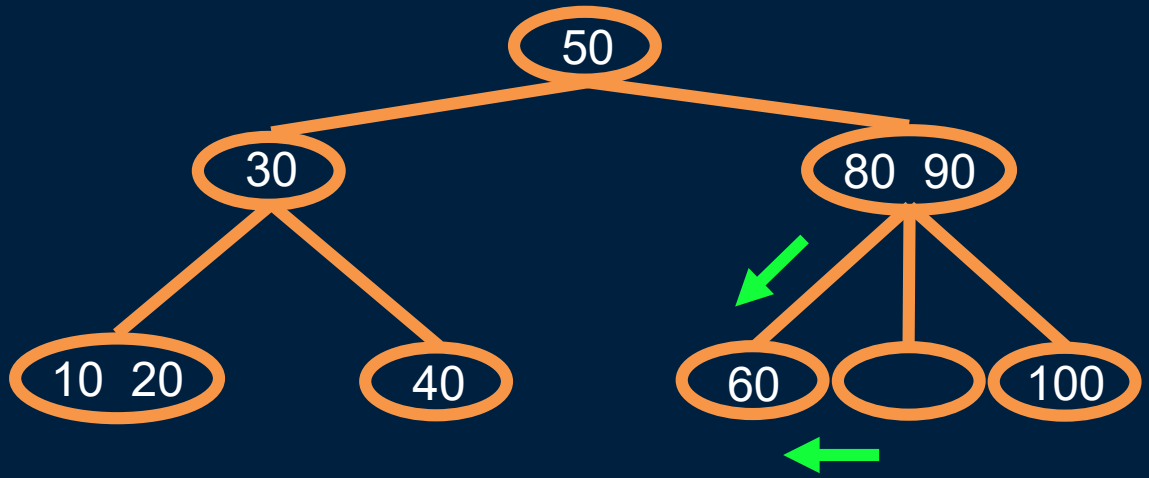
Delete 70

Deletion Examples



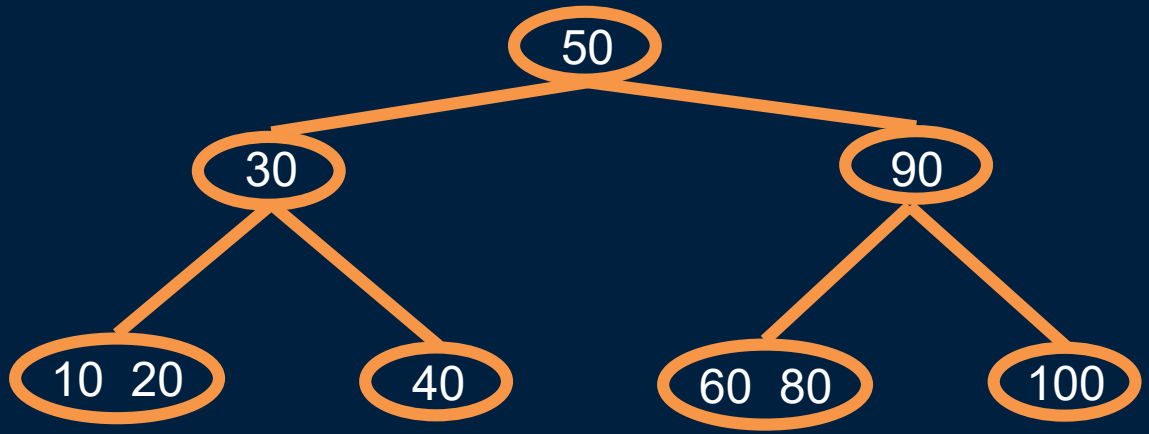
Swap with in-order successor

Deletion Examples



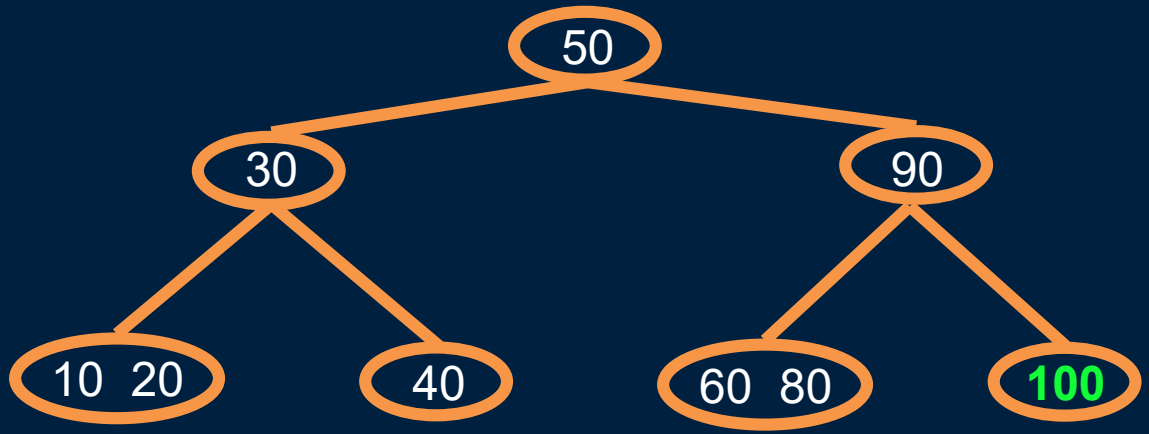
Merge and pull down

Deletion Examples



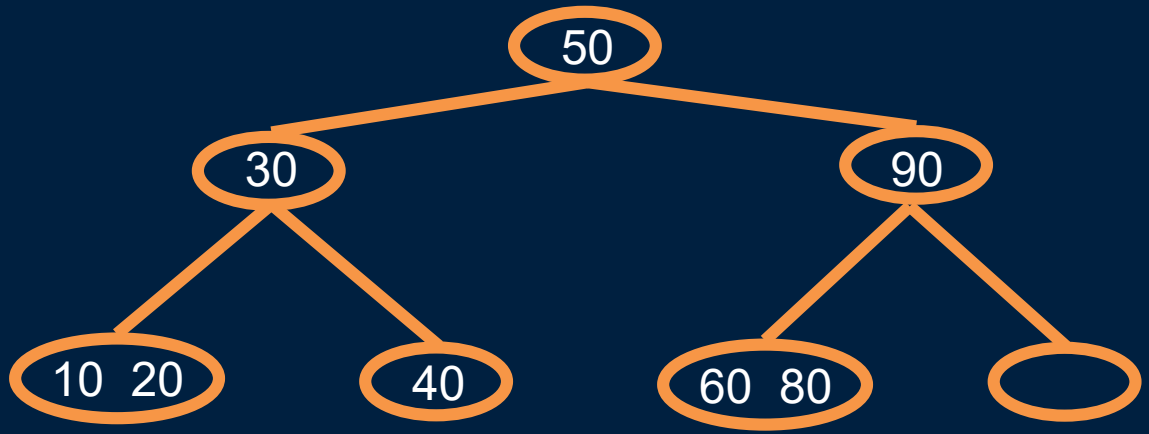
Done!

Deletion Examples



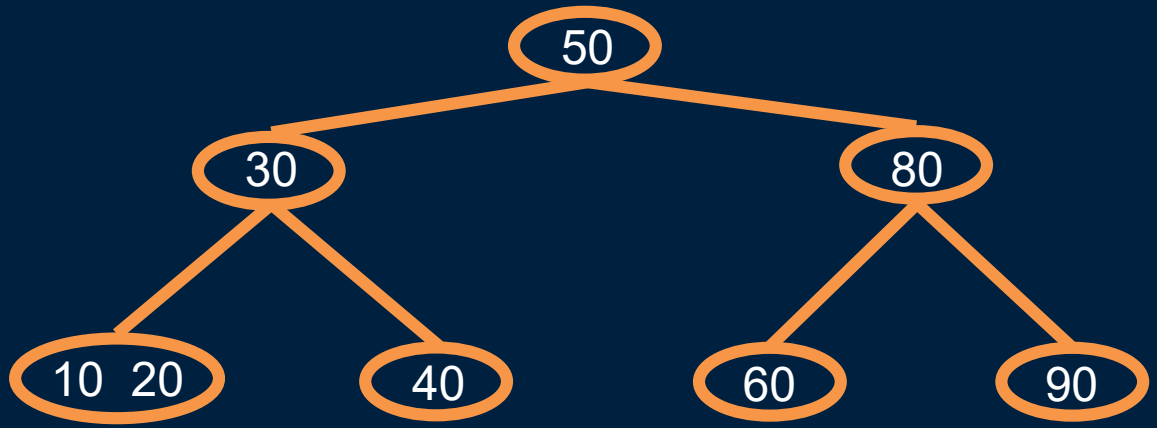
Delete 100

Deletion Examples



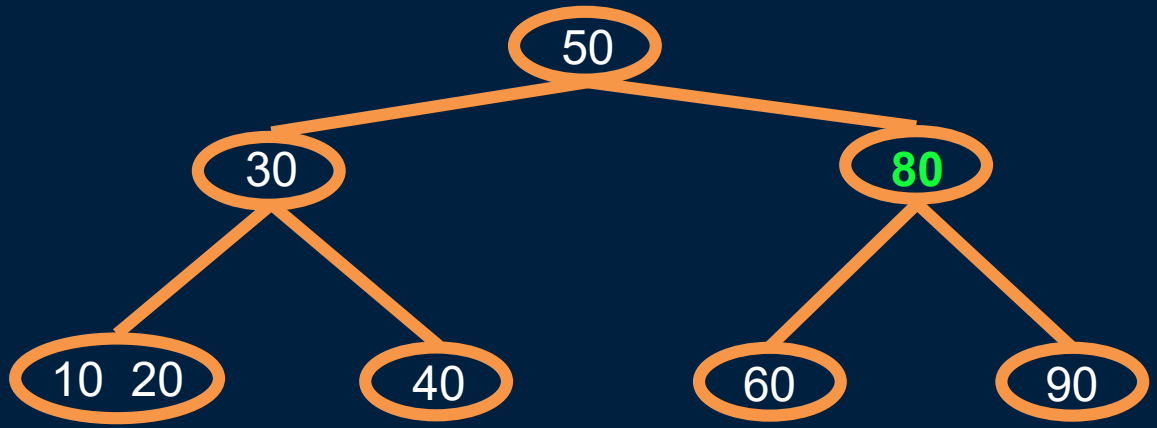
Redistribute

Deletion Examples



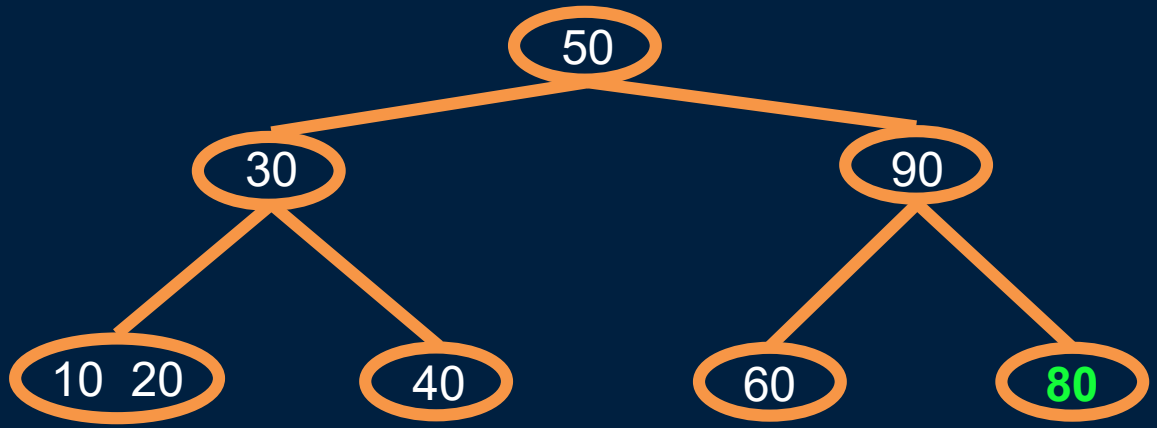
Done!

Deletion Examples



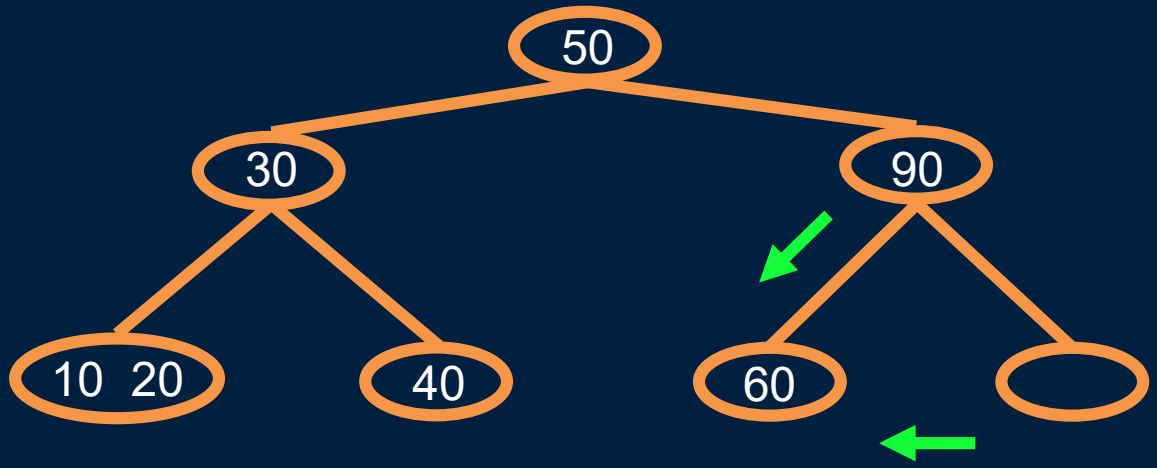
Delete 80

Deletion Examples



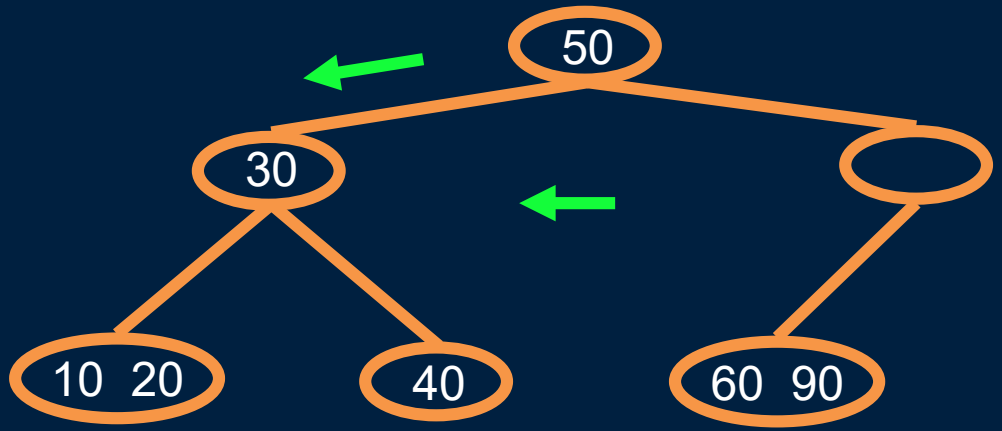
Swap with in-order successor

Deletion Examples



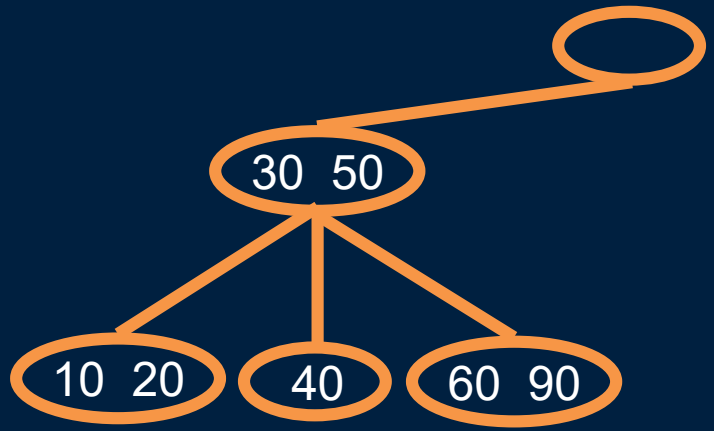
Merge and pull down

Deletion Examples



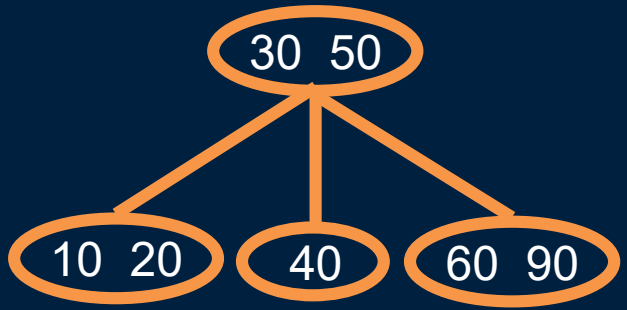
Merge and pull down

Deletion Examples



Merge and pull down

Deletion Examples



Done

B-tree Deletion Practice Question

Using the previous tree constructed by inserting into a B-tree of ***order 1 (max. keys=2)*** the keys:

- 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150

Delete these keys (in order):

- 40
- 70
- 80

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.

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, the pointer to the data record is 8 bytes, and the child pointers are 8 bytes.

Answer:

Assuming no header information is kept in blocks:

node size = keySize*numKeys + dataPtrSize*numKeys
+ childPtrSize*(numKeys+1)

Let k=numKeys.

size of one node = 4*k + 8*k + 8*(k+1) <= 4096

k = 204 keys

Maximum order is 102.

B-tree Question

Question: Given a block of 4096 bytes, calculate the maximum number of keys in a node 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.

A) 18

B) 36

C) 340

D) 680

Advantages of B-trees

The advantages of a B-tree 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

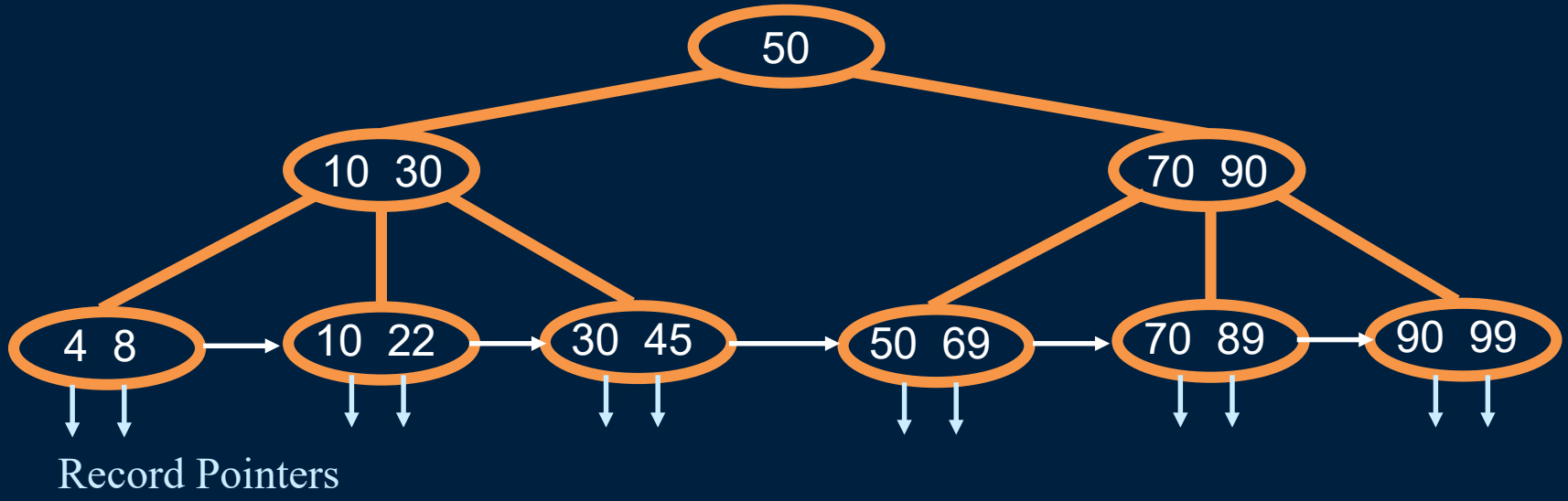
A ***B+-tree*** is a multi-level index structure like a B-tree except that *all* data is stored at the leaf nodes of the resulting tree instead of within the tree itself.

- 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 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

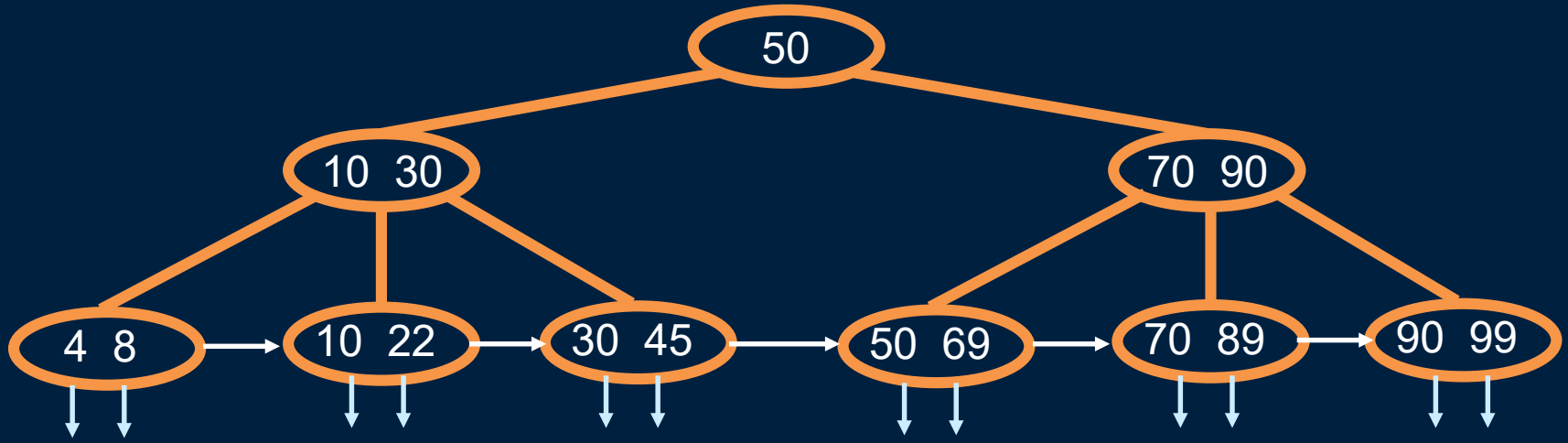
The general algorithms for inserting and deleting from a B+-tree are similar to B-trees except for one important difference:

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 from leaves are **copies**.

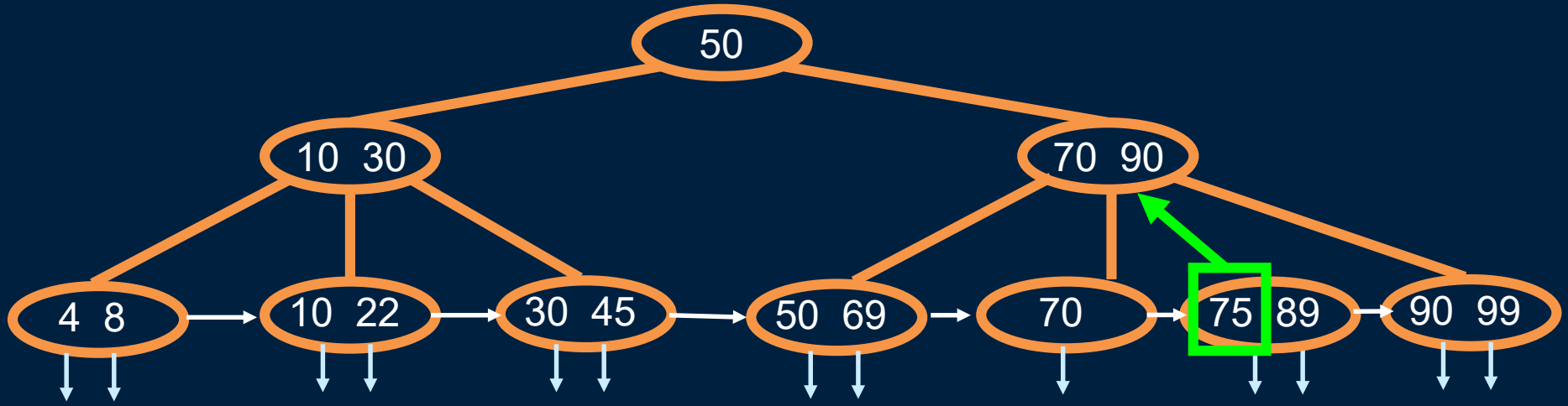
- All non-leaf levels do not store actual data, they are simply a hierarchy of multi-level index to the data.

B+-tree Insert Example



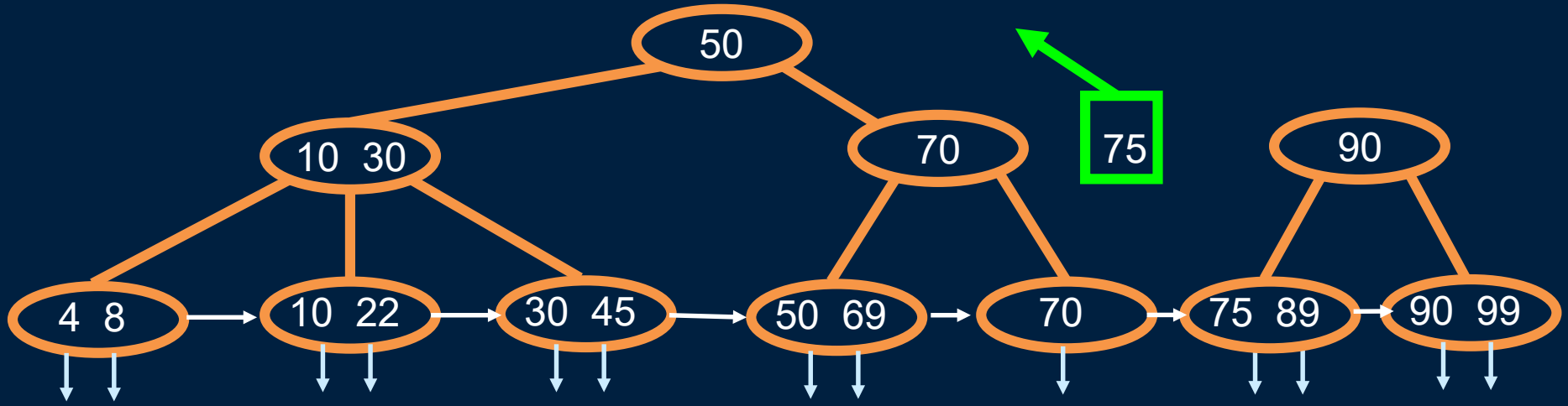
Insert 75

B+-tree Insert Example (2)



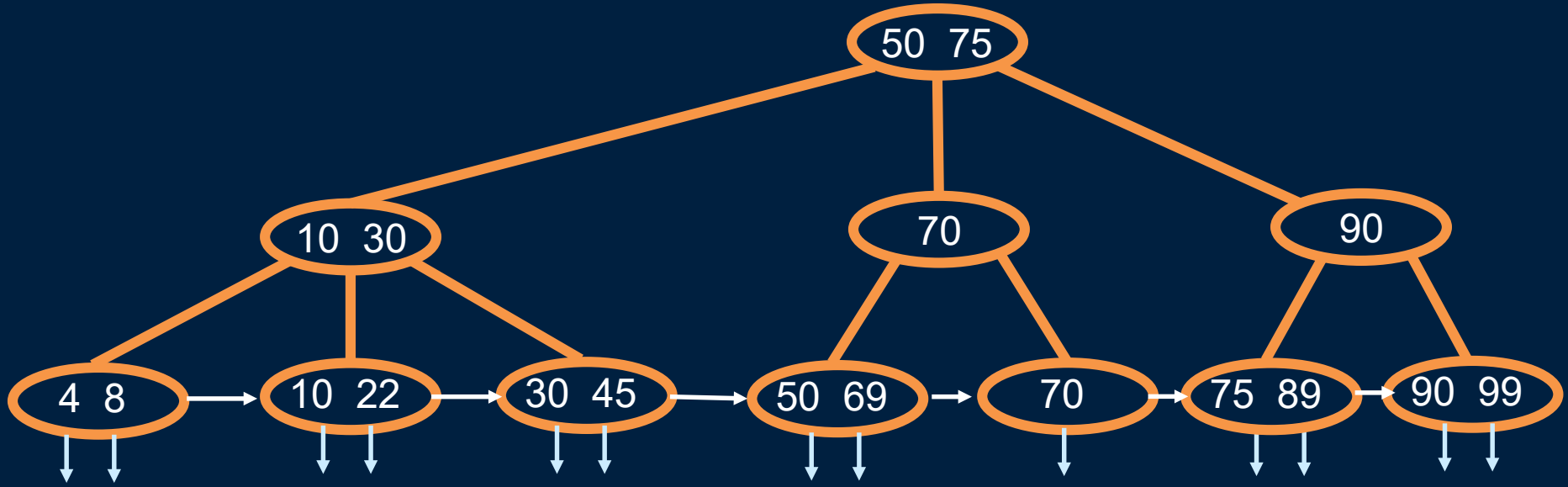
75 goes in 2nd last block.
Split block to handle overflow.
Promote 75. Note that 75 stays in a leaf!

B+-tree Insert Example (3)



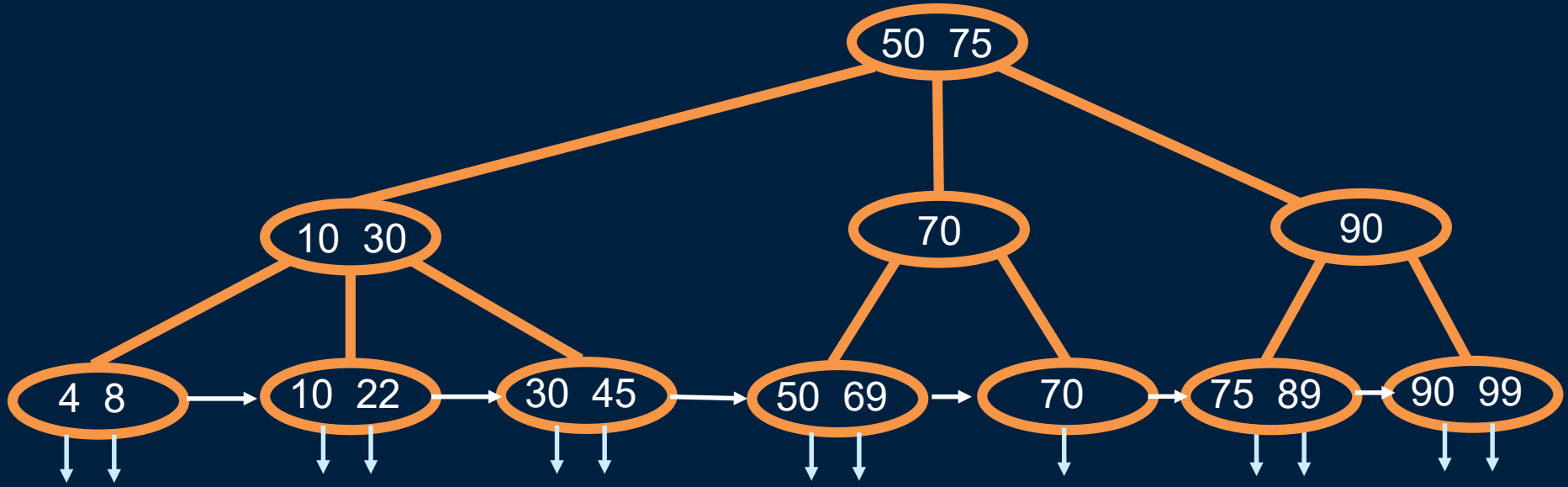
75 goes in 2nd last block.
Split block to handle overflow.
Promote 75. Note that 75 stays in a leaf!

B+-tree Insert Example (4)



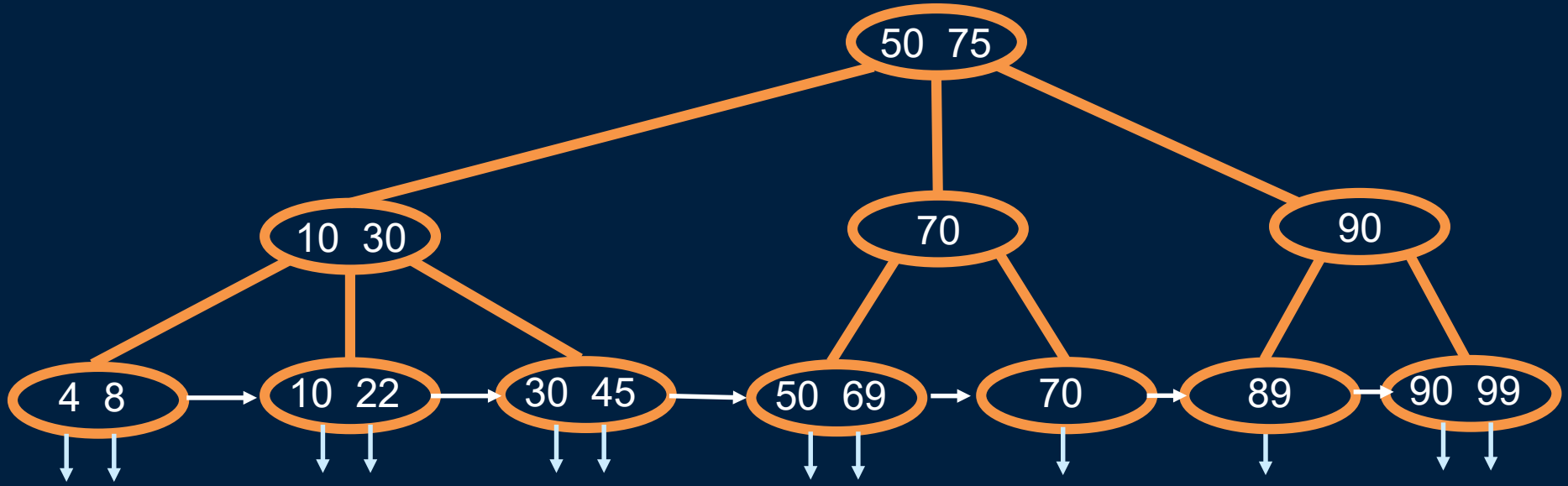
Insertion done!

B+-tree Delete Example



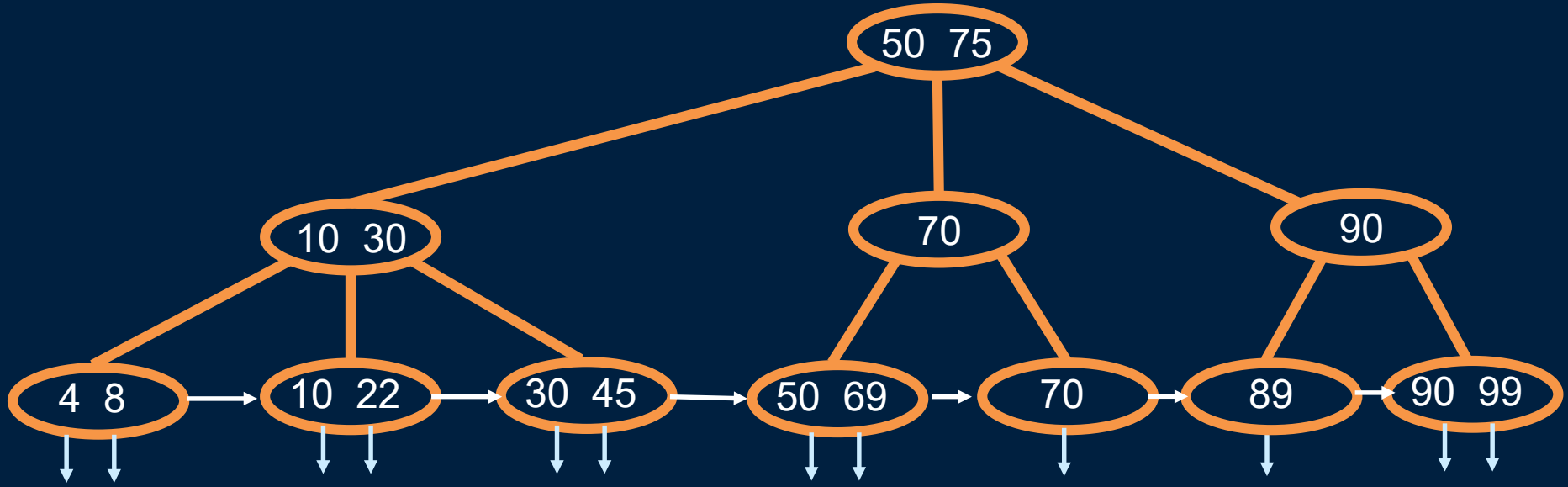
Delete 75.

B+-tree Delete Example (2)



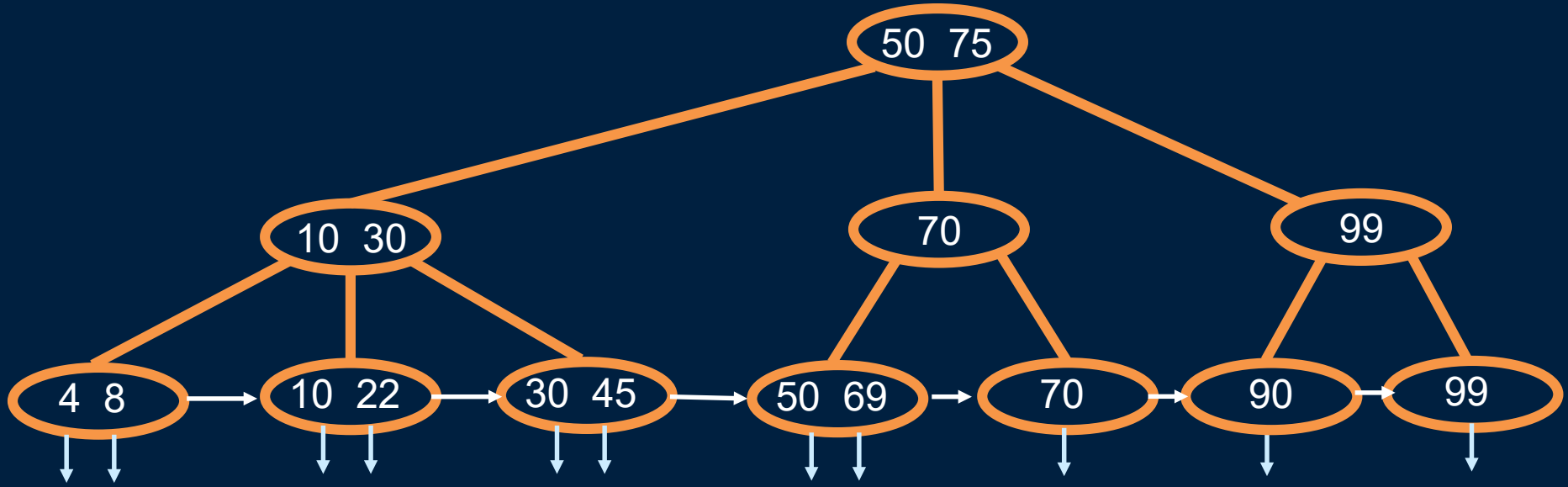
Remove from leaf node.
No other updates.

B+-tree Delete Example 2



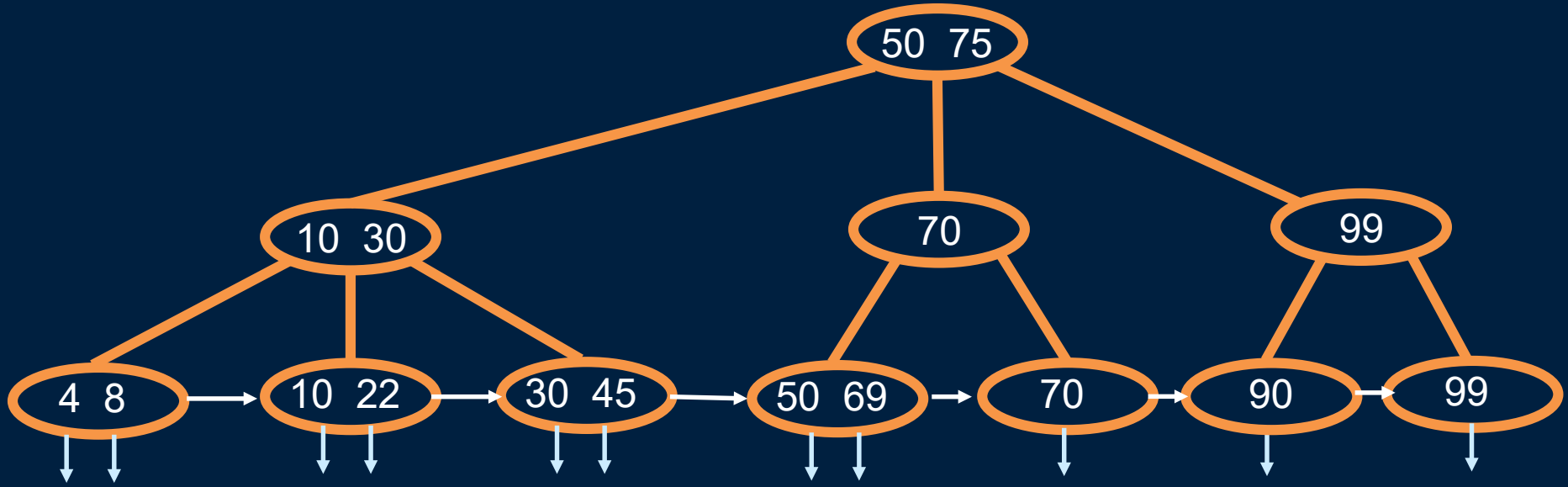
Delete 89.

B+-tree Delete Example 2 (2)



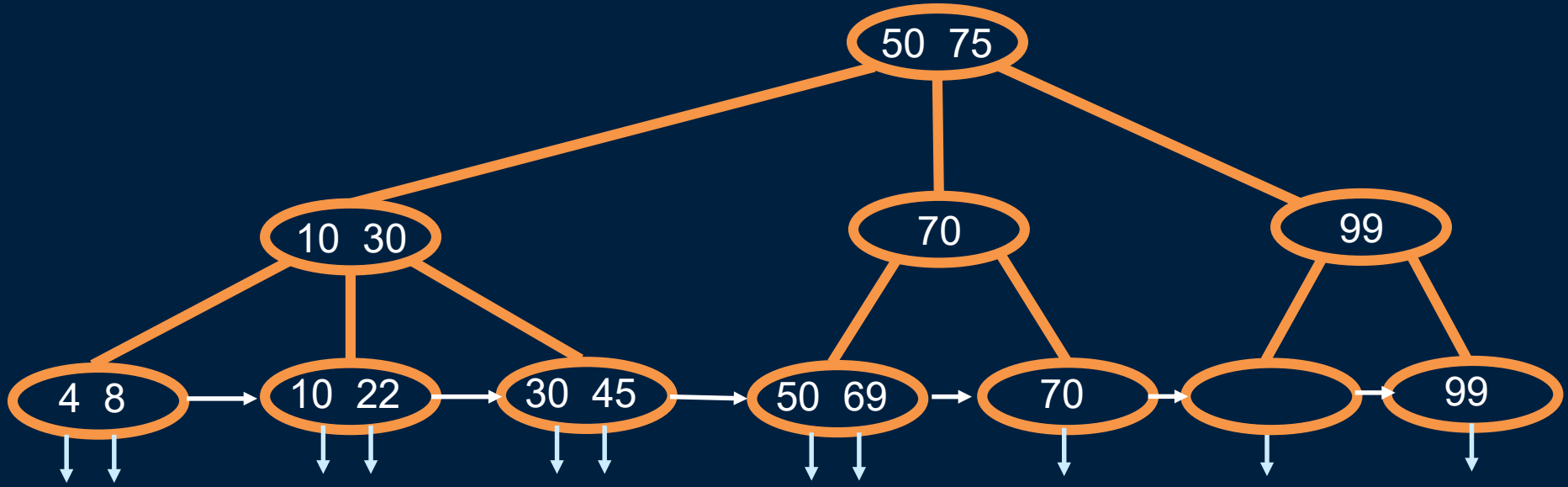
Redistribute keys 90 and 99.

B+-tree Delete Example 3



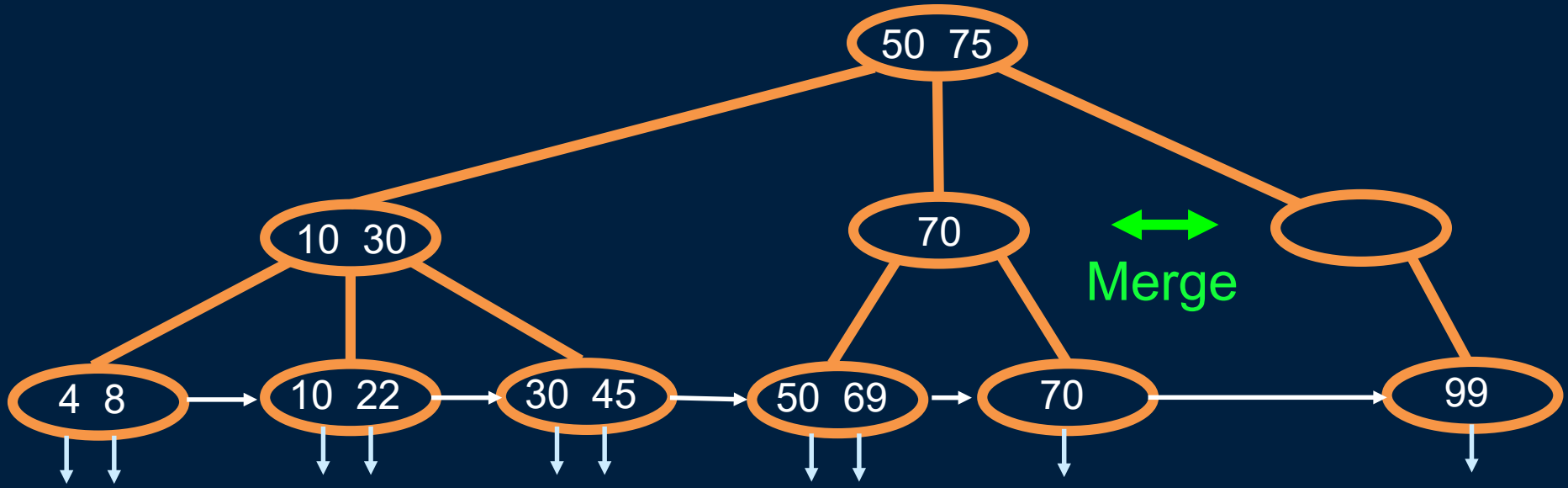
Delete 90.

B+-tree Delete Example 3 (2)



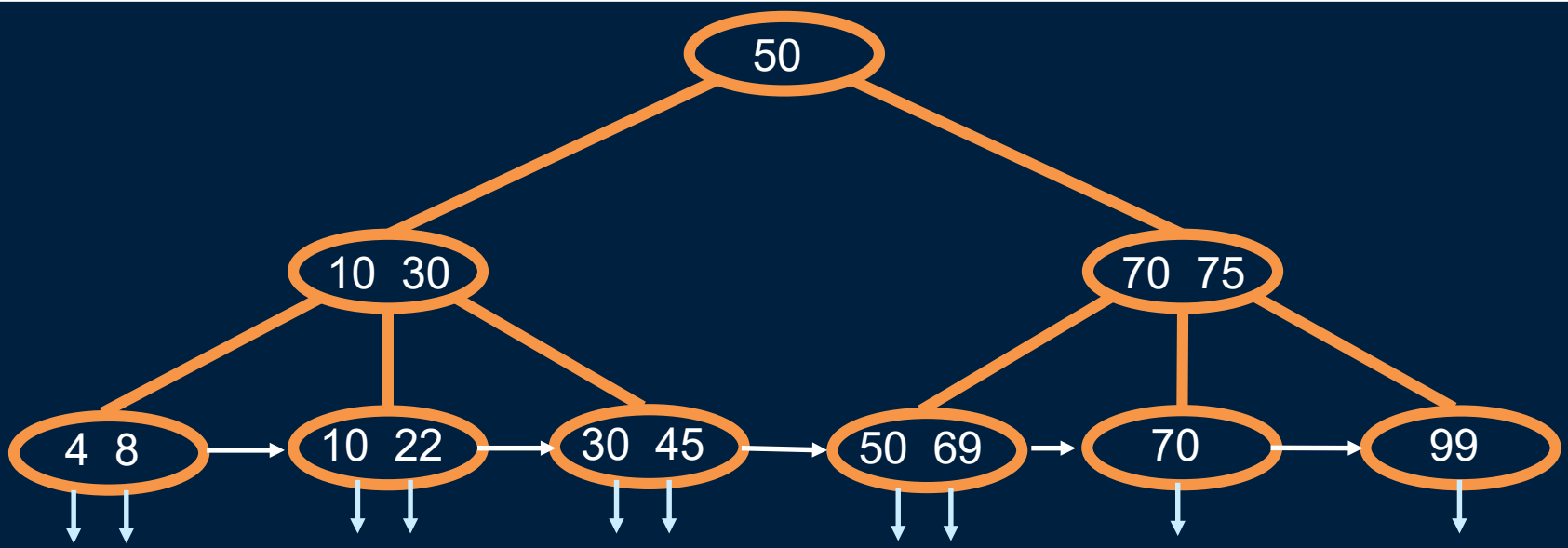
Empty leaf node. Merge with sibling.

B+-tree Delete Example 3 (3)



Empty interior node. Merge with sibling.

B+-tree Delete Example 3 (4)



Bring down 75 from parent node. Done.

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?
  - a) 110 b) 120 c) 130 d) 140 e) 150

Show the tree after deleting the following keys:

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

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.

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 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.

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!

B+-trees Discussion

By isolating the data records in the leaves, we also introduce 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.

B-trees
Summary

A ***B-tree*** is a search tree where each node has >= ***n*** data values and <= ***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 complex.
- Note difference between B+-tree and B-tree for insert/delete!

B+-trees are a good index structure because they can be searched/updated in logarithmic time, manage record pointer allocation on blocks, and support sequential access.

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 index method due to their time/space efficiency and logarithmic time for insertions/deletions.

Other objectives:

- Calculate query access times using B-trees indexes.
- Compare/contrast B-trees and B+-trees.


UBC
THE UNIVERSITY OF BRITISH COLUMBIA