## 2-3 Trees



In a 2-3 tree:

- each node contains either one or two key values
- every internal node either holds one key value and has two children, or holds two key values and has three children
- every leaf is at the same level in the tree
- there is a BST-like arrangement of values for efficient searching

Compared to a BST:

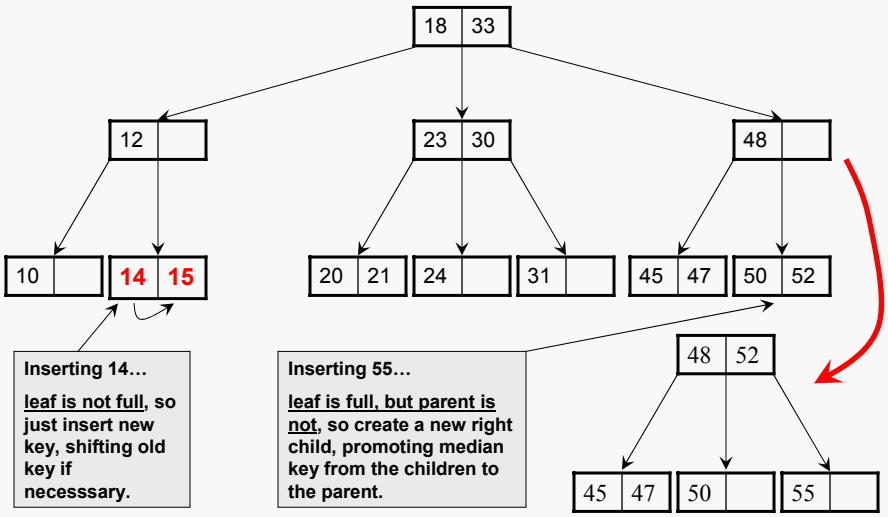
- 2-3 trees have lower update cost on average
- a 2-3 tree storing N keys will be shallower than the corresponding BST, but…
- … search in a 2-3 tree has essentially the same cost as in a BST




## 2-3 Tree Insertion



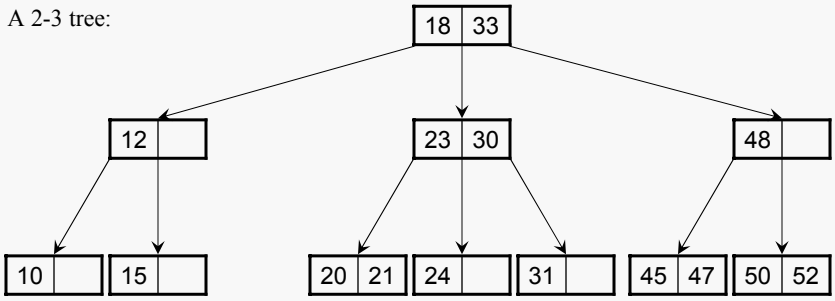
Insertion into a 2-3 tree is similar to a BST. First find the appropriate leaf…



## 2-3 Tree Properties



A 2-3 tree:




Each node can store up to two key values and up to three pointers.

The left child holds values less than the first key.

The middle child holds values greater than the first key and less than the second key.

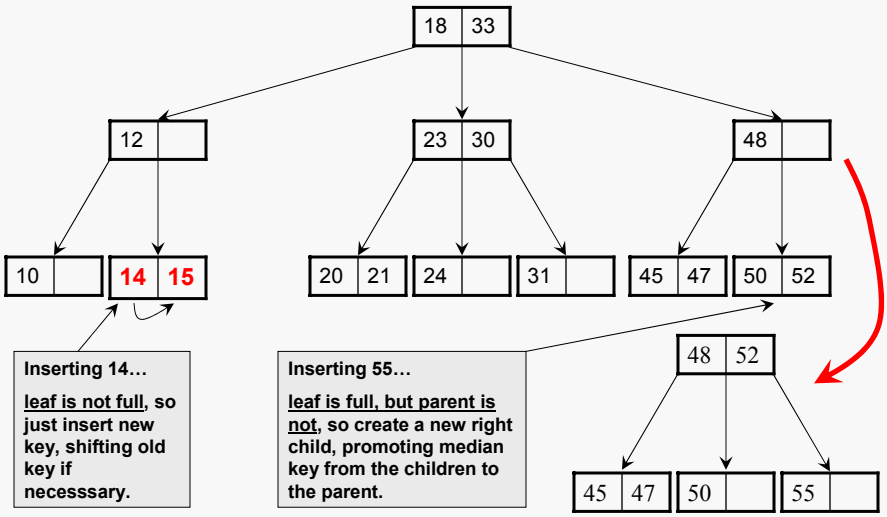
The right child holds values greater than the second key.




## 2-3 Tree Insertion



Insertion into a 2-3 tree is similar to a BST. First find the appropriate leaf…




**Inserting 14…**

**leaf is not full, so just insert new key, shifting old key if necesssary.**

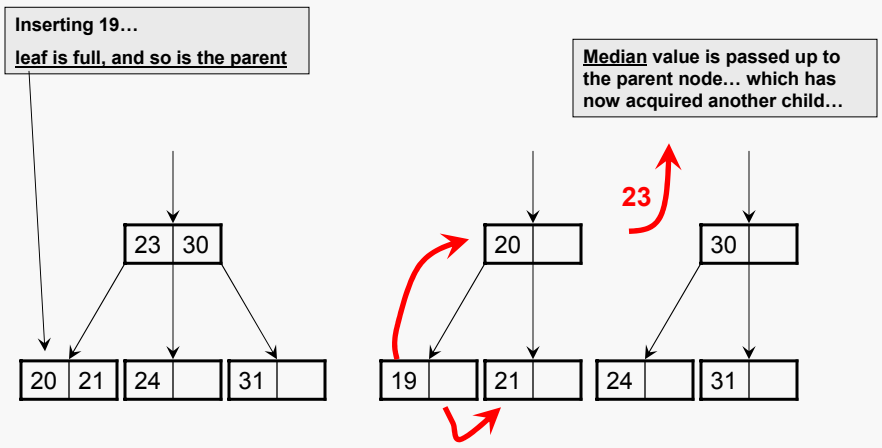
**Inserting 55…**

**leaf is full, but parent is not, so create a new right child, promoting median key from the children to the parent.**

## 2-3 Tree Splitting



If a subtree is sufficiently full, insertion may cause the parent to split:



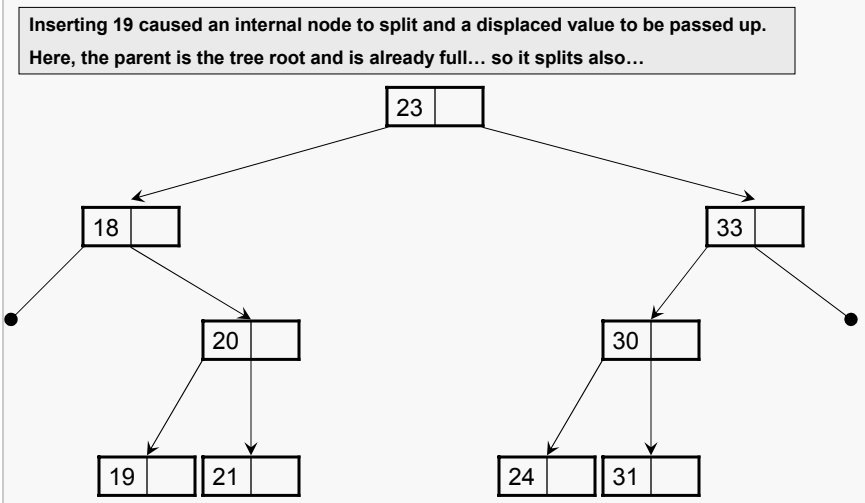




## 2-3 Tree Splitting the Root



In the worst case, the tree root splits, increasing the height of the tree:



## 2-3 Tree Deletion



Deletion from a 2-3 tree is, of course, essentially just the inverse of insertion. As with the BST, there are cases:

Deletion of a value that lies in an internal node:

- replace the value with its closest successor (which must lie in a leaf)
- delete the successor from the leaf

Deletion of a value that lies in a leaf:

- if the leaf is still non-empty, nothing else needs to be done
- if the leaf is now empty, the parent has an empty child, which is not allowed, and merely deleting the leaf would leave the parent with too many values
  - if the left or right sibling has two values, we can borrow and rearrange values
  - if not, we merge the leaf and its sibling (if any), demoting a value from the parent

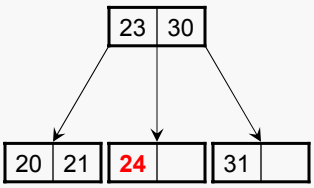
Details are interesting…some are addressed on the following slides.



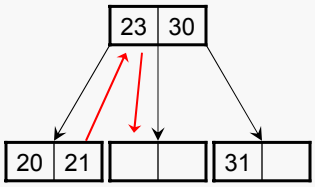

## Deletion from a Leaf I



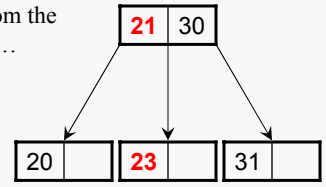
Consider deletion of the value 24:




This would leave the middle leaf empty, which is not allowed. However, the left sibling contains an "extra" value. We may manage the deletion by "demoting and borrowing":

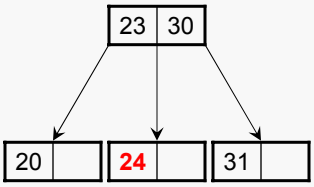



We move the "separator" value between the siblings to the leaf, and move the appropriate value from the sibling to the parent…

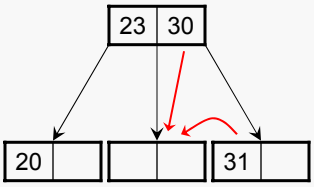


## Deletion from a Leaf II

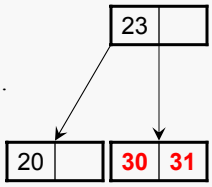
Consider deletion of the value 24:




This would also leave the middle leaf empty, but neither sibling has any values to spare. So, again we demote and borrow, but we also delete an empty leaf.




We pick a sibling, move the *separating value* between the empty leaf and the sibling, and the sibling's value both into the empty leaf, and delete the sibling...



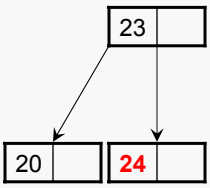




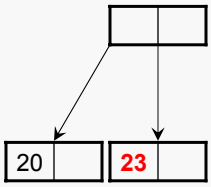


## Deletion from a Leaf III

Consider deletion of the value 24:




This would also leave the middle leaf empty, but neither sibling has any values to spare, nor can we delete either child without leaving the parent with a single child, which is unacceptable… we can still demote a value from the parent…




But, now the parent has too few values (underflow), and we must handle that recursively by considering the siblings and parent of that node…