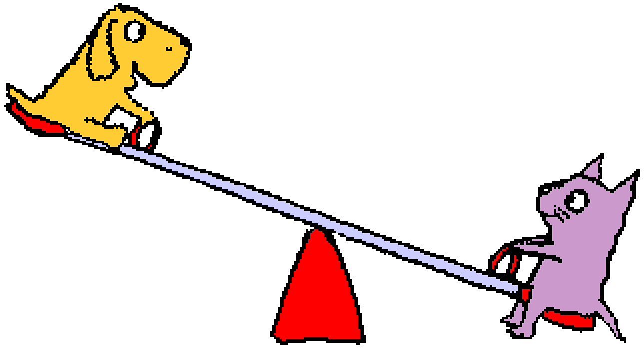
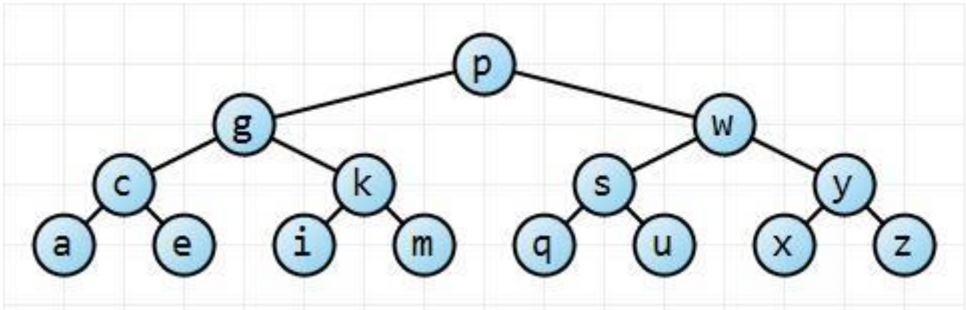


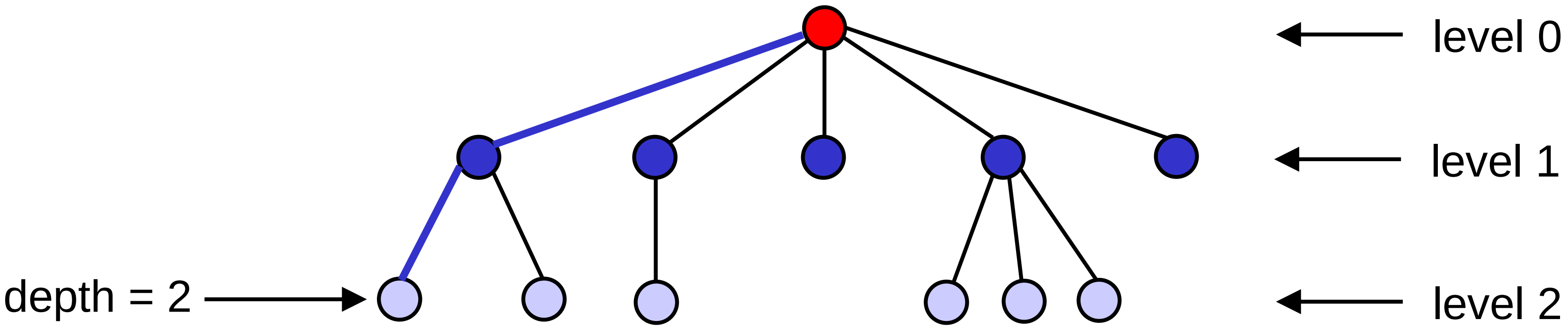
Balanced Search Trees



Computer Science 112
Boston University

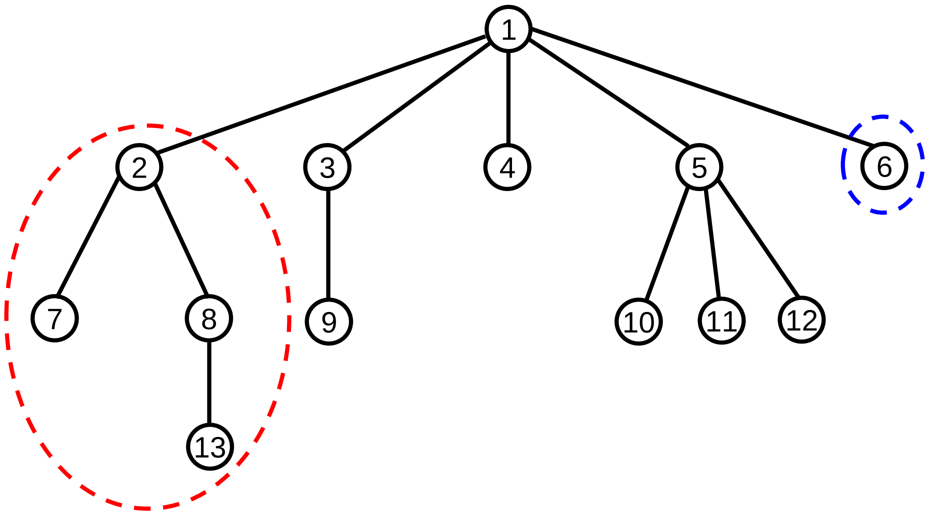
Christine Papadakis-Kanaris

Recall: Path, Depth, Level, and Height



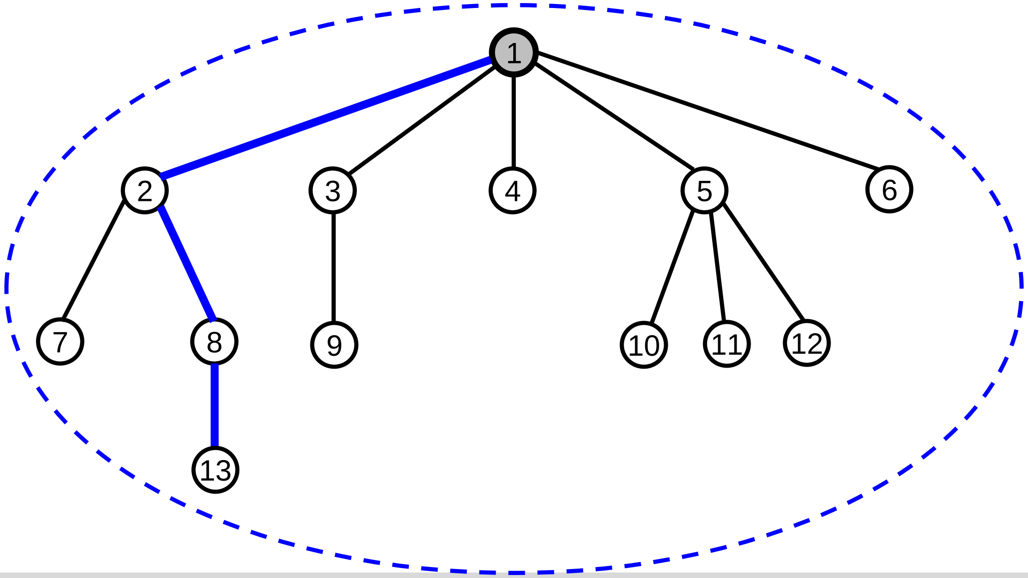
- There is exactly one *path* (one sequence of edges) connecting each node to the root.
- *depth* of a node = # of edges on the path from it to the root
- Nodes with the same depth form a *level* of the tree.
- The *height* of a tree is the ***maximum depth of its nodes***.
    - example: the tree above has a height of 2

Recall: A Tree is a Recursive Data Structure



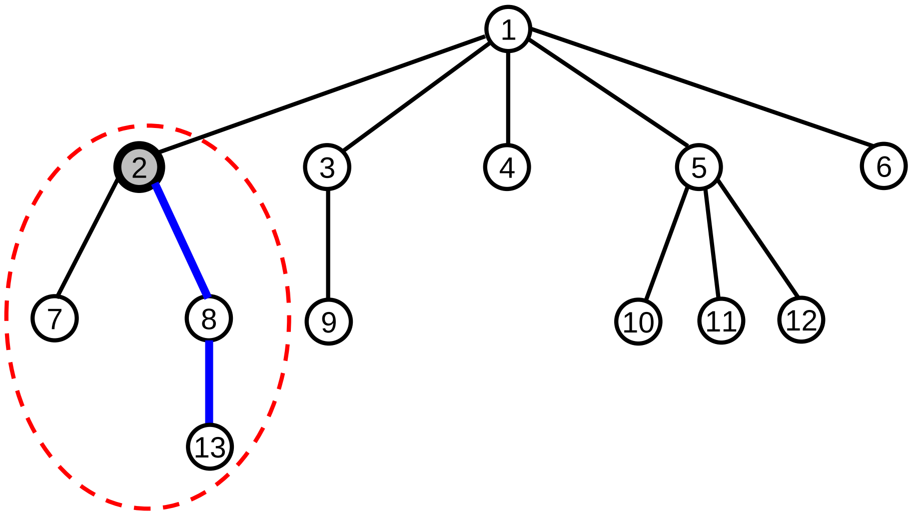
- **Each node in the tree is the root of a smaller tree!**
  - refer to such trees as *subtrees* to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node

Recall: A Tree is a Recursive Data Structure



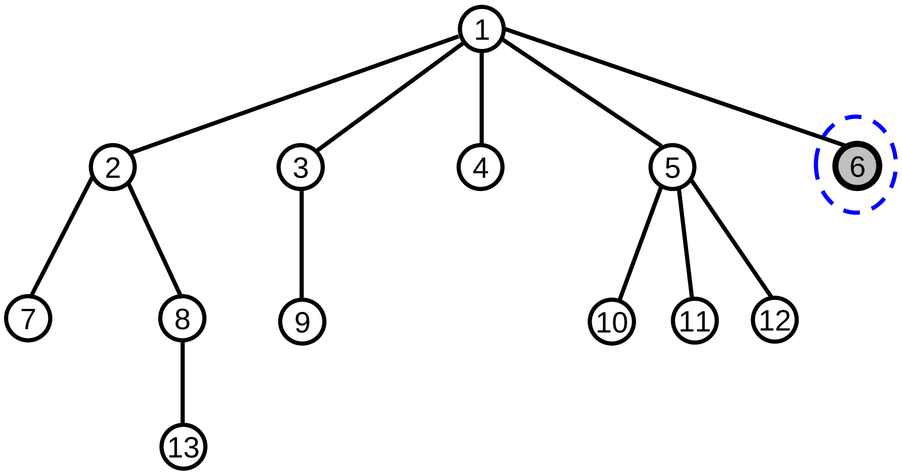
- Height 3

Recall: A Tree is a Recursive Data Structure



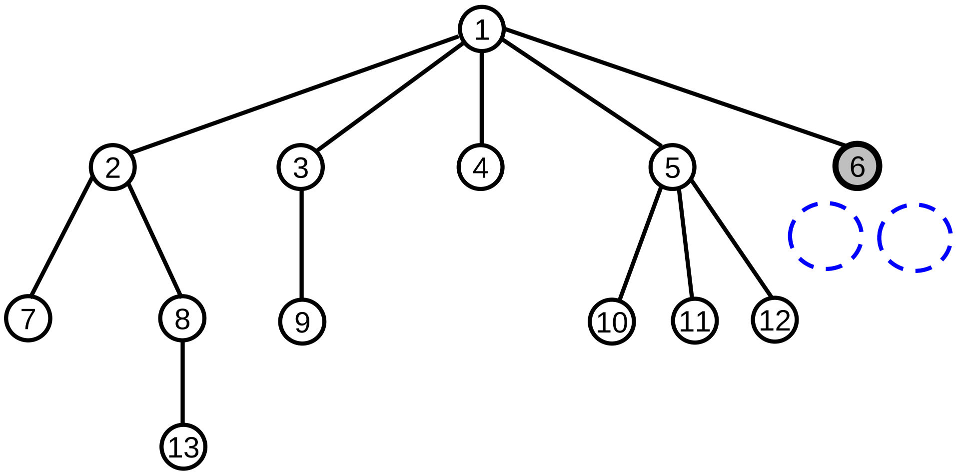
- Height 2

Recall: A Tree is a Recursive Data Structure



- Height 0

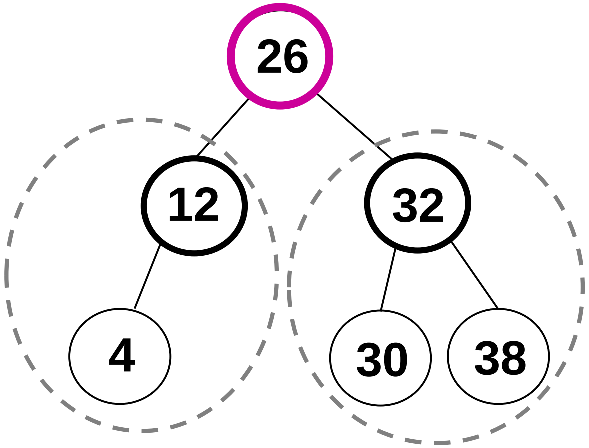
Recall: A Tree is a Recursive Data Structure



- What is the height of empty nodes? -1

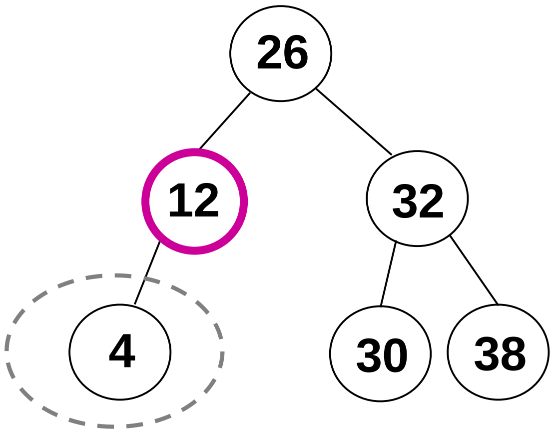
Defining
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
    - example:
        - 26: both subtrees have a height of 1



Defining
Balanced Trees

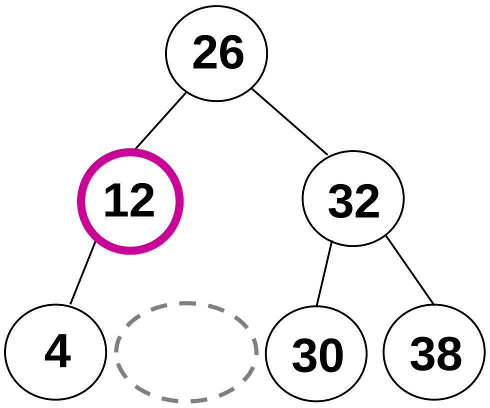
- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0



Defininf

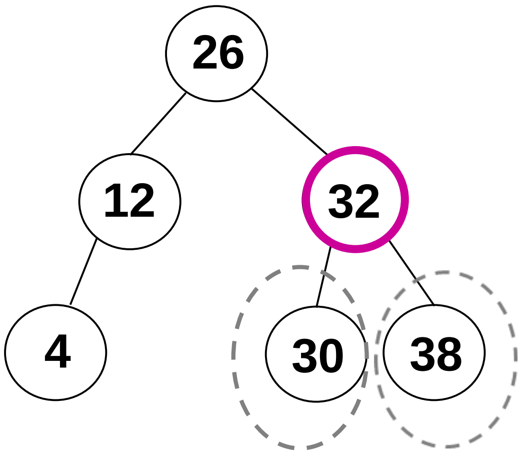
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)



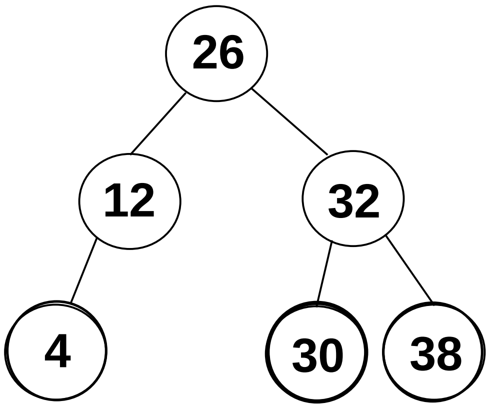
Defining
Balanced Trees

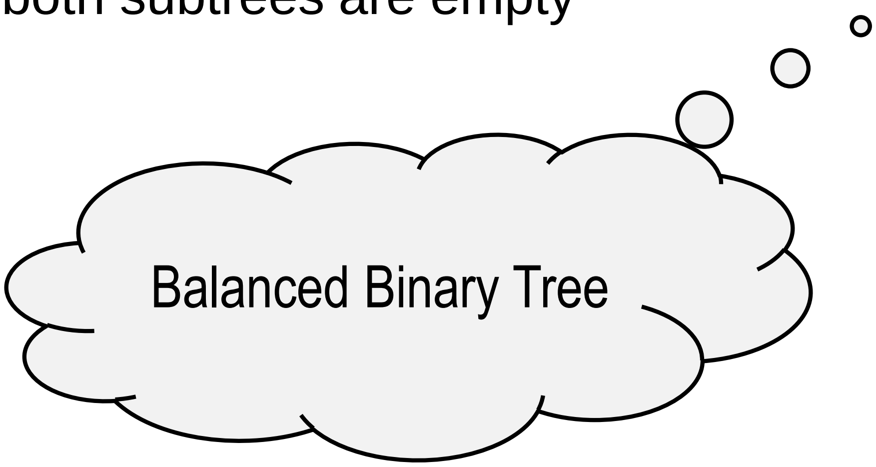
- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0



Defining
Balanced Trees

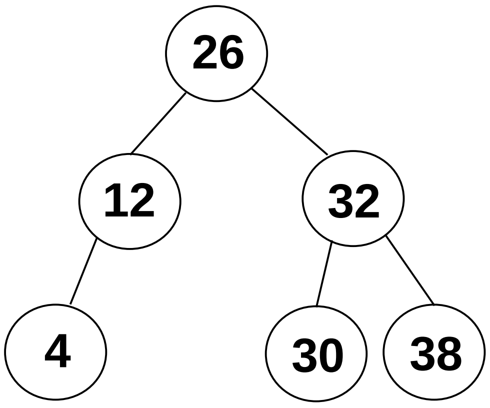
- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0
    - all leaf nodes: both subtrees are empty





Efficiency of
Balanced Trees

- A tree is *balanced* if, for *each* of its nodes, the node's subtrees have the same height or have heights that differ by 1.
  - example:
    - 26: both subtrees have a height of 1
    - 12: left subtree has height 0
      right subtree is empty (height = -1)
    - 32: both subtrees have a height of 0
    - all leaf nodes: both subtrees are empty

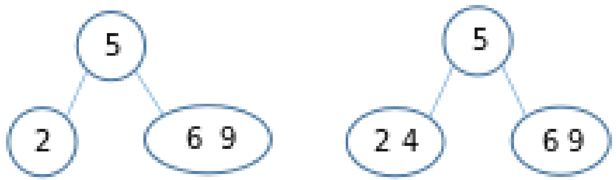


- For a balanced tree with n nodes, height = $O(\log n)$
  - each time that you follow an edge down the longest path, you cut the problem size roughly in half!

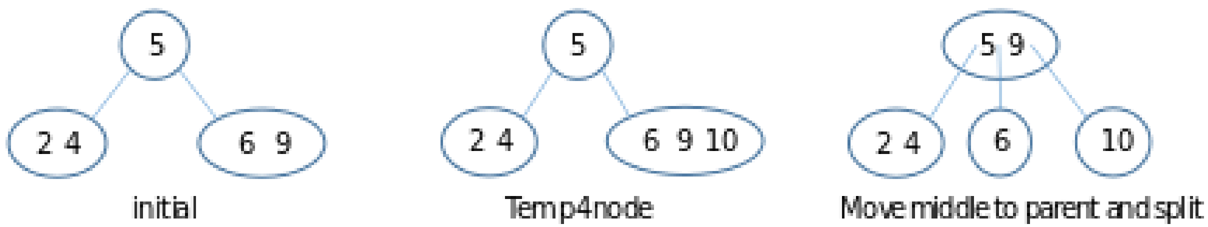
- Therefore, for a *balanced* binary search tree, the worst case for search / insert / delete is $O(h) = O(\log n)$
  - the "best" worst-case time complexity

2-3 Trees

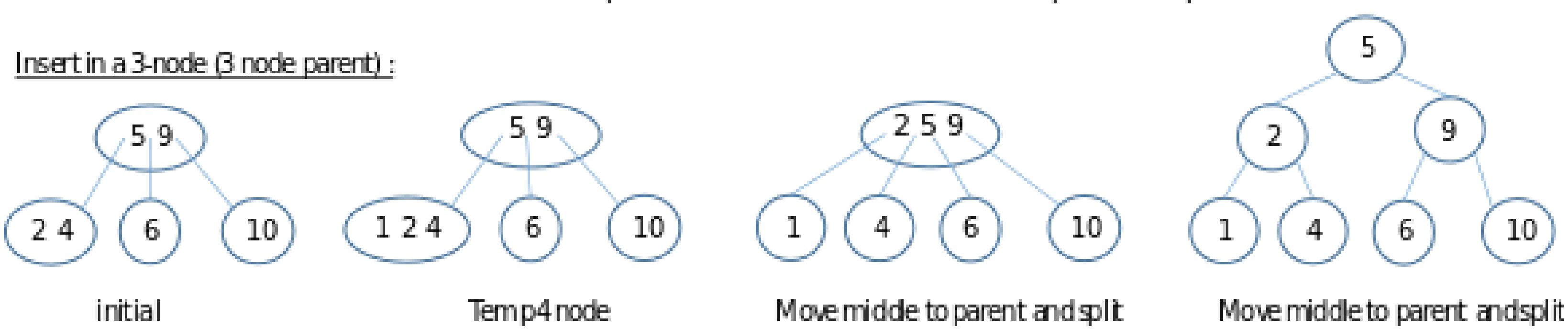
Insert in a 2-node :



Insert in a 3-node (2 node parent) :



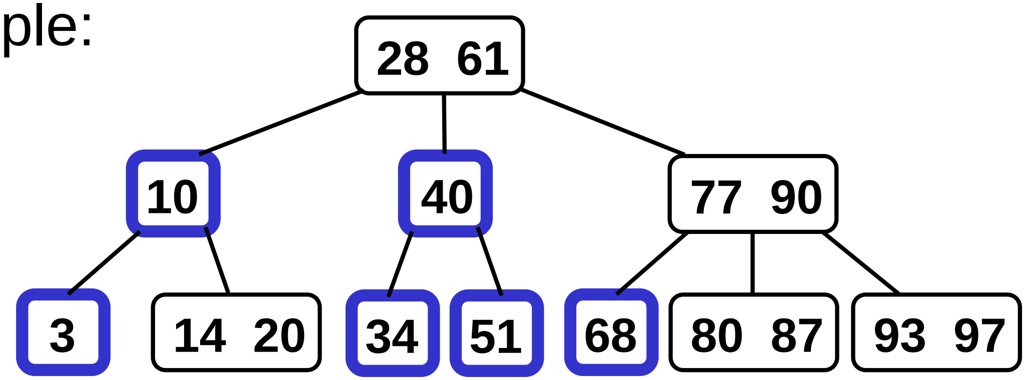
Insert in a 3-node (3 node parent) :



2-3 Trees

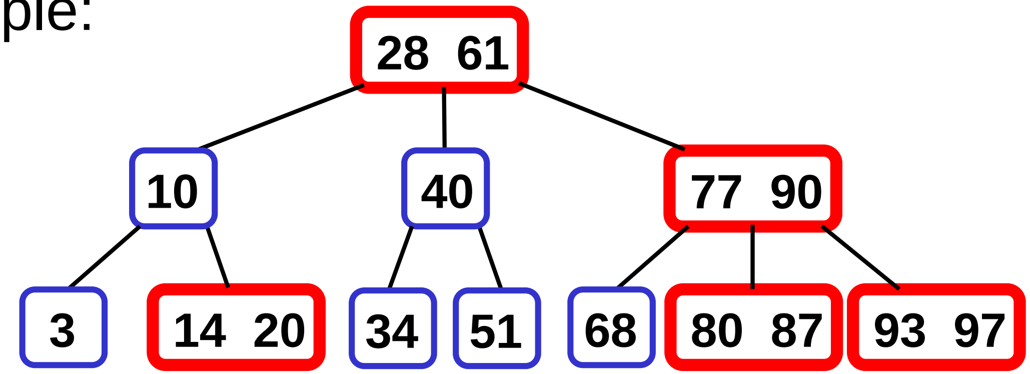
- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item <u>and</u> *exactly* 0 **or** 2 children

- Example:



2-3 Trees

- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and exactly 0 or 2 children
    - a **3-node**, which contains two data items and *exactly* 0 or 3 children
- Example:

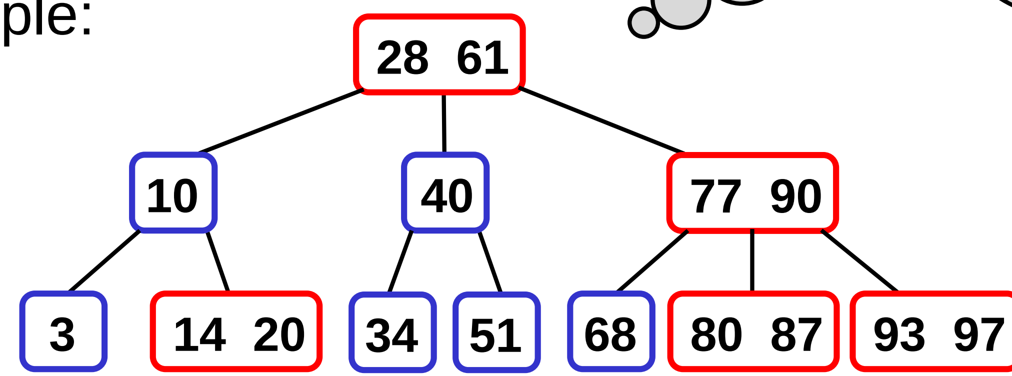


2-3 Trees

- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height sub
  - each node is either
    - a **2-node**, which contains on ildren
    - a **3-node**, which contains tw
      children

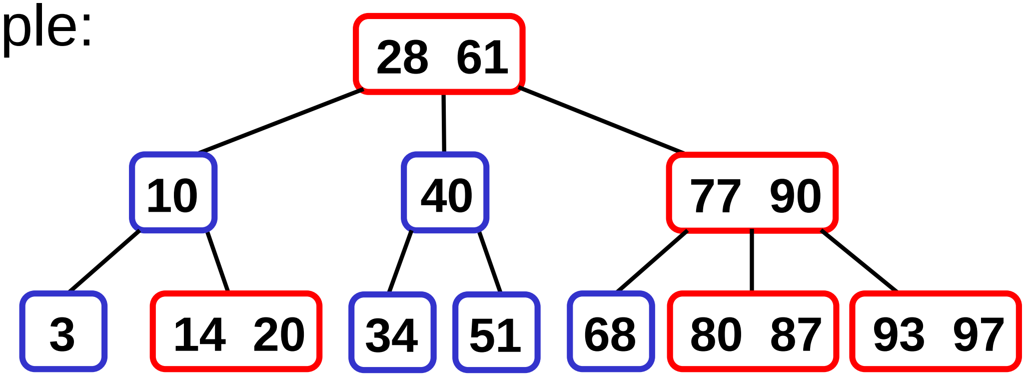
Assume the data in the node are the keys then…

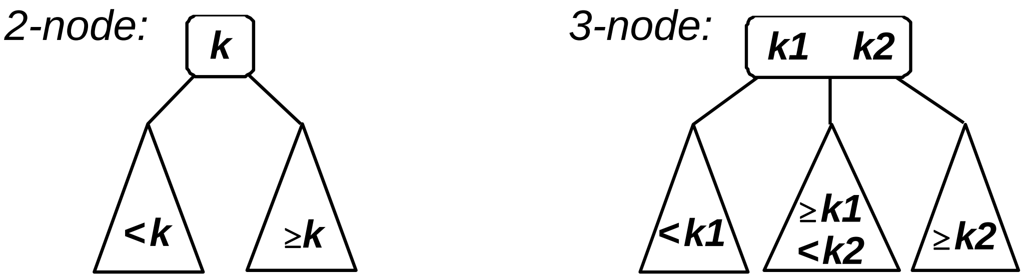
- Example:



2-3 Trees

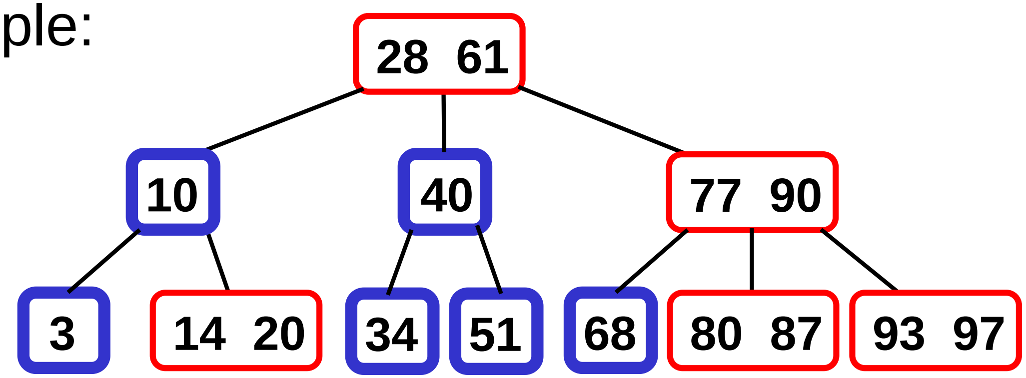
- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and exactly 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
  - the keys form a search tree
- Example:

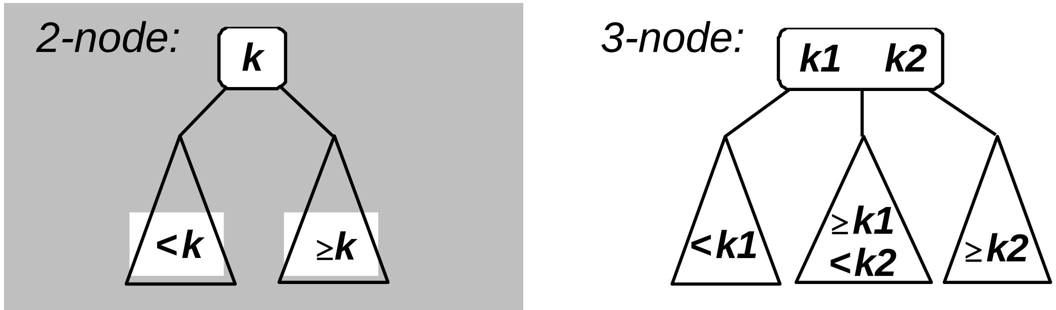




2-3 Trees

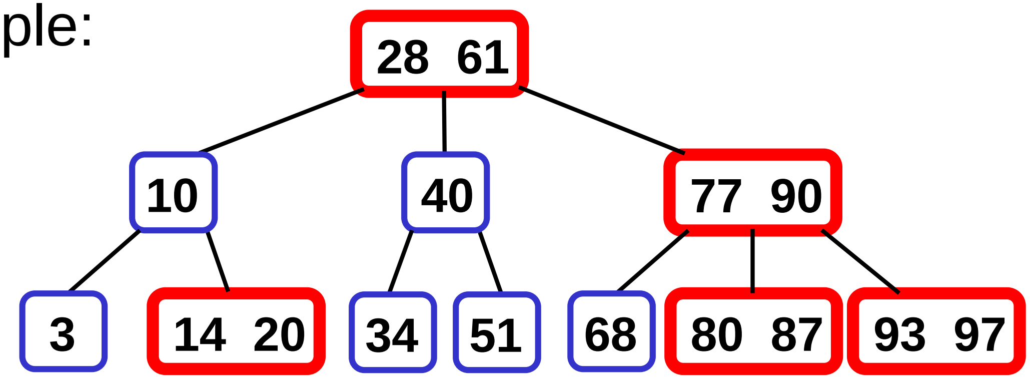
- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
  - the keys form a search tree
- Example:

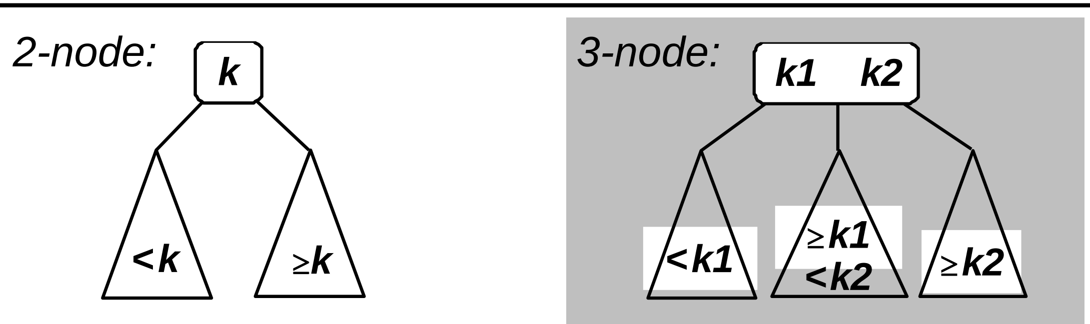




2-3 Trees

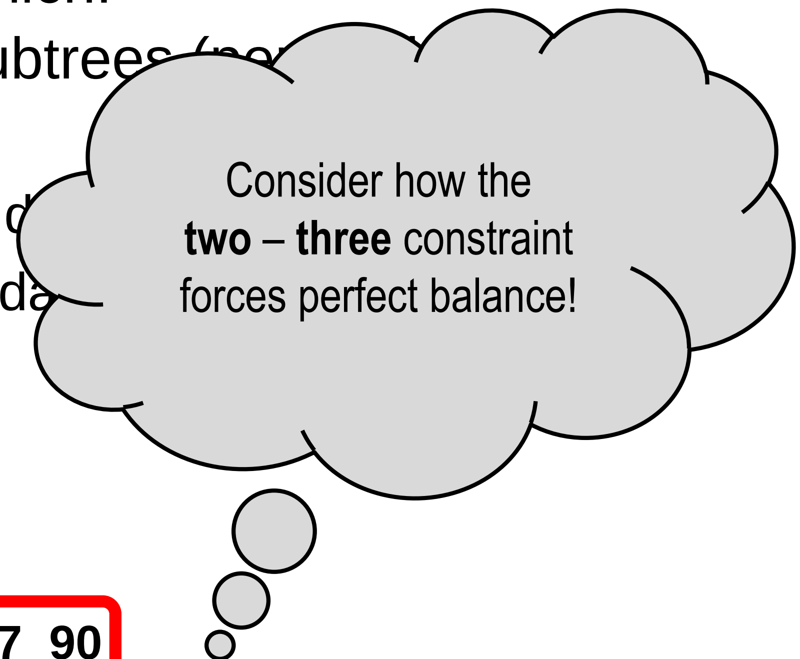
- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (perfect balance)
  - each node is either
    - a **2-node**, which contains one data item and 0 or 2 children
    - a **3-node**, which contains two data items and 0 or 3 children
  - the keys form a search tree
- Example:



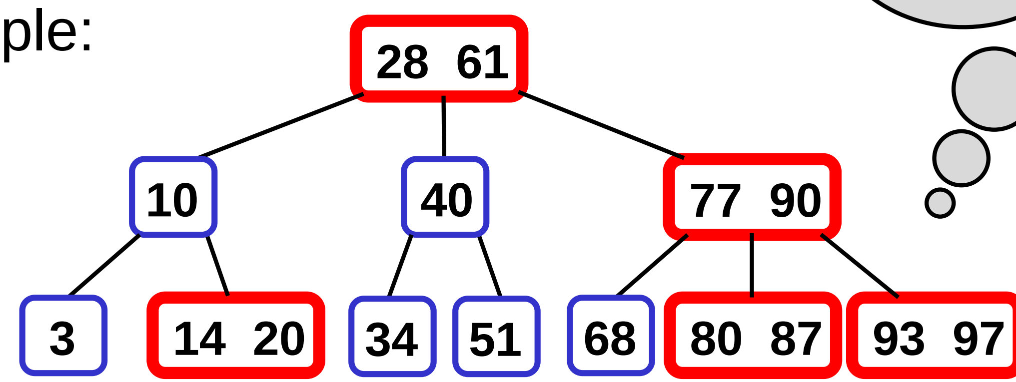


2-3 Trees

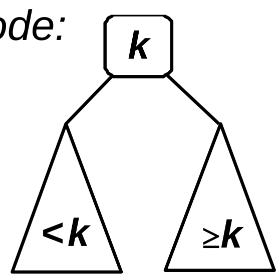
- A 2-3 tree is a balanced tree in which:
  - *all* nodes have equal-height subtrees (pe
  - each node is either
    - a **2-node**, which contains one d
    - a **3-node**, which contains two da
  - the keys form a search tree



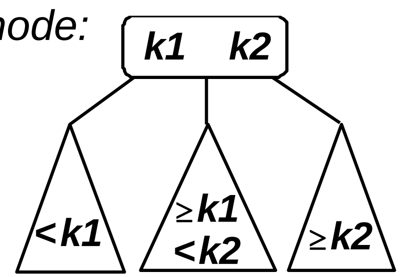
- Example:



2-node:

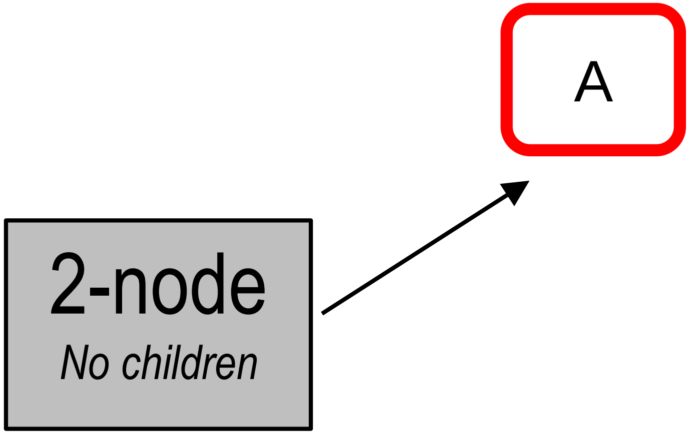


3-node:



2-3 Trees

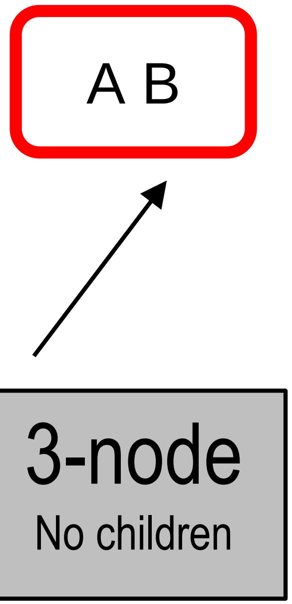
Add an A into our 2-3 tree:



2-3 Trees

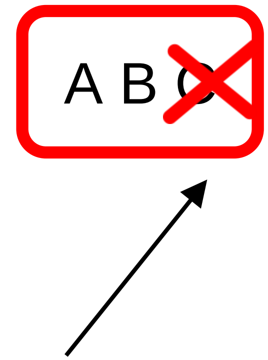
Add an A into our 2-3 tree:

Add a B into our 2-3 tree:



2-3 Trees

Add an A into our 2-3 tree:



Add a B into our 2-3 tree:

Can only have ***one*** or ***two*** data items!

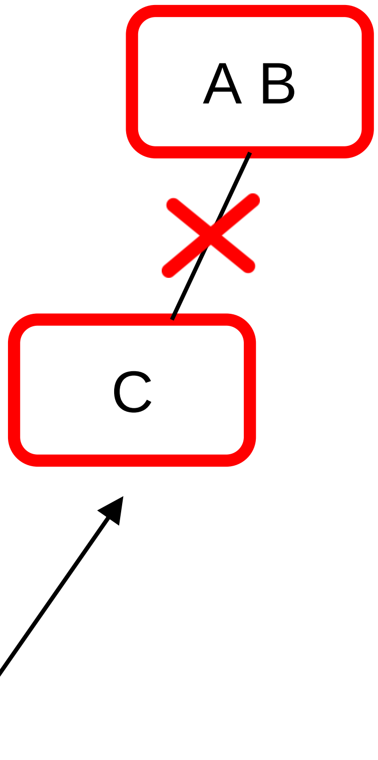
Add a C into our 2-3 tree?

2-3 Trees

Add an A into our 2-3 tree:

Add a B into our 2-3 tree:

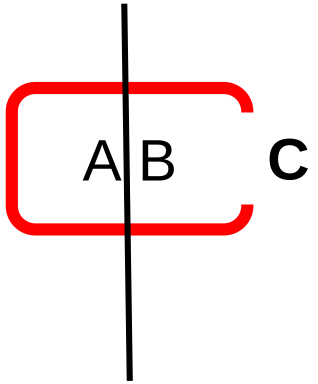
Add a C into our 2-3 tree?



A 3-node must have exactly
zero or ***three***
children!

2-3 Trees

Add an A into our 2-3 tree:



Add a B into our 2-3 tree:

Split the node!

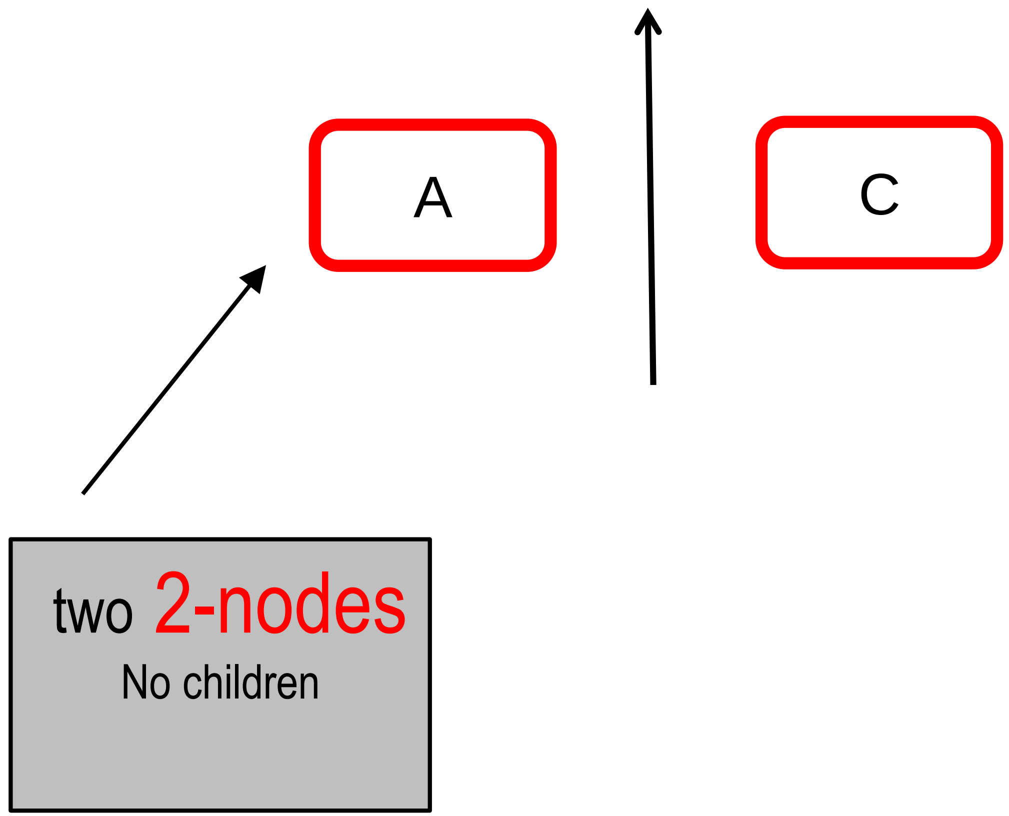
Add a C into our 2-3 tree?

2-3 Trees

Add an A into our 2-3 tree:

Add a B into our 2-3 tree:

Add a C into our 2-3 tree?

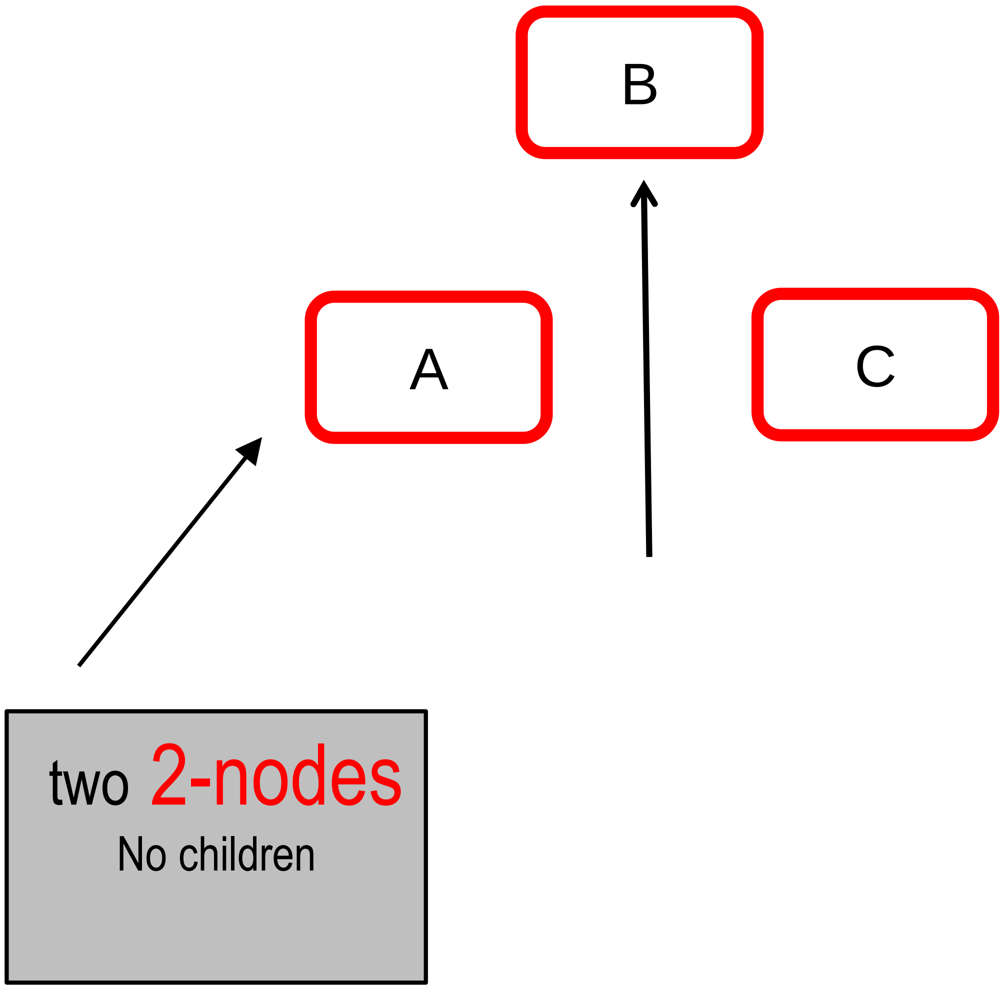


2-3 Trees

Add an A into our 2-3 tree:

Add a B into our 2-3 tree:

Add a C into our 2-3 tree?

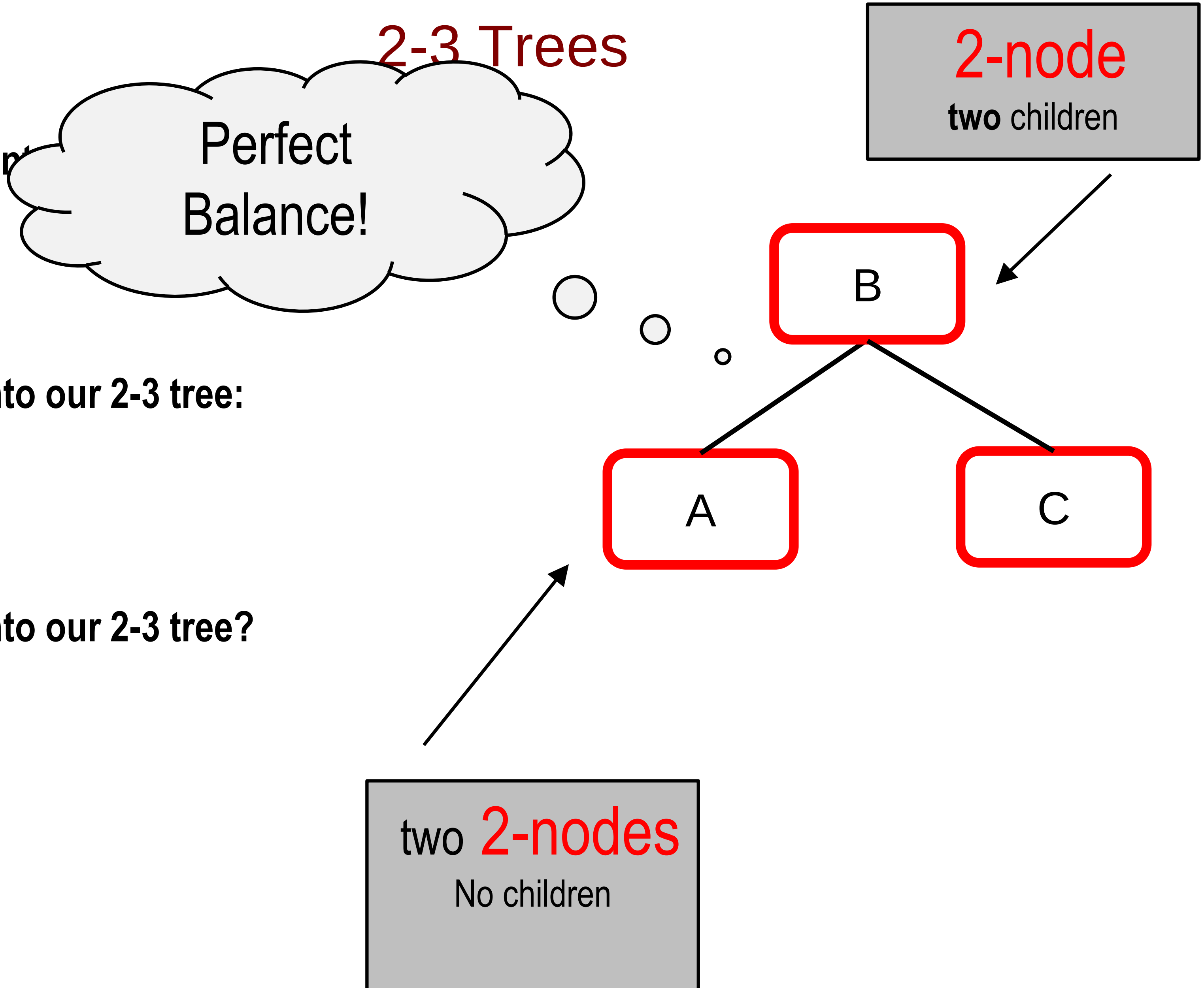


2-3 Trees

Add an A in

Add a B into our 2-3 tree:

Add a C into our 2-3 tree?



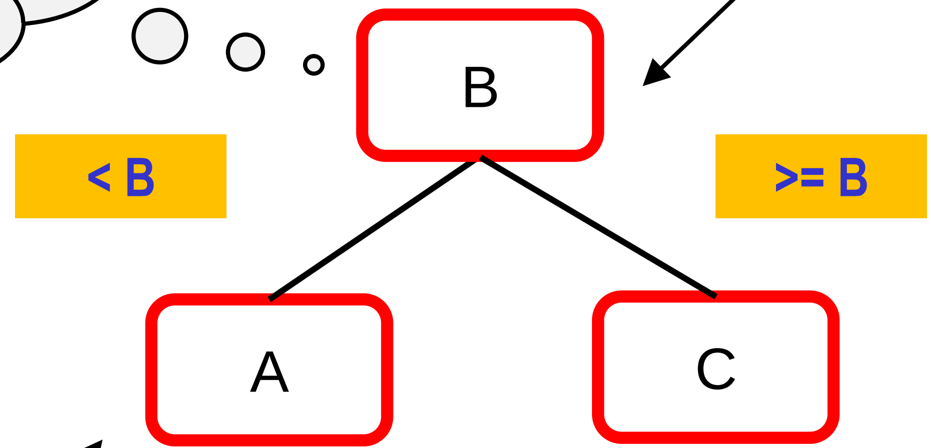
2-3 Trees

Add an A in

Need to maintain the property of the search tree!

Add a B into our 2-3 tree:

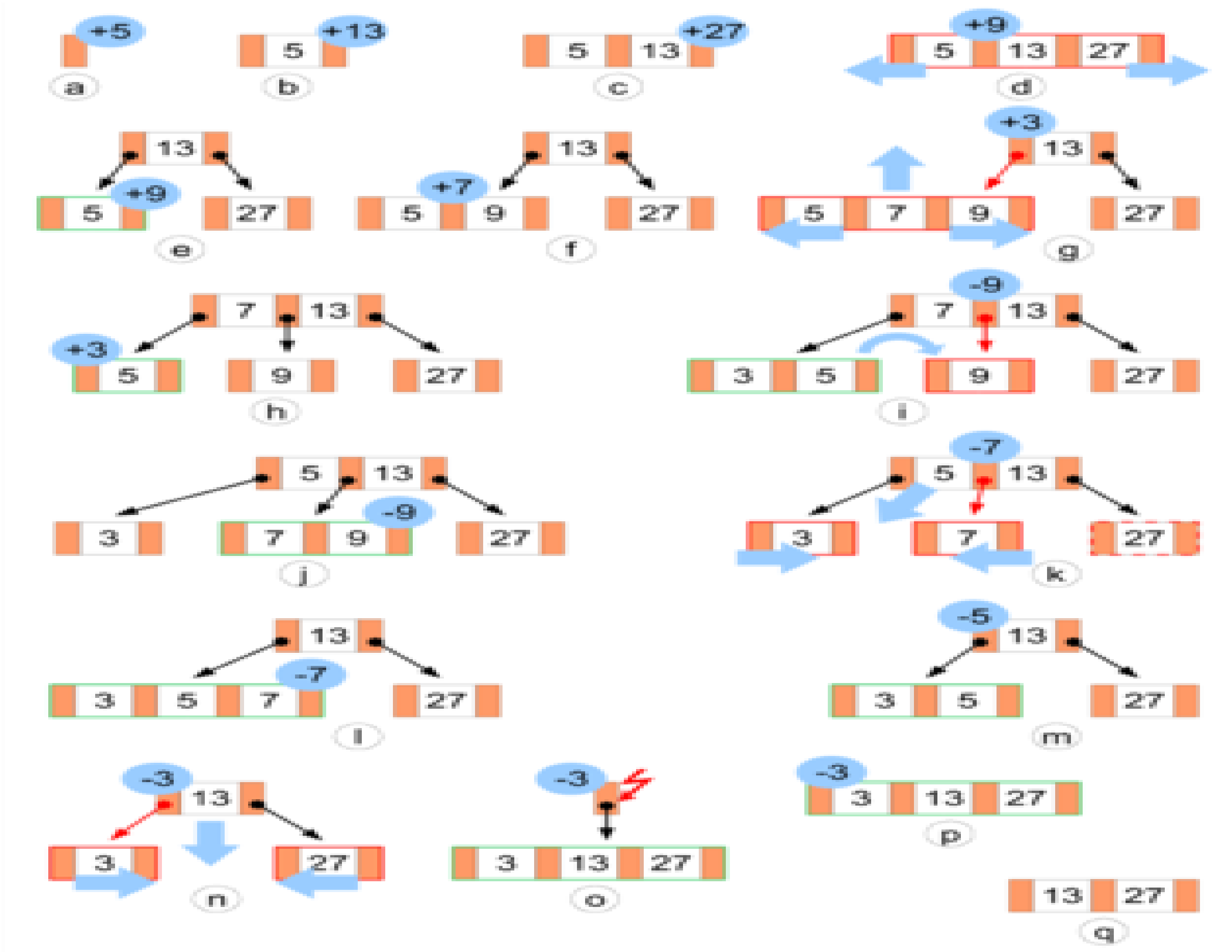
Add a C into our 2-3 tree?



2-node
two children

two 2-nodes
No children

Operations on 2-3 trees



Search in 2-3 Trees

- Algorithm for searching for an item with a key *k*:

  if *k* == one of the root node's keys, you're done

  else if *k* < the root node's first key

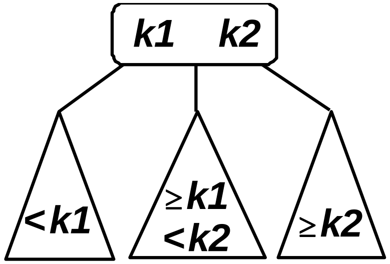
  search the left subtree

  else if the root is a 3-node and k < its second key

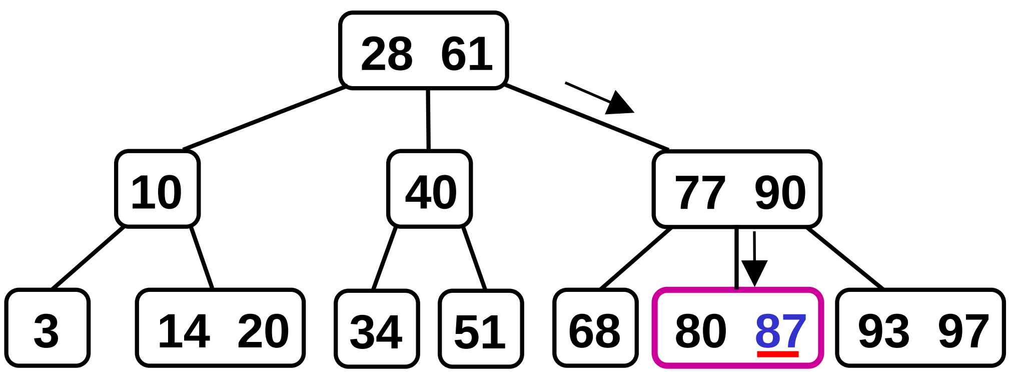
  search the middle subtree

  else

  search the right subtree



- Example: search for 87



Insertion in 2-3 Trees

- Algorithm for inserting an item with a key *k*:

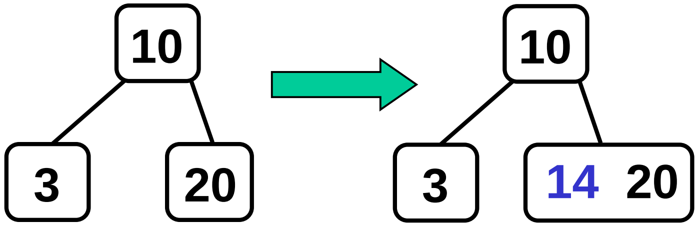
  search for *k,* but don't stop until you hit a leaf node

  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, making it a 3-node

  example : add 14



Insertion in 2-3 Trees

- Algorithm for inserting an item with a key *k*:

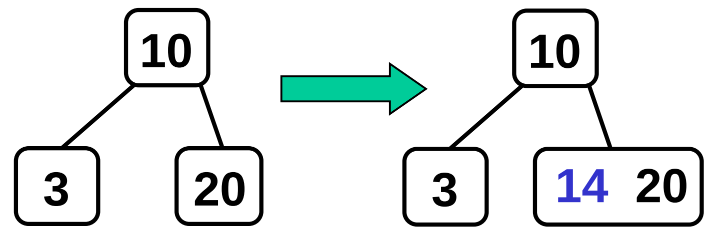
  search for *k,* but don't stop until you hit a leaf node

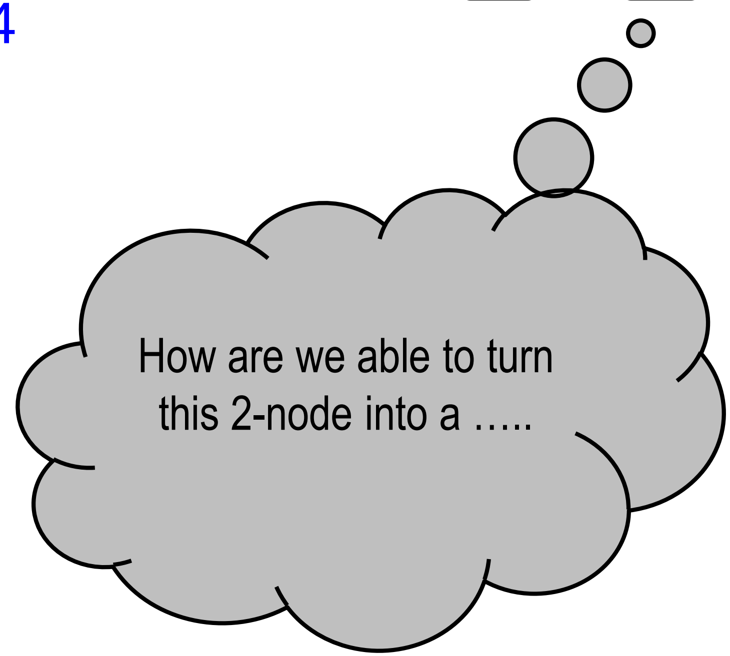
  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, making it a 3-node

  example : add 14





Insertion in 2-3 Trees

- Algorithm for inserting an item with a key *k*:

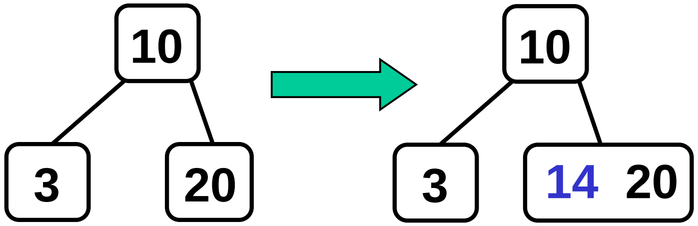
  search for *k,* but don't stop until you hit a leaf node

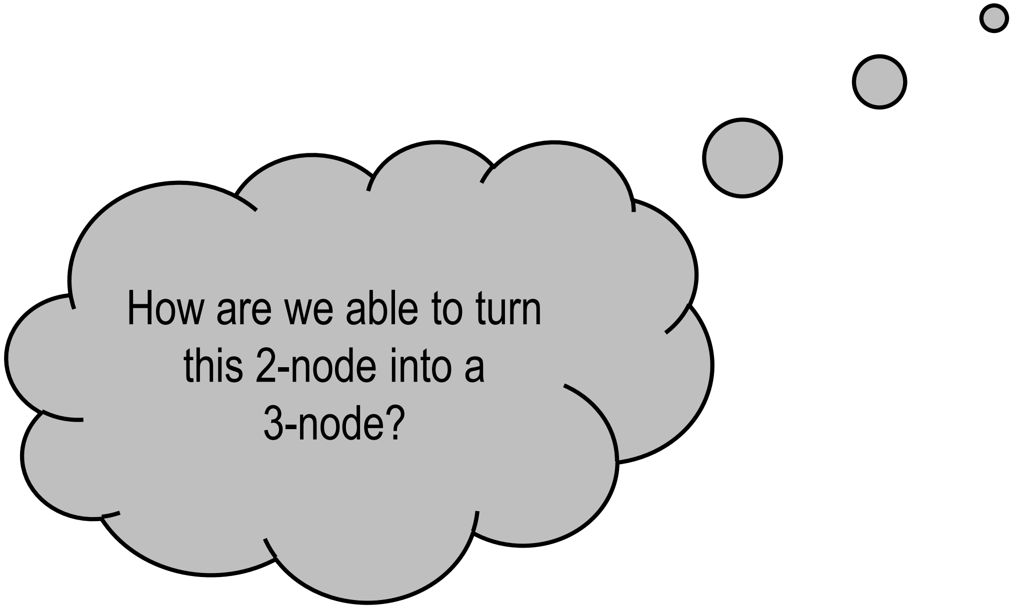
  let L be the leaf node at the end of the search

  if L is a 2-node

    add *k* to L, making it a 3-node

  example : add 14





Insertion in 2-3 Trees

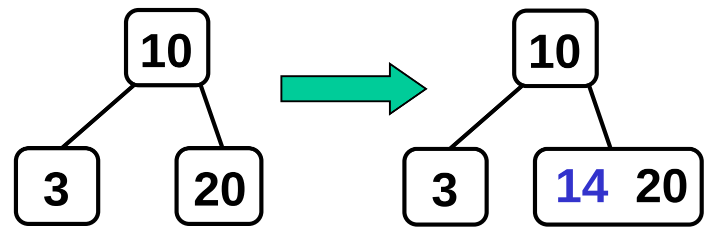
- Algorithm for inserting an item with a key *k*:

  search for *k,* but don’t stop until you hit a leaf node

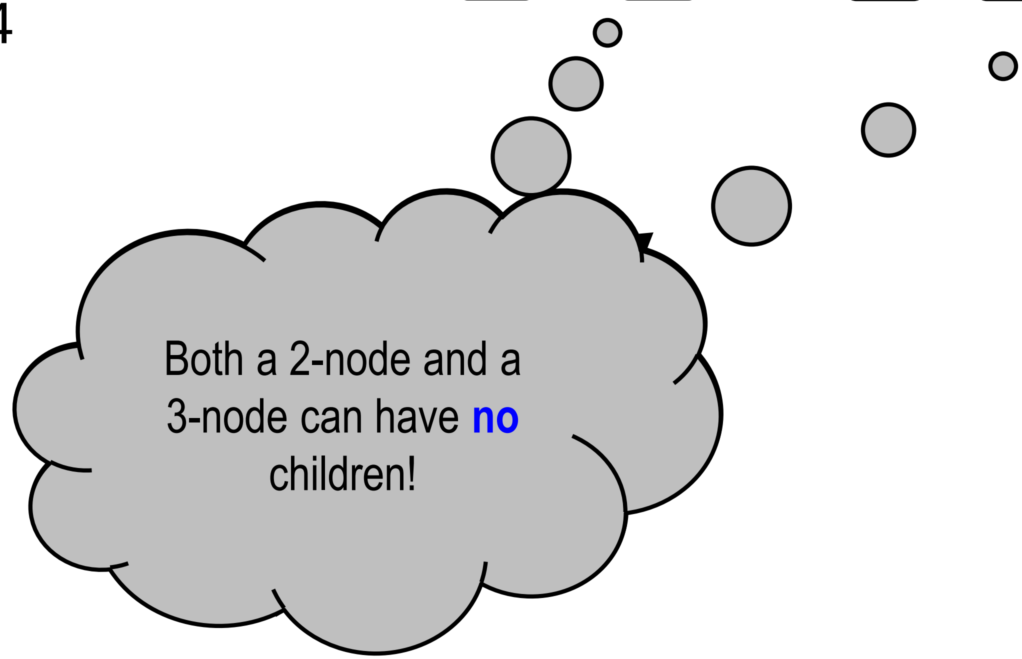
  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, making it a 3-node



example : add 14



Insertion in 2-3 Trees

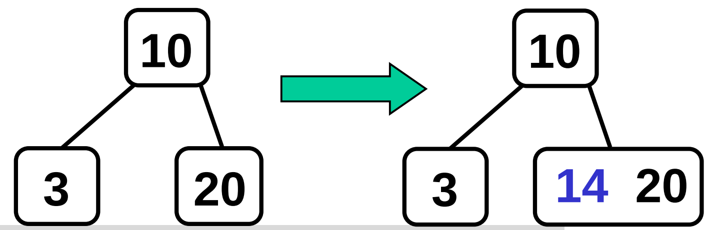
- Algorithm for inserting an item with a key *k*:

  search for *k,* but don't stop until you hit a leaf node

  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, making it a 3-node

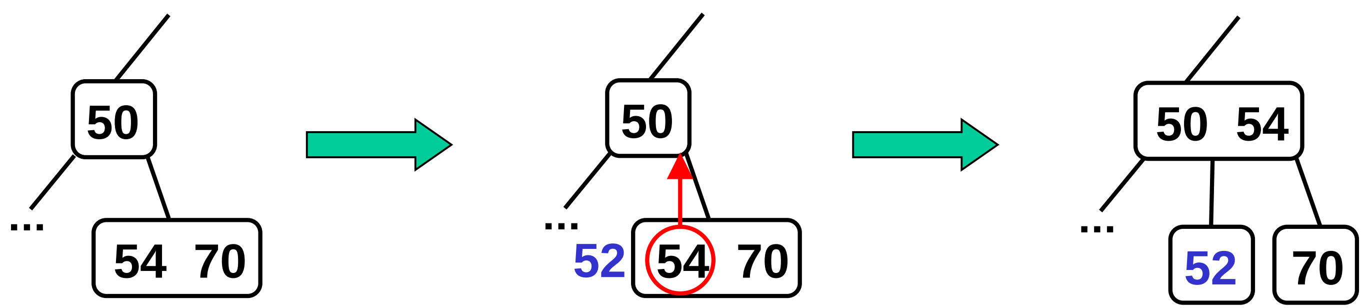


  else if L is a 3-node

  split L into two 2-nodes containing the items with the smallest and largest of: *k,* L's 1st key, L's 2nd key

  the middle item is "sent up" and inserted in L's parent

  example: add 52



Insertion in 2-3 Trees

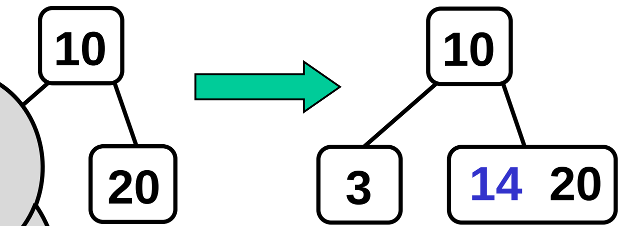
- Algorithm for inserting an item with a key *k*:

  search for *k,* but don't stop until you hit a leaf node

  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, maki



Why not just add the **52** to the node containing 50?

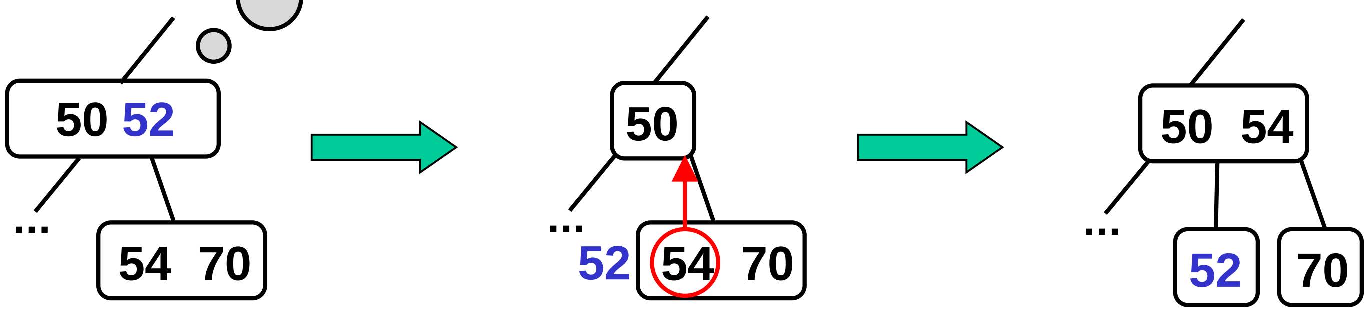
else if L is a 3-no

split L into tw ... tems with the

smallest an ... ey, L's 2nd key

the middle item ... serted in L's parent

example: add 52



Insertion in 2-3 Trees

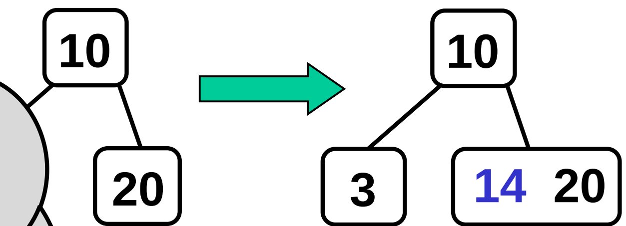
- Algorithm for inserting an item with a key *k*:

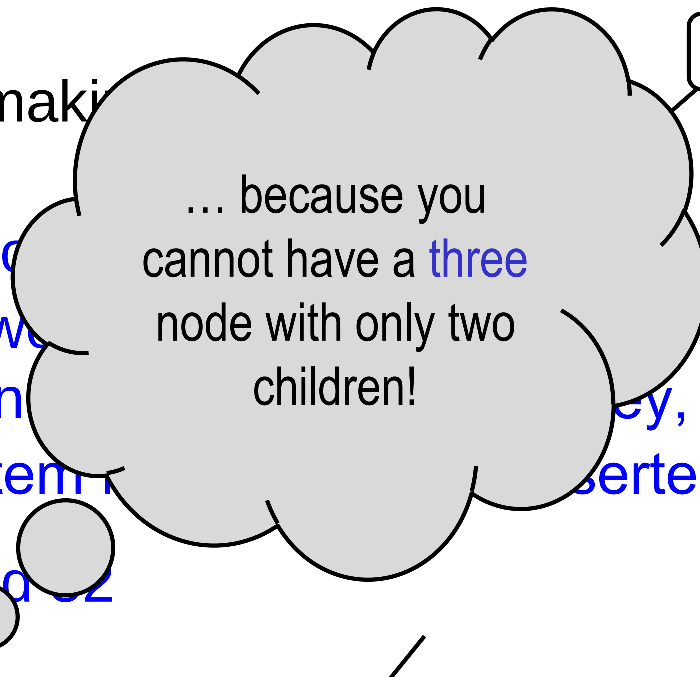
  search for *k*, but don't stop until you hit a leaf node

  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, maki





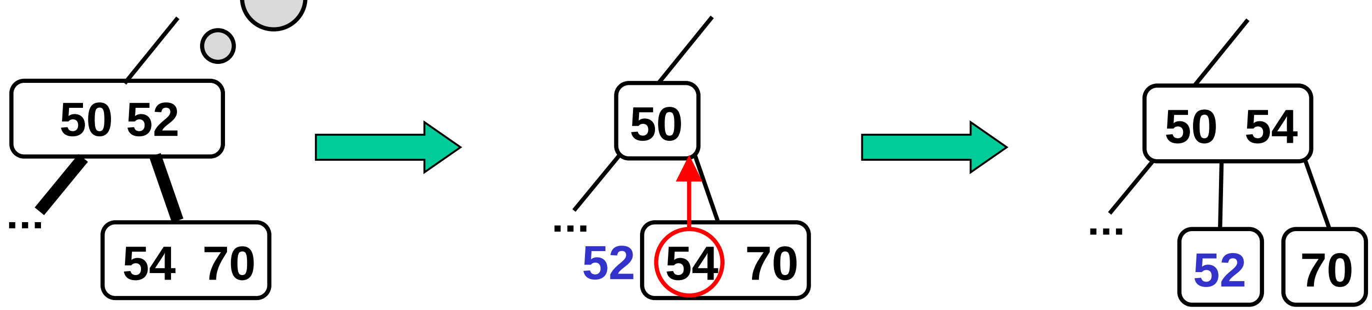
  else if L is a 3-no

  split L into tw … tems with the

  smallest an … ey, L's 2^{nd} key

  the middle item … serted in L's parent

  example: add 52



Insertion in 2-3 Trees

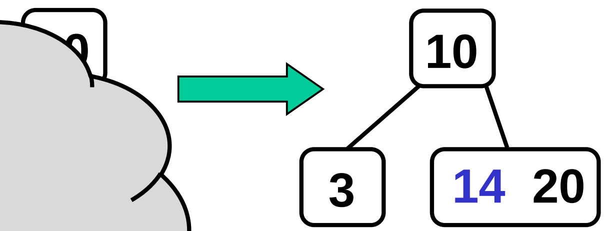
- Algorithm for inserting an item with a key *k*:

  search for *k,* but don't stop until you hit a leaf node

  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, maki



Note how the split and the move transitioned the node from a 2-node to a …

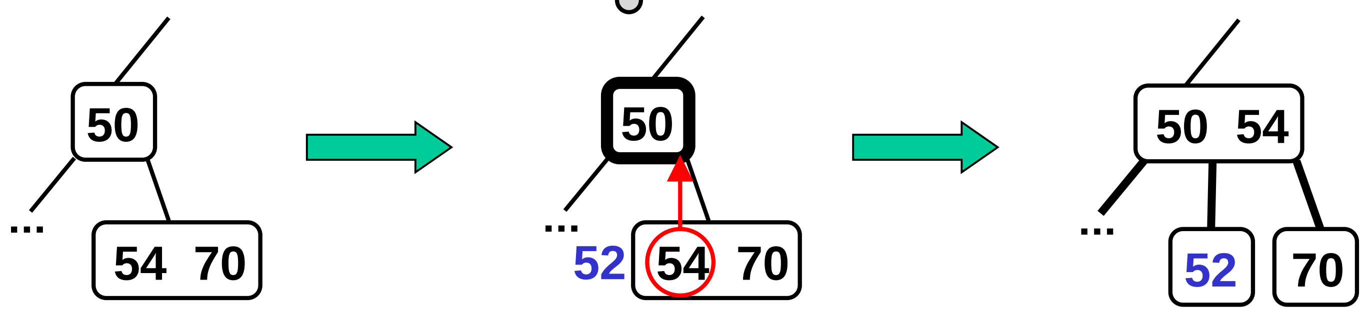
else if L is a 3-n

split L into tw ms with the

smallest and lar key, L's 2nd key

the middle item is "sent up" and inserted in L's parent

example: add 52



Insertion in 2-3 Trees

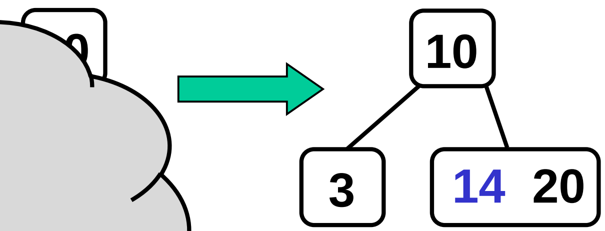
- Algorithm for inserting an item with a key *k*:

  search for *k*, but don't stop until you hit a leaf node

  let L be the leaf node at the end of the search

  if L is a 2-node

  add *k* to L, maki



Note how the split and the move transitioned the node from a 2-node to a 3-node!

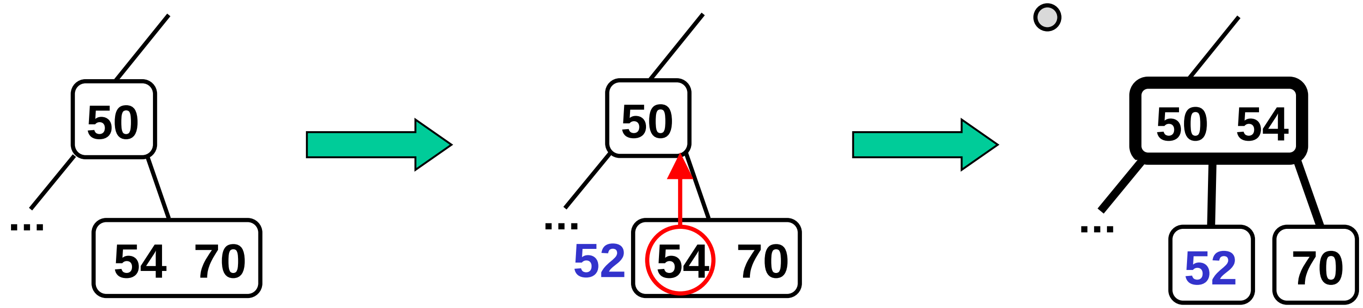
else if L is a 3-n

split L into tw ems with the

smallest and lar key, L's 2^{nd} key

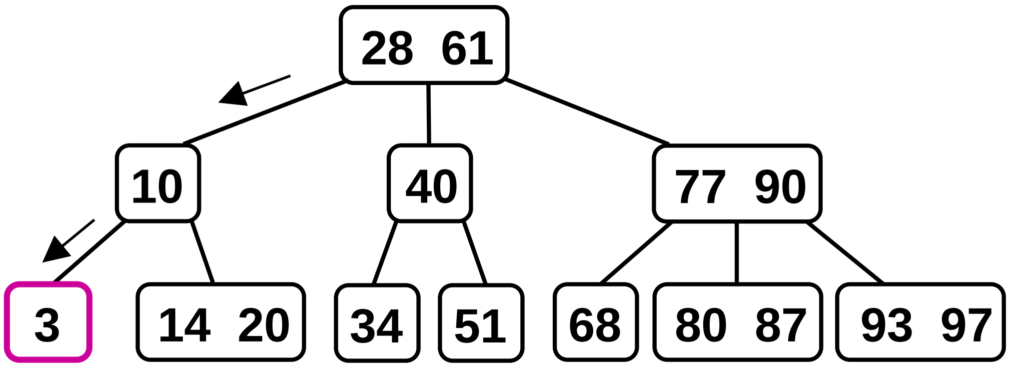
the middle item is "sent up" and inse ted in L's parent

example: add 52

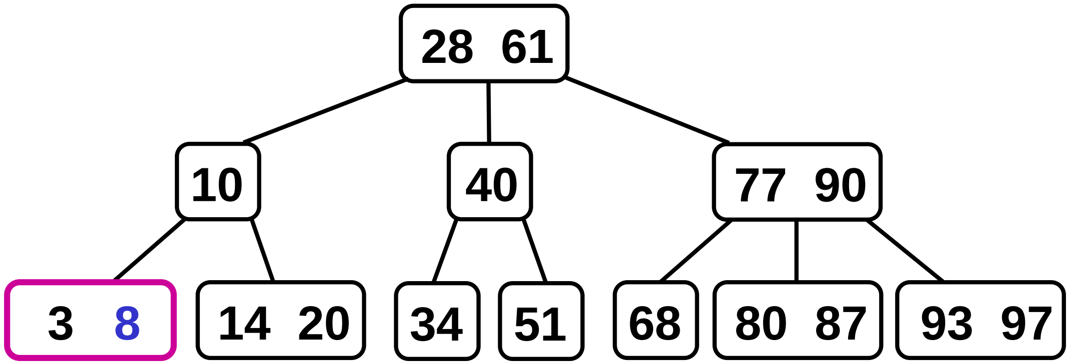


Example 1: Insert 8

- Search for 8:

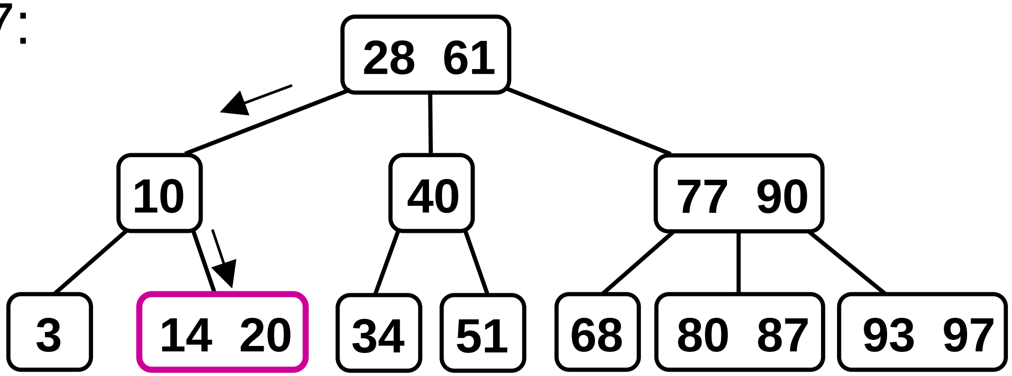


- Add 8 to the leaf node, *making it a 3-node*:

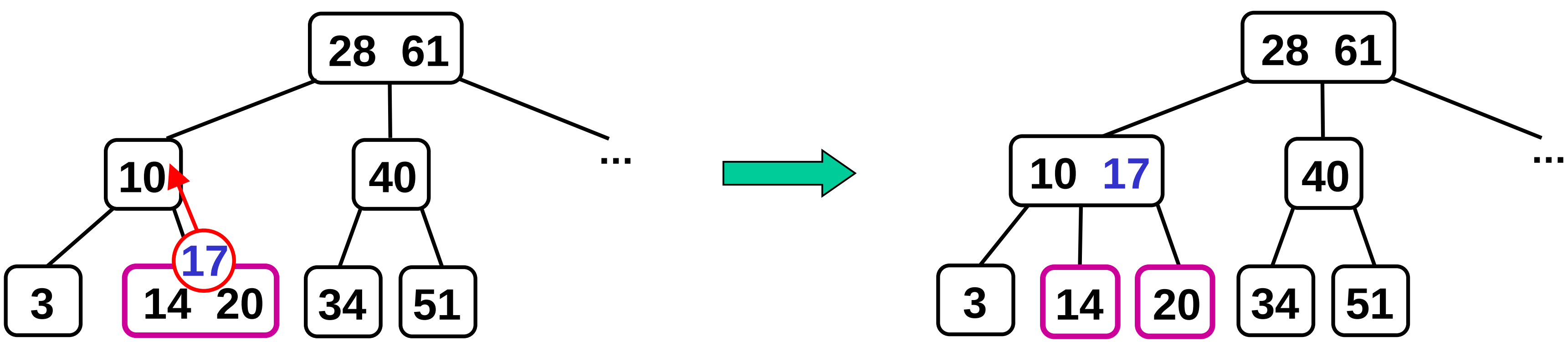


Example 2: Insert 17

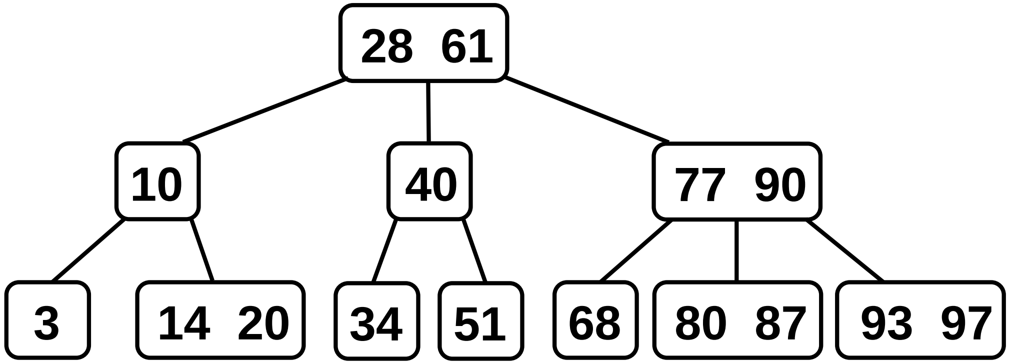
- Search for 17:



- Split the leaf node, and send up the middle of 14, 17, 20 and insert it the leaf node's parent:

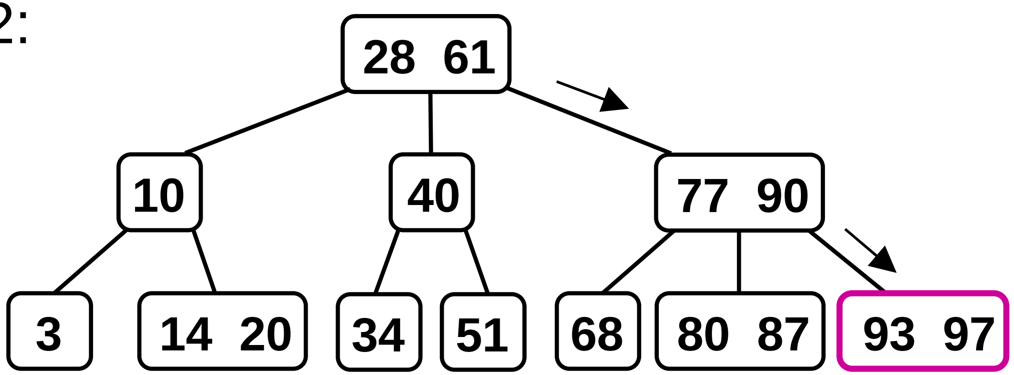


Example 3: Insert 92

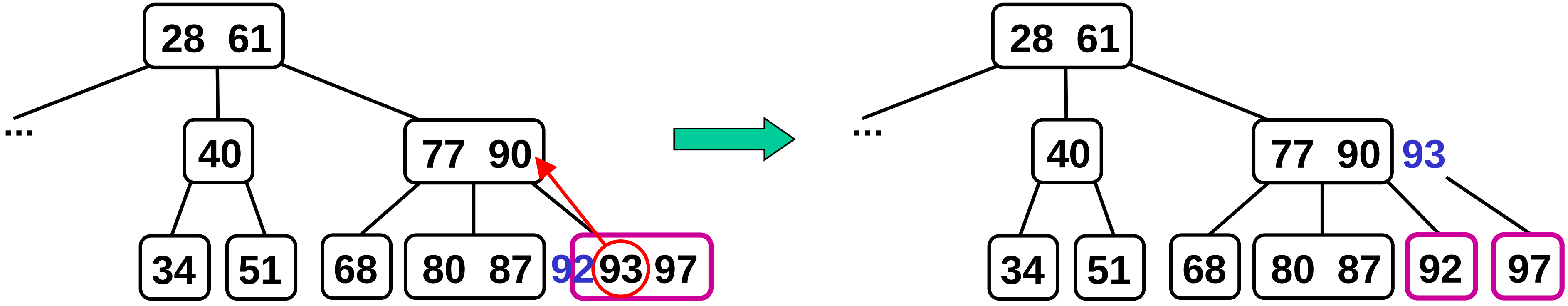


Example 3: Insert 92

- Search for 92:



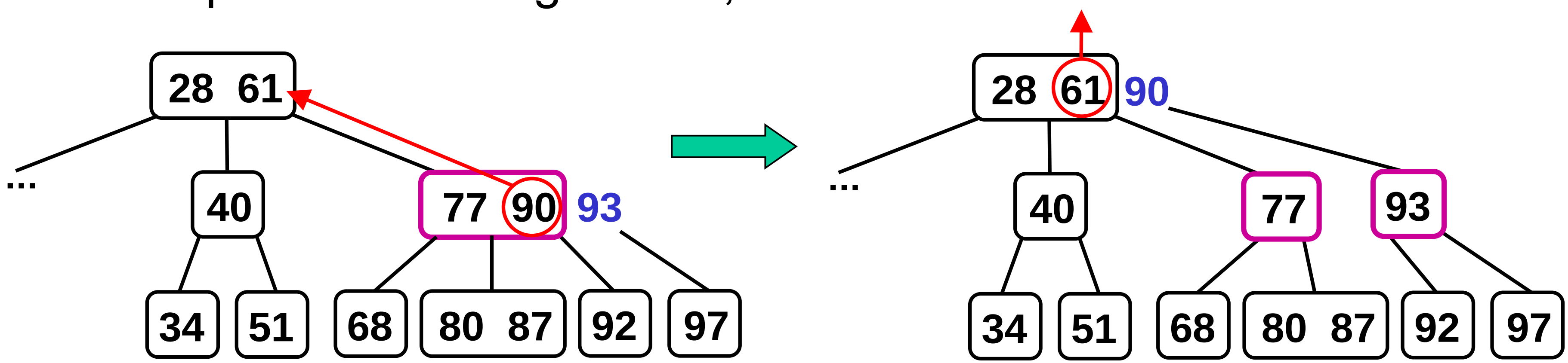
- Split the leaf node, and send up the middle of 92, 93, 97 and insert it the leaf node's parent:



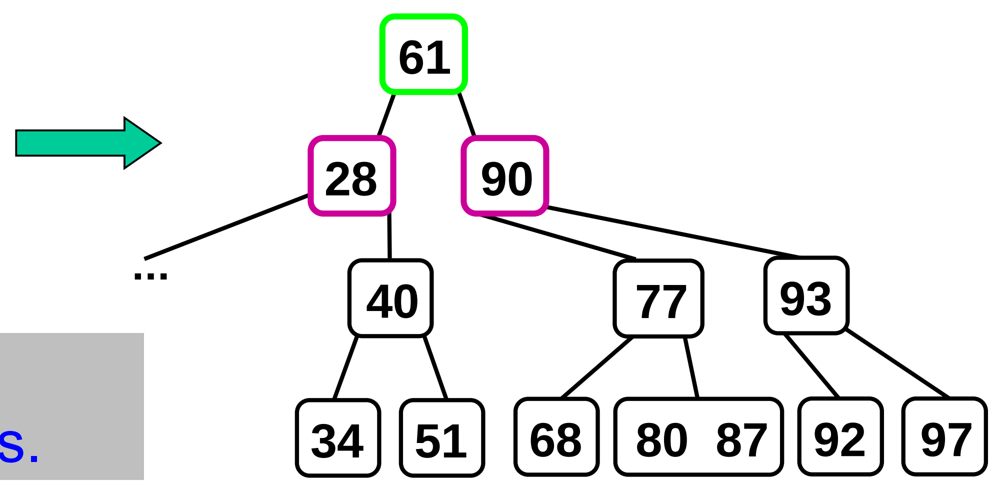
- In this case, the leaf node's parent is also a 3-node, so we need to split is as well…

Example 3 (cont.)

- We split the root's right child, but the root is also full!

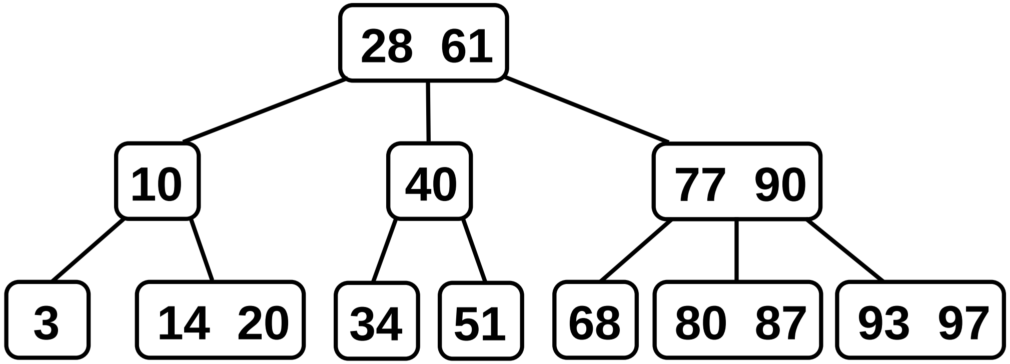


- Then we split the root, which increases the tree's height by 1, but the tree is still balanced.



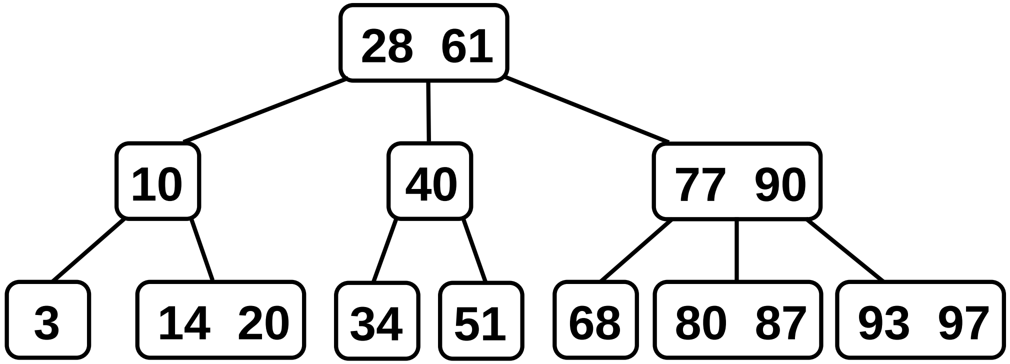
- This is only case in which the tree's height increases.

Efficiency of 2-3 Trees



- A 2-3 tree containing n items has a height <= $\log_2 n$.
- Thus, search and insertion are both $O(\log n)$.
  - a search visits at most $\log_2 n$ nodes
  - an insertion begins with a search; in the worst case, it goes all the way back up to the root performing splits, so it visits at most $2\log_2 n$ nodes

Efficiency of 2-3 Trees



- A 2-3 tree containing n items has a height <= $\log_2 n$.
- Thus, search and insertion are both $O(\log n)$.
  - a search visits at most $\log_2 n$ nodes
  - an insertion begins with a search; in the worst case, it goes all the way back up to the root performing splits, so it visits at most $2\log_2 n$ nodes
- Deletion is tricky – you may need to coalesce nodes! However, it also has a time complexity of $O(\log n)$.
- Thus, we can use 2-3 trees for a $O(\log n)$-time data dictionary!