

Balanced Trees - 2-3 Tree

2-3 Trees

1

Balanced Trees

- ◆ Binary search trees are not guaranteed to be balanced given random inserts and deletes
 - Tree could degrade to $O(n)$ operations
- ◆ Balanced search trees
 - Operations maintain a balanced tree

2-3 Trees

2

2-3 Tree

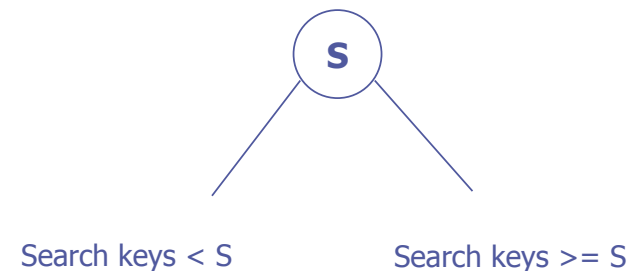
- ◆ Guaranteed to always be balanced
 - $O(\lg n)$ operations
- ◆ Each interior node has two or three children
 - Nodes with 2 children are called 2 nodes
 - Nodes with 3 children are called 3 nodes
 - NOT A BINARY TREE
- ◆ Data is stored in both internal nodes and leaves

2-3 Trees

3

2 Node

- ◆ 2 nodes have one data item and 2 children

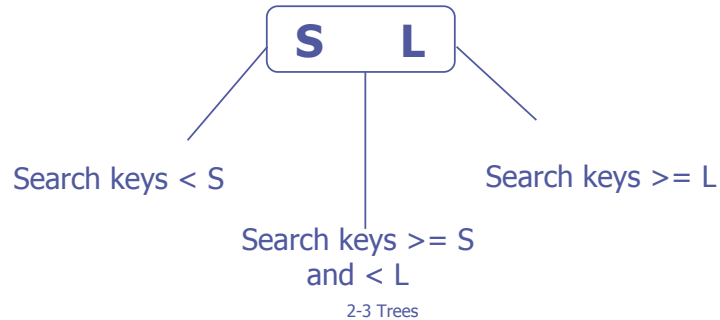


2-3 Trees

4

3 Node

- ◆ 3 nodes have two data items and 3 children (a left child, a middle child, and a right child)



2-3 Tree

- ◆ A leaf may contain 1 or 2 data items
- ◆ 2-3 trees are good because they are easy to maintain as balanced
 - Operations take care of that for you

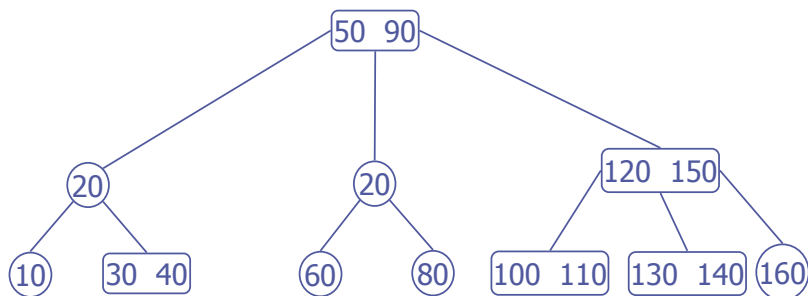
Node class

```
itemtype smallItem, largeItem
Node *left, *middle, *right, *parent
// you may find that you need other data members
// to make your life easier
// do not add a third item or a fourth child
```

2-3 Trees

6

2-3 Tree



Traversing a 2-3 Tree

- ◆ Inorder traversal -

```
inorder (node* cur)
  if current
    inorder(cur->left)
    visit small item if it exists
    inorder(cur->middle)
    visit large item if it exists
    inorder(cur->right)
```

2-3 Trees

8

Searching a 2-3 Tree

```

// Assumes small and large exist. You will need to modify
// to account for nodes with only one value
search (Node* cur, itemtype key)
    if (cur)
        if (key is in cur)
            return cur
        else
            if (key < cur->small)
                search down left child
            else if (key > cur->large)
                search down right child
            else
                search down middle child
    
```

2-3 Trees

9

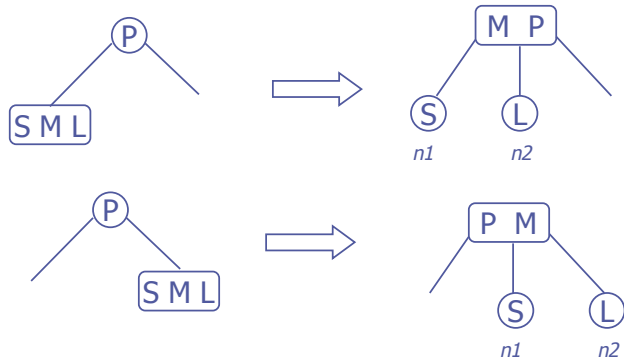
Insert

- ◆ To insert an item, find a leaf to put the item in then split nodes if necessary

2-3 Trees

10

Splitting a Leaf

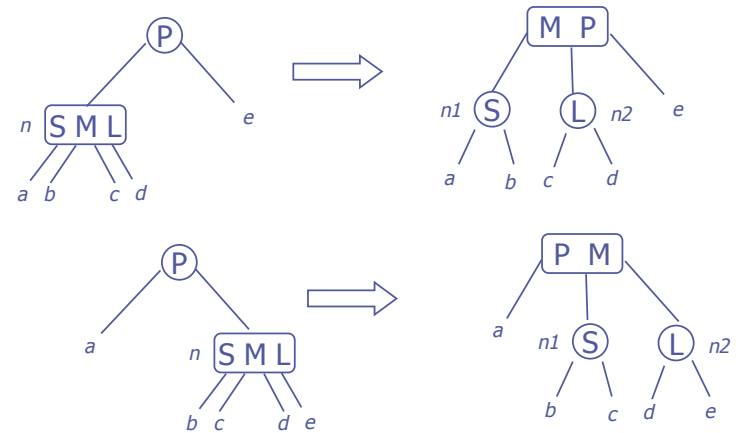


If splitting node causes the parent to have 3 items and 4 children, you will then an internal node...

2-3 Trees

11

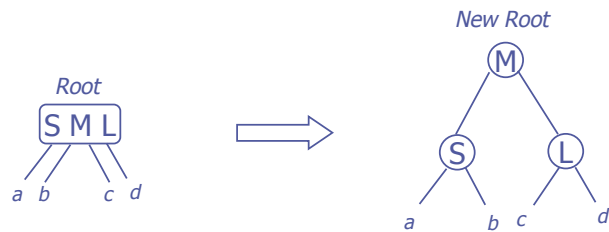
Splitting an Internal Node



2-3 Trees

12

Splitting the Root

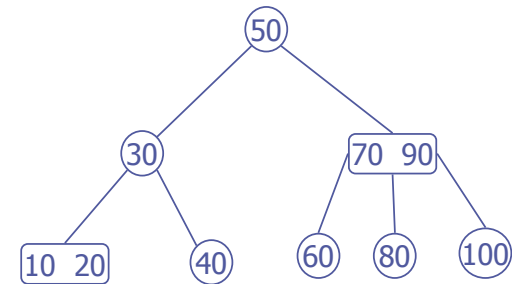


2-3 Trees

13

2-3 Tree

Start with this tree

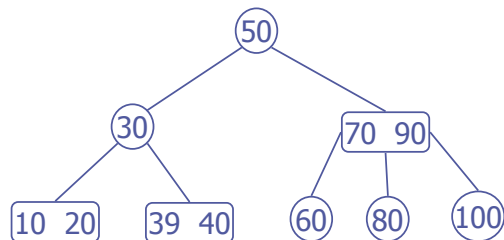


2-3 Trees

14

Insert 39

◆ Locate leaf to insert 39



◆ Leaf to insert only has 1 data item

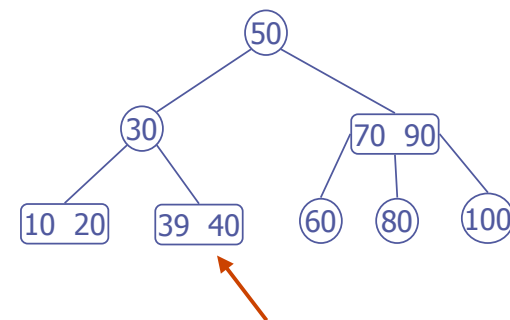
- Add 39 to the leaf

2-3 Trees

15

Insert 38

◆ Locate leaf to insert 38

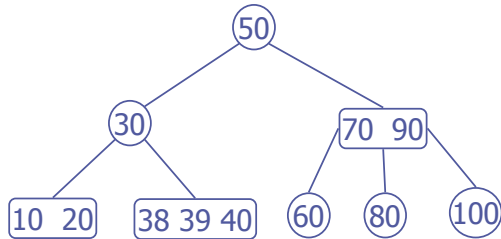


2-3 Trees

16

Insert 38

- ◆ Conceptualize inserting 38 into this leaf
 - Do not actually add the item because the node can only hold 2 data items



2-3 Trees

17

Insert 38

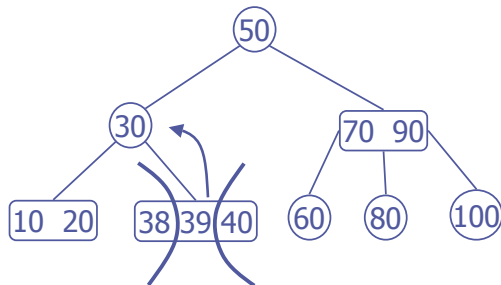
- ◆ Determine
 - Smallest = 38
 - Middle = 39
 - Largest = 40

2-3 Trees

18

Insert 38

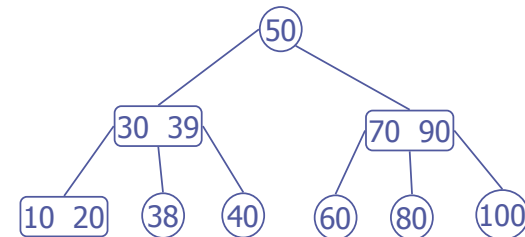
- ◆ Move middle value up to parent p
- ◆ Separate small and large values into two separate nodes that will be children of p



2-3 Trees

19

Insert 38

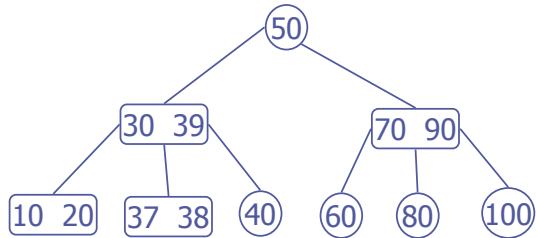


2-3 Trees

20

Insert 37

- ◆ Locate leaf to insert 37
- ◆ Leaf contains 1 data value, just insert value

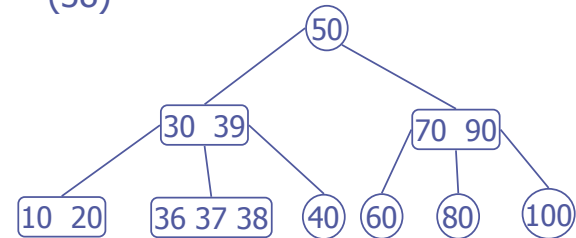


2-3 Trees

21

Insert 36

- ◆ Locate leaf to insert 36
- ◆ Conceptualize inserting 36 into this leaf
 - Determine small (36), middle (37), and large (38)

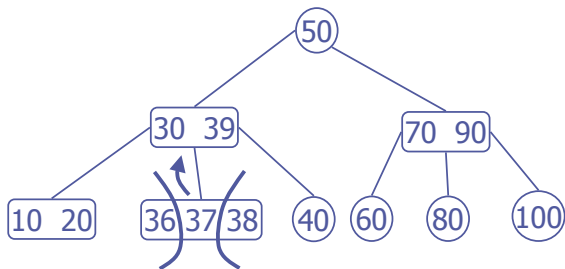


2-3 Trees

22

Insert 36

- ◆ Conceptualize moving middle value up to parent p
 - Do not actually move, node can't have 3 data values

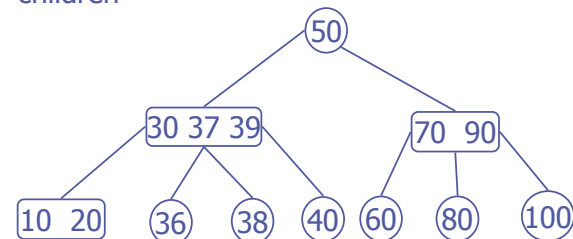


2-3 Trees

23

Insert 36

- ◆ Conceptualize attaching as children to p the smallest and largest values
 - Do not actually attach because a node can't have 4 children

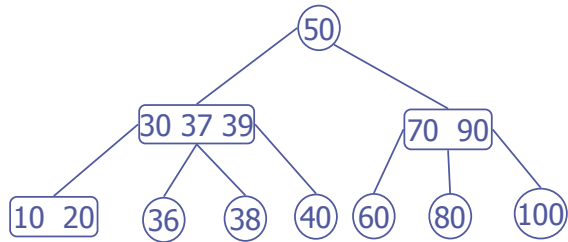


2-3 Trees

24

Insert 36

- ◆ Parent p now has 3 data values and 4 children
- ◆ Split - similar to leaf situation where leaf has 3 data values
 - You can generalize both situations into one

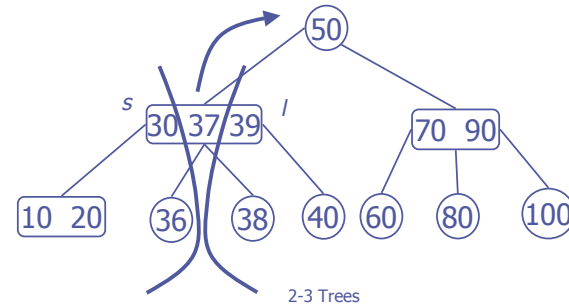


2-3 Trees

25

Insert 36

- ◆ Split parent p
 - Divide to small (30), middle (37), and large (39)
 - Move middle value to nodes parent
 - Small and large become new children, s and l

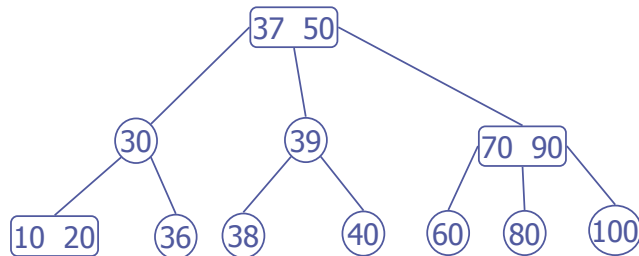


2-3 Trees

26

Insert 36

- ◆ Divide 4 children
 - Two leftmost become children of s
 - Two rightmost become children of l

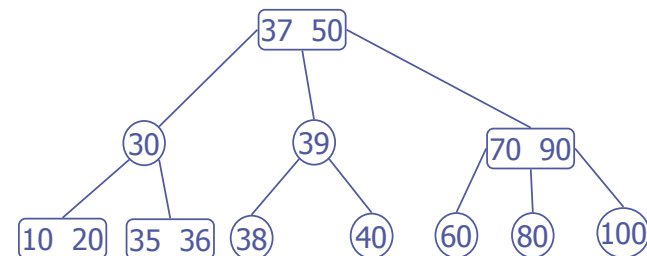


2-3 Trees

27

Insert 35

- ◆ Insert 35
 - Inserts into leaf

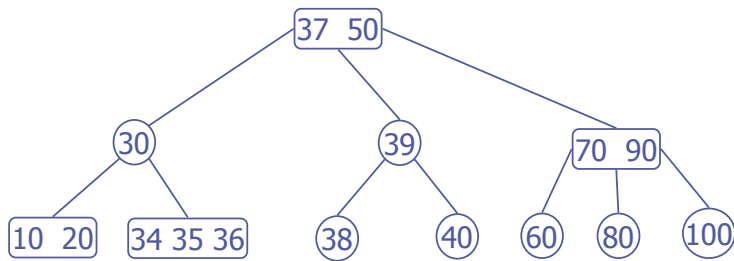


2-3 Trees

28

Insert 34

- ◆ Insert 34
 - Causes a split

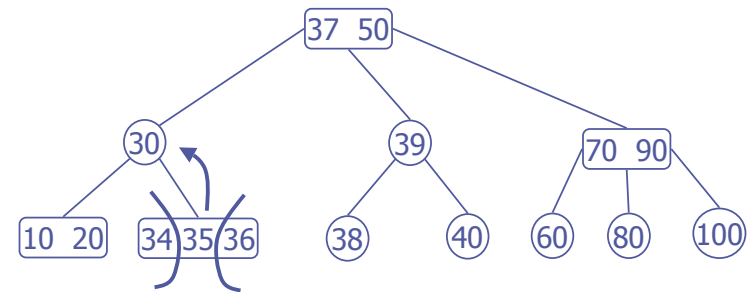


2-3 Trees

29

Insert 34

- ◆ Insert 34
 - Causes a split

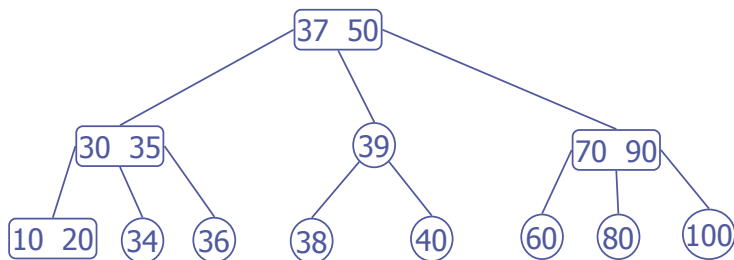


2-3 Trees

30

Insert 34

- ◆ Insert 34
 - Causes a split

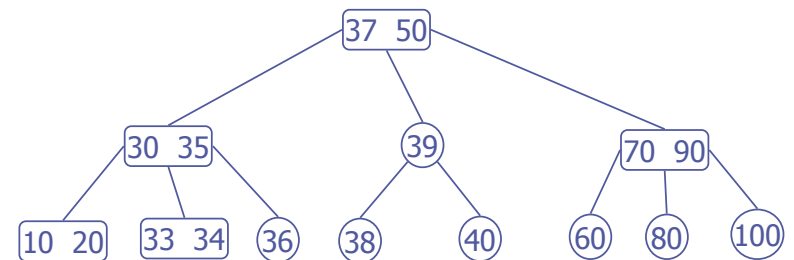


2-3 Trees

31

Insert 33

- ◆ Insert 33
 - Inserts into leaf



2-3 Trees

32

Insert

◆ Insert into a tree without duplicates

```
insert (itemtype item)
  leaf = leaf node to insert item (may be null or have 1 or 2 data items)

  if (leaf is null - only happens when root is null)
    add new root to tree with item
  else if (# data items in leaf = 1)
    add item to node
  else // leaf has 2 data items
    split ( leaf, item )
```

Insert (continued)

```
// Item is to be inserted into splitNode. The insertion
// of item will cause splitNode to have 3 items so splitNode
// must be split
split ( Node* splitNode, itemtype item, ... // you may need more )
  if ( splitNode is the root )
    create a new node p
  else
    let p be the parent of n

  Replace node n with 2 nodes, n1 and n2, so that p is their parent

  Give n1 the item in n with the smallest value
  Give n2 the item in n with the largest value

  // continued on next slide...
```

Insert (continued)

```
if (n is not a leaf)
  n1 becomes the parent of n's two leftmost children
  n2 becomes the parent of n's two rightmost children

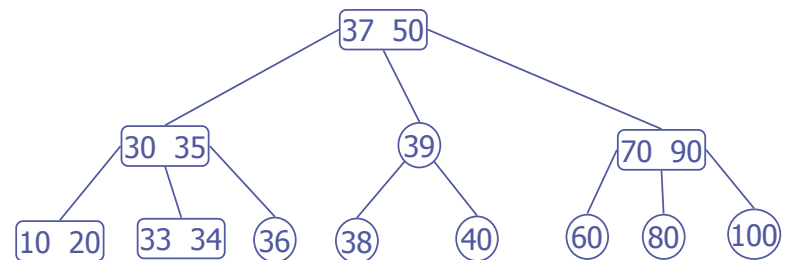
x = the item in n that has the middle key value

if ( adding x to p would cause p to have 3 items )
  split ( p, x )
else
  add x to p
```

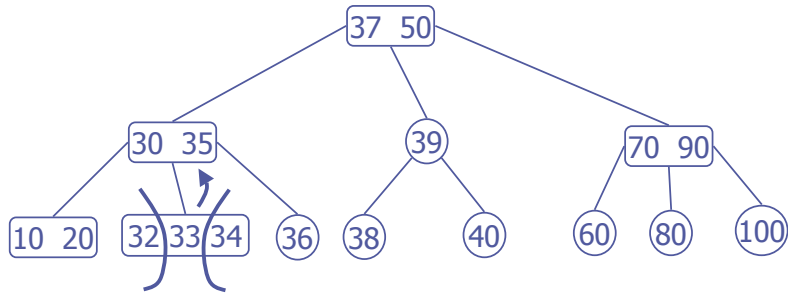
Insert 32

◆ In class exercise

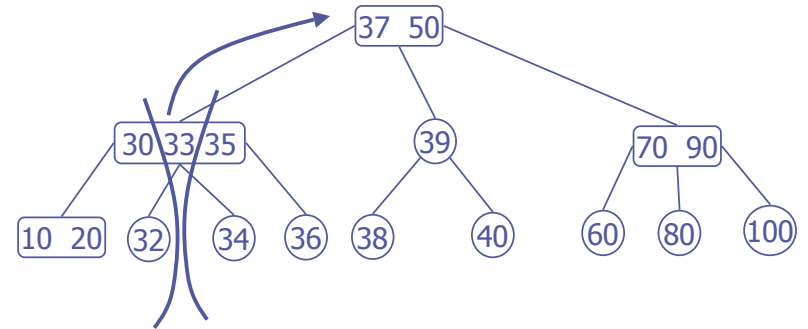
- Insert 32 into the tree below



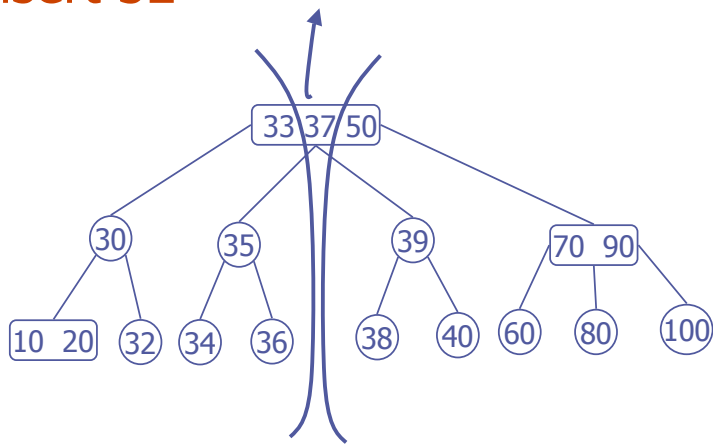
Insert 32



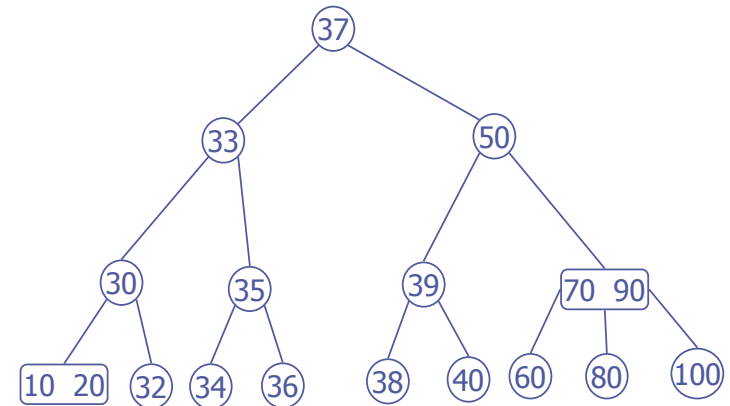
Insert 32



Insert 32



Insert 32

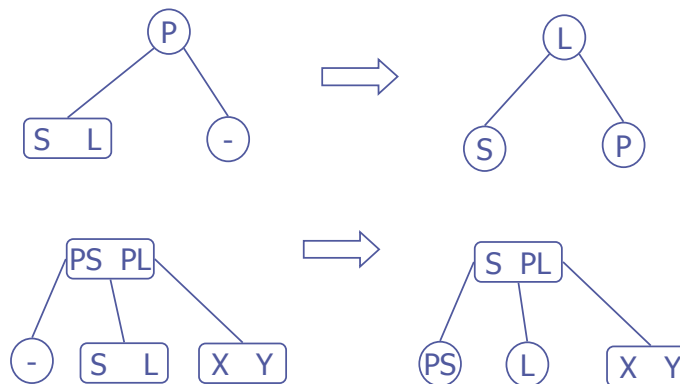


Remove

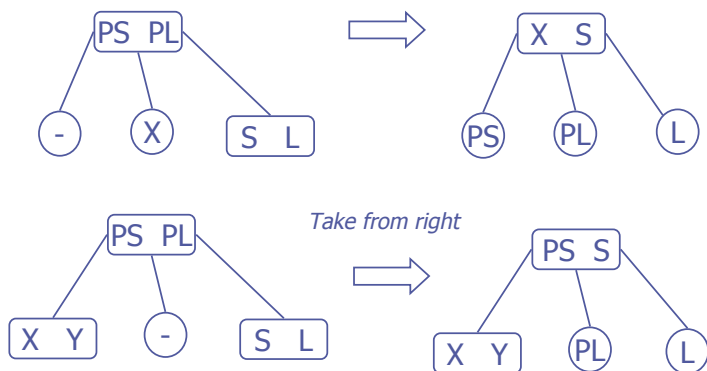
- ◆ With insertion, we split nodes. With removing, we merge nodes
- ◆ Deletion process needs to begin with a leaf but you might be deleting a value that is not a leaf
 - Swap item to delete with inorder successor



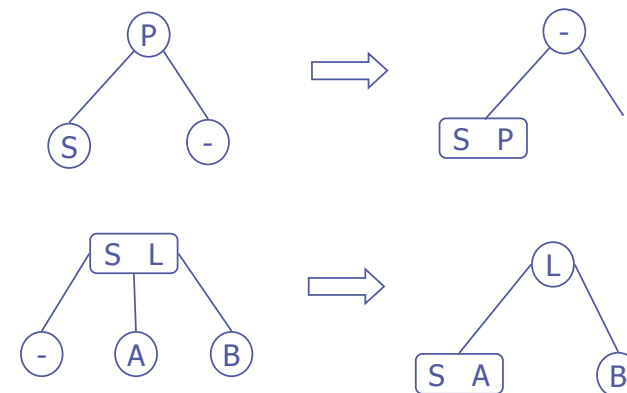
Remove - Redistribute



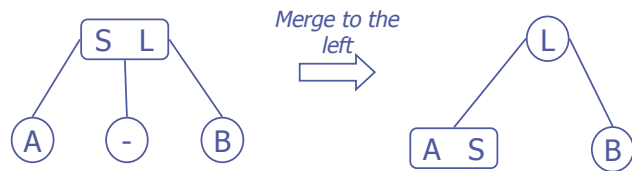
Remove - Redistribute



Remove - Merge



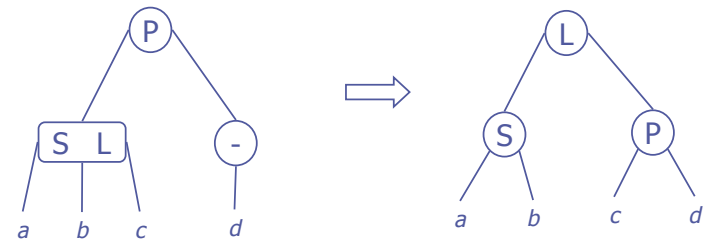
Remove - Merge



2-3 Trees

45

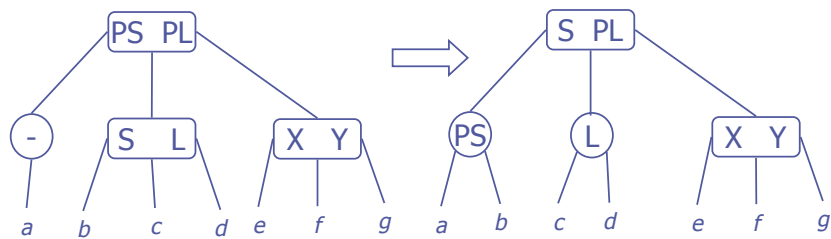
Remove - Redistribute



2-3 Trees

46

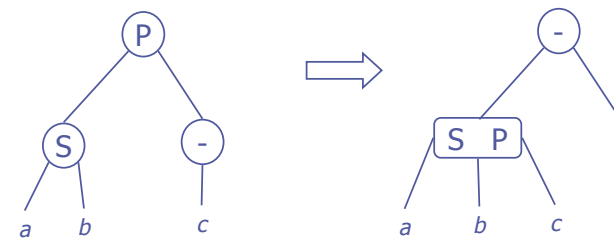
Remove - Redistribute



2-3 Trees

47

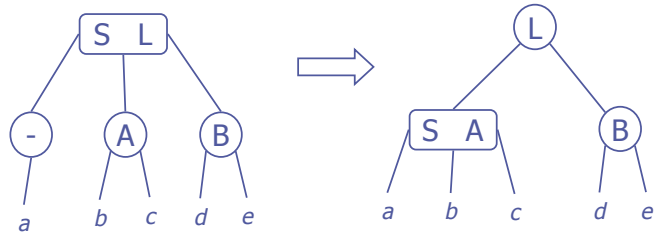
Remove - Merge



2-3 Trees

48

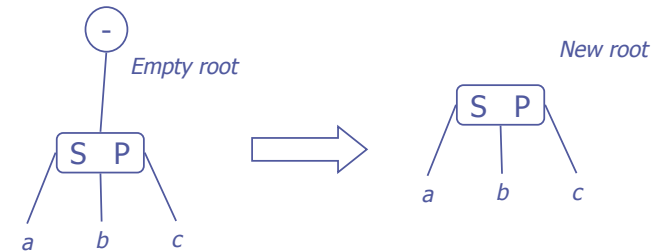
Remove - Merge



2-3 Trees

49

Deleting the Root

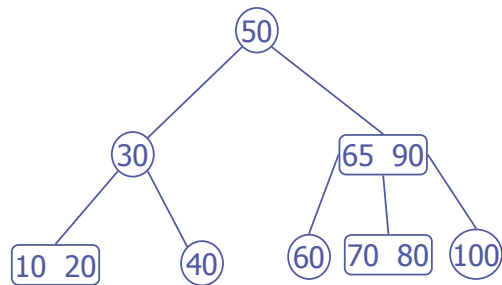


2-3 Trees

50

2-3 Tree

Start with this tree

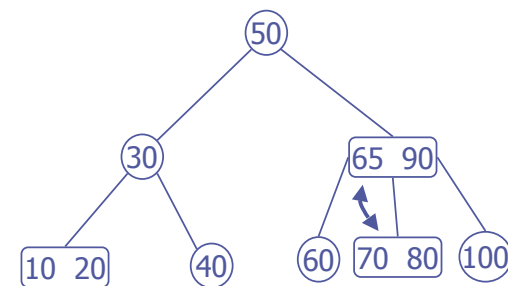


2-3 Trees

51

Remove 65

- ◆ 65 is an internal node - swap with inorder successor
 - Inorder successor will always be in a leaf

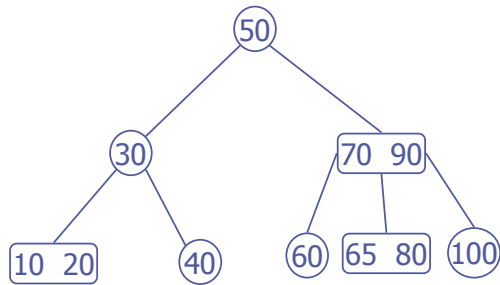


2-3 Trees

52

Remove 65

- ◆ 65 is now in an invalid location but that is okay because we will remove it

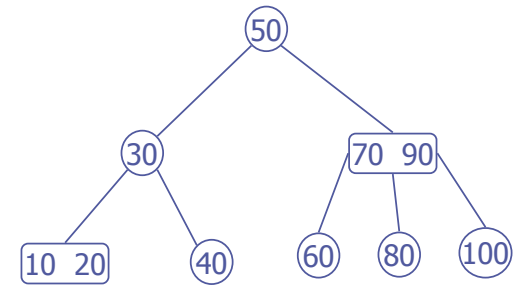


2-3 Trees

53

Remove 65

- ◆ Since there are 2 data values in the leaf, just remove data value

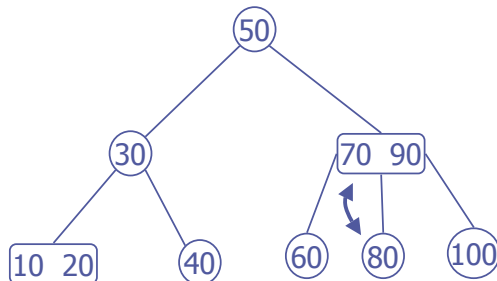


2-3 Trees

54

Delete 70

- ◆ 70 is an internal node - swap with inorder successor
 - Inorder successor will always be in a leaf

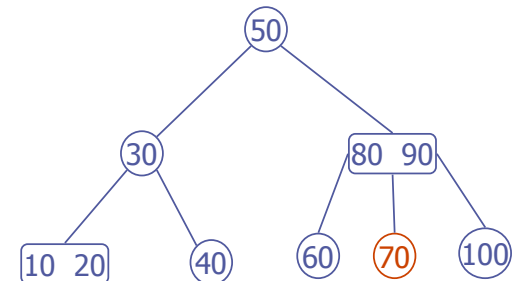


2-3 Trees

55

Delete 70

- ◆ 70 is now in an invalid location but that is okay - we will be removing that node

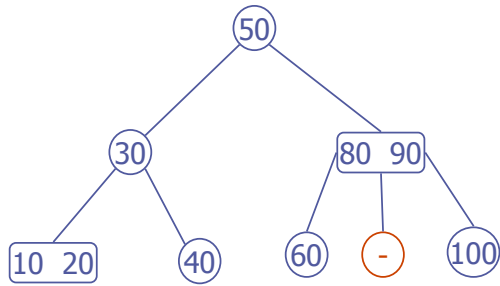


2-3 Trees

56

Delete 70

- ◆ Removing leaf leaves us with an invalid 2-3 tree

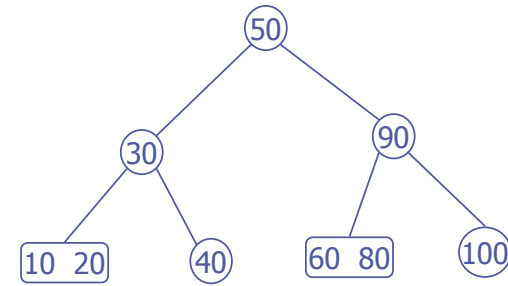


2-3 Trees

57

Delete 70

- ◆ Merge nodes to fix tree

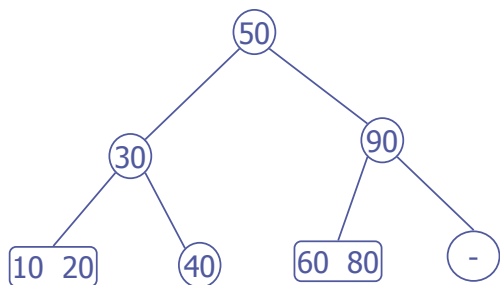


2-3 Trees

58

Delete 100

- ◆ 100 is already leaf, just remove leaf

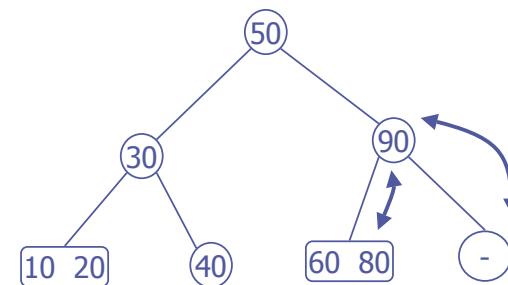


2-3 Trees

59

Delete 100

- ◆ Sibling has data item to spare, redistribute

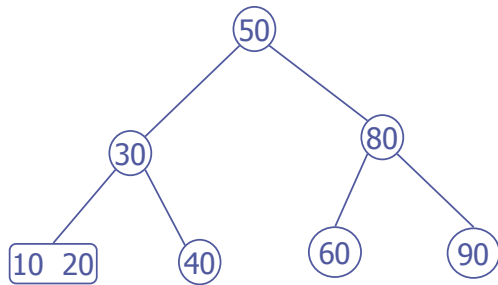


2-3 Trees

60

Delete 100

- ◆ Sibling has data item to spare, redistribute

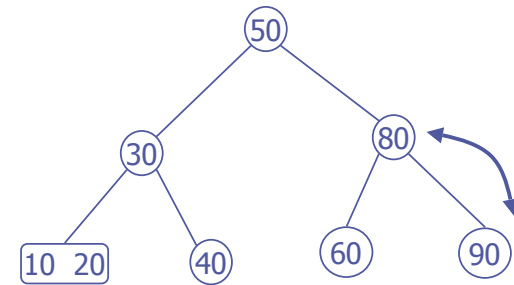


2-3 Trees

61

Delete 80

- ◆ Swap 80 with inorder successor

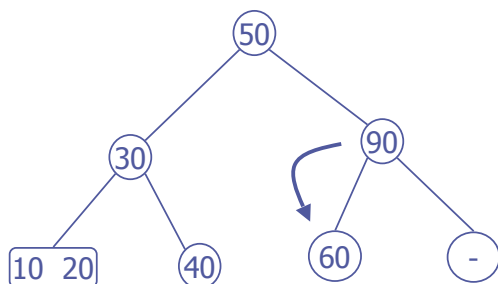


2-3 Trees

62

Delete 80

- ◆ Can't redistribute so merge nodes

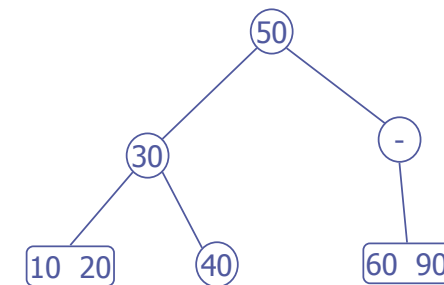


2-3 Trees

63

Delete 80

- ◆ Can't redistribute so merge nodes
- ◆ Invalid 2-3 tree, continue recursively up the tree

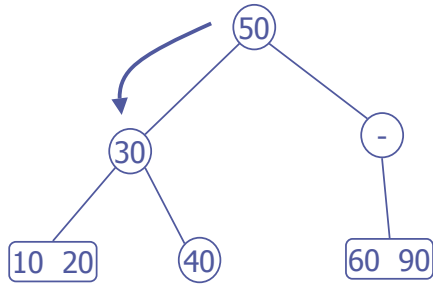


2-3 Trees

64

Delete 80

◆ Can't redistribute so merge nodes

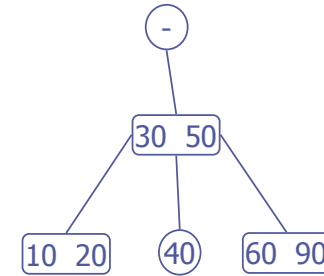


2-3 Trees

65

Delete 80

◆ Can't redistribute so merge nodes

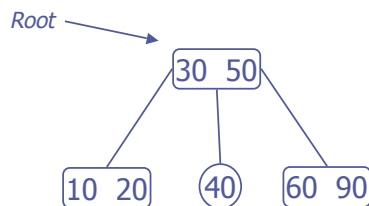


2-3 Trees

66

Delete 80

◆ Root is now empty, set new root pointer



2-3 Trees

67

Delete

deleteItem (itemtype item)

node = node where item exists (may be null if no item)

if (node)

if (item is not in a leaf)

swap item with inorder successor (always leaf)

leafNode = new location of item to delete

delete item from leafNode

if (leafNode now contains no items)

fix (leafNode)

2-3 Trees

68

Delete

*// completes the deletion when node n is empty by either
// removing the root, redistributing values, or merging nodes.*

// Note: if n is internal, it has only one child

```
fix (Node* n, ...) //may need more parameters {  
    if (n is the root) {  
        remove the root  
        set new root pointer  
    }  
    else {  
        Let p be the parent of n
```

Delete

```
if ( some sibling of n has two items ) {  
    Distribute items appropriately among n, the  
    sibling and the parent (remember take from  
    right first)
```

```
if ( n is internal ) {  
    Move the appropriate child from sibling n  
    (May have to move many children if  
    distributing across multiple siblings)
```

```
}
```

Delete

```
else { //merge nodes
```

*Choose an adjacent sibling s of n (remember,
merge left first)*

Bring the appropriate item down from p into s

```
if ( n is internal )
```

move n's child to s

Remove node n

```
if ( p is now empty )
```

fix (p)

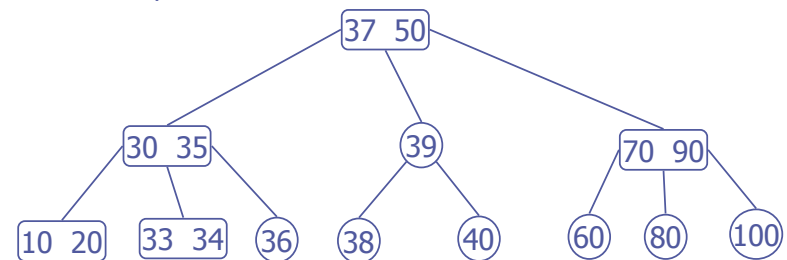
```
} //end if
```

```
//end if
```

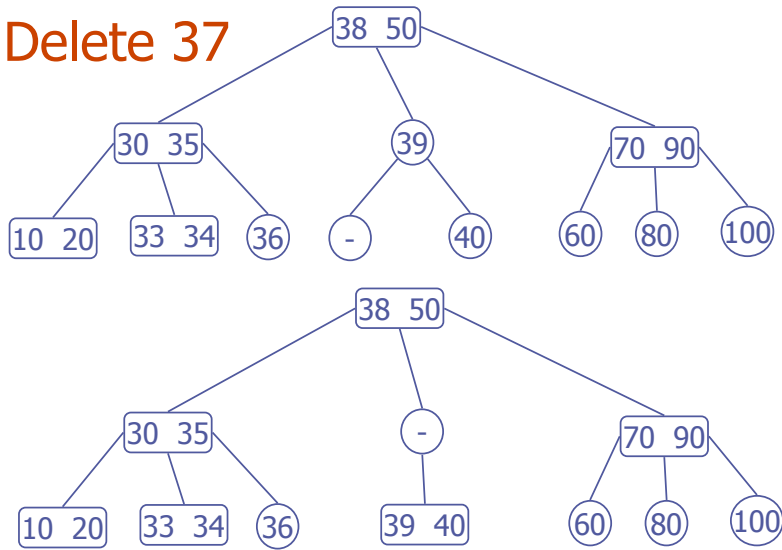
Delete

◆ In class exercise - remove the following values from the tree in this order

- 37, 70



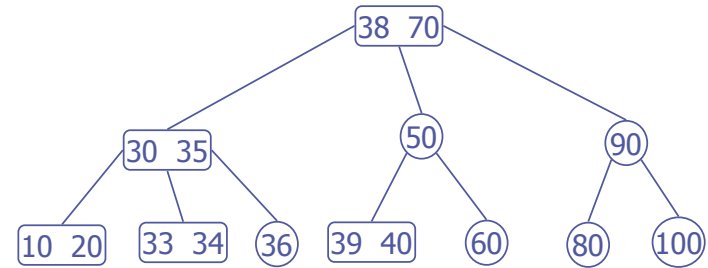
Delete 37



2-3 Trees

73

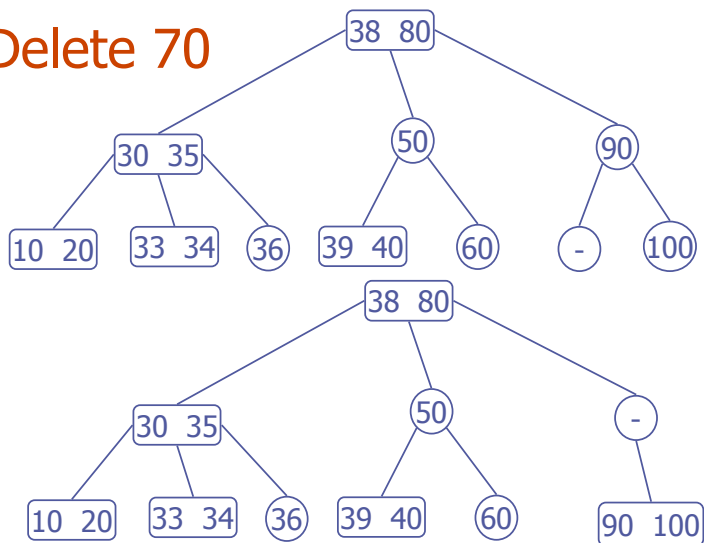
Delete 37



2-3 Trees

74

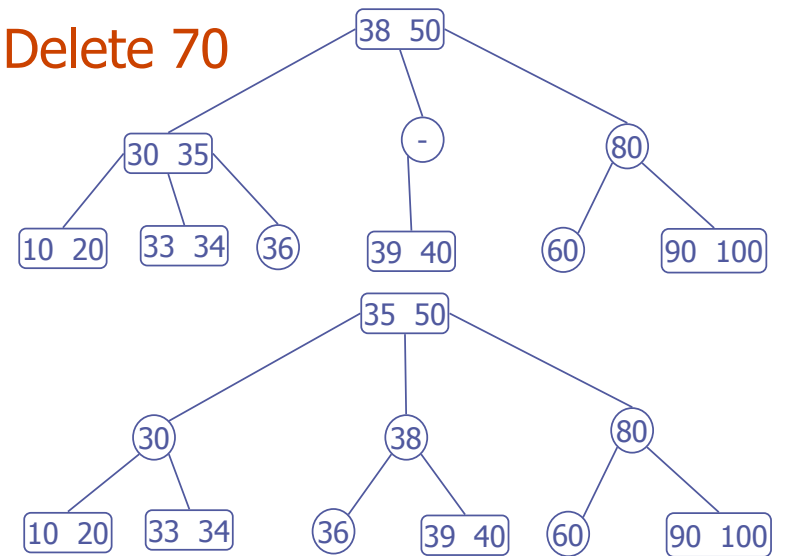
Delete 70



2-3 Trees

75

Delete 70



2-3 Trees

76