Balanced Trees

- AVL Trees
- 2-3 Trees (B Trees)
- Red-Black Trees

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
  - Also called self-balancing trees

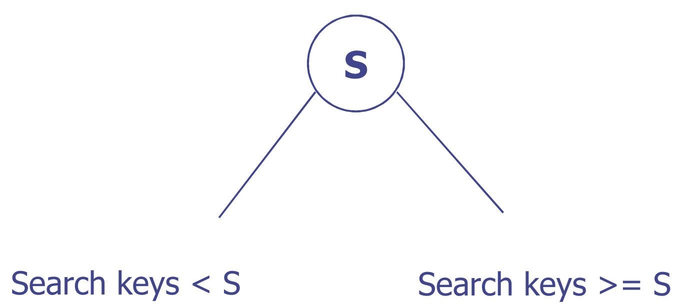
2-3 Tree

- 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

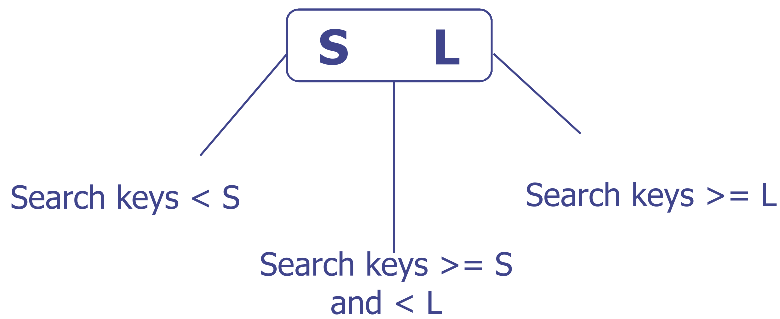
- 2 nodes have one data item and 2 children



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)

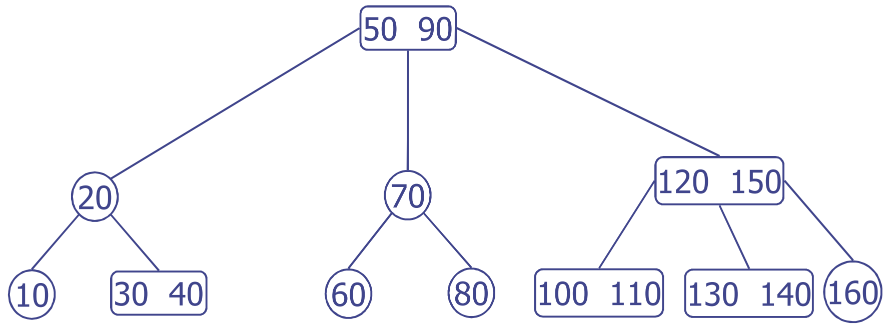


2-3 Tree

- 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
```

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)

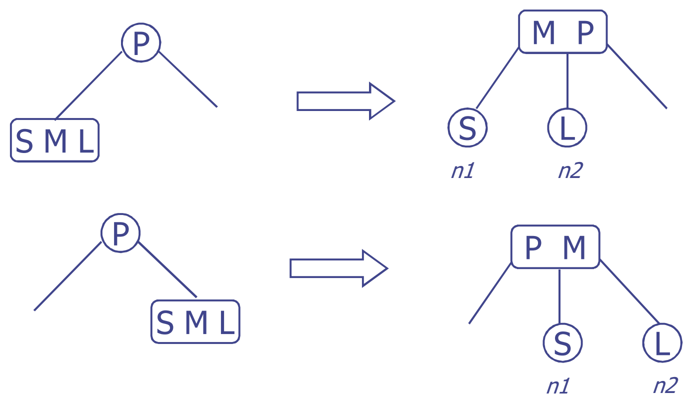
Searching a 2-3 Tree

```
// Assumes small and large exist. You would 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 subtree
                        else if (key < cur->large)
                                search down middle subtree
                        else
                                search down right subtree
```


Insert

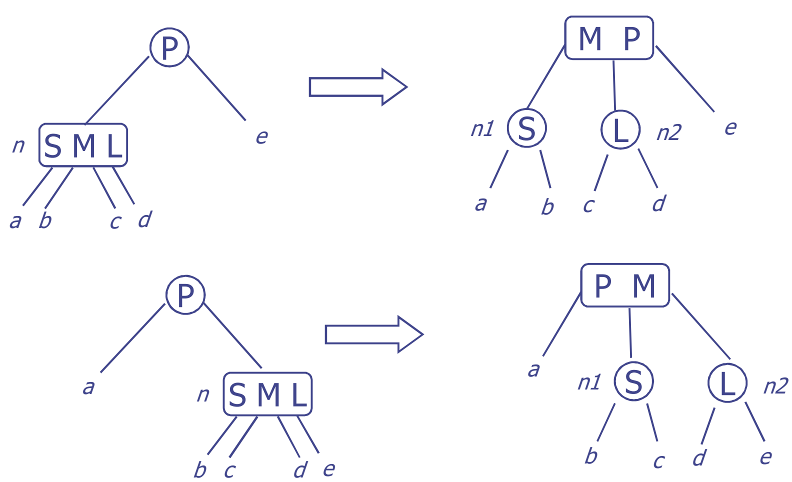
- To insert an item, find a leaf to put the item in then split nodes if necessary
    - Always insert into existing leaf

Splitting a Leaf

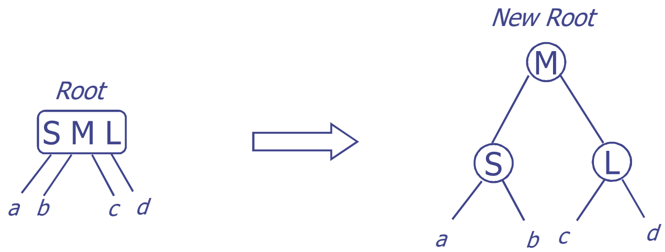


If splitting node causes the parent to have 3 items and 4 children, you will then split an internal node…

Splitting an Internal Node



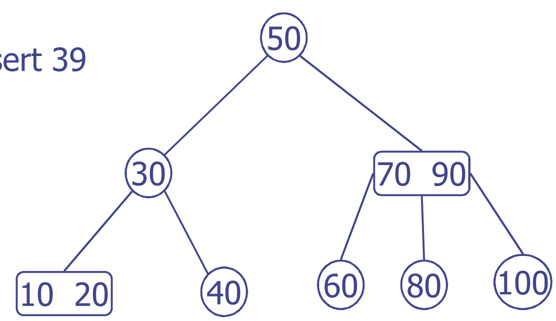
Splitting the Root



2-3 Tree

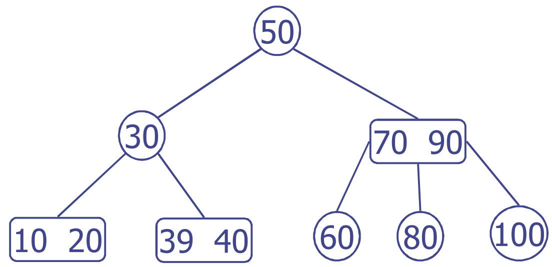
Start with this tree

Insert 39



Insert 39

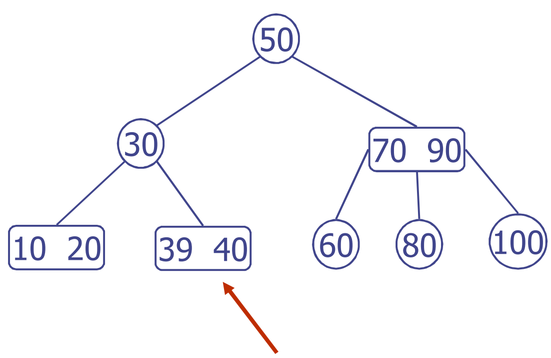
- Locate leaf to insert 39



- Leaf to insert only has 1 data item
  - Add 39 to the leaf

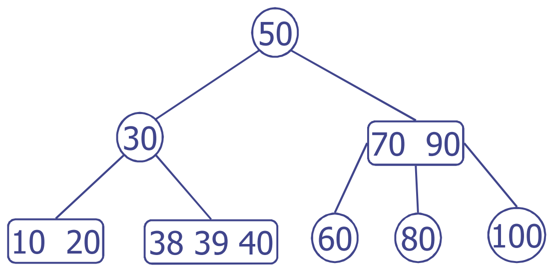
Insert 38

- Locate leaf to insert 38



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

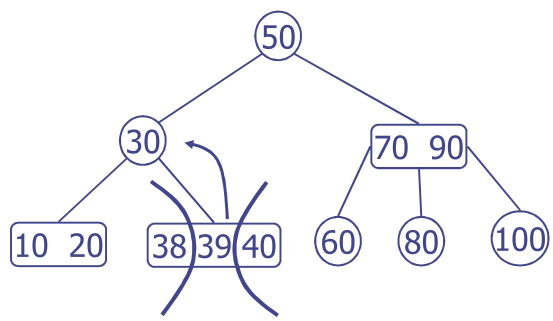
Insert 38

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

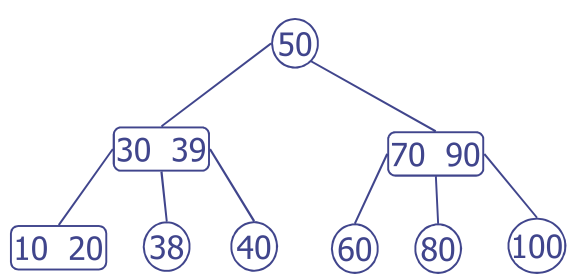
2-3 Trees 18

Insert 38

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

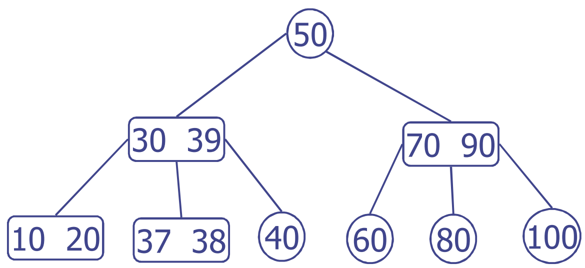


Insert 38



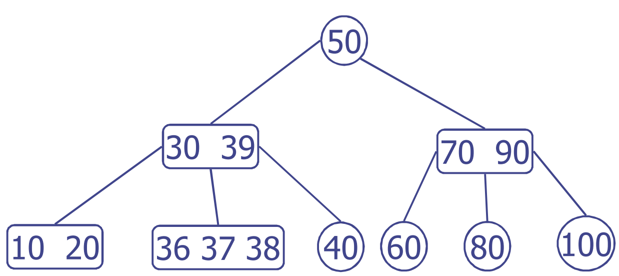
Insert 37

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



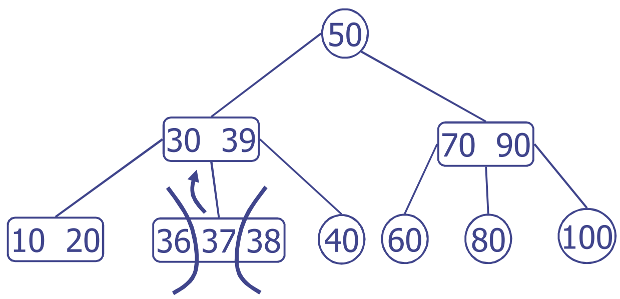
Insert 36

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



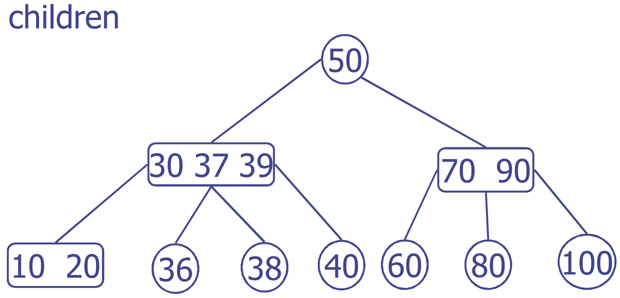
Insert 36

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



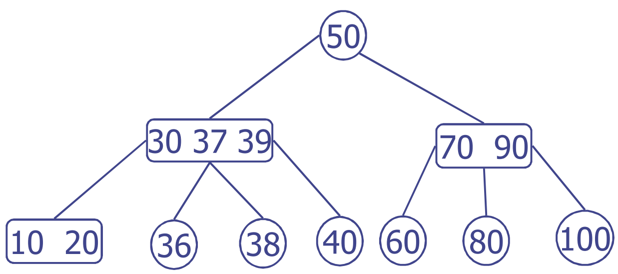
Insert 36

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



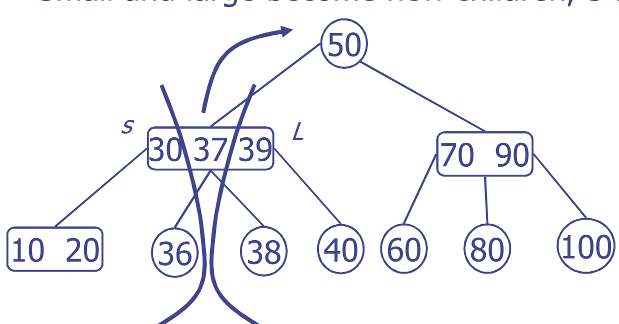
Insert 36

- 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



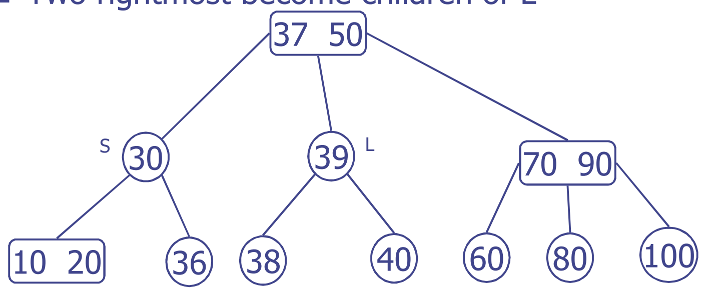
Insert 36

- 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



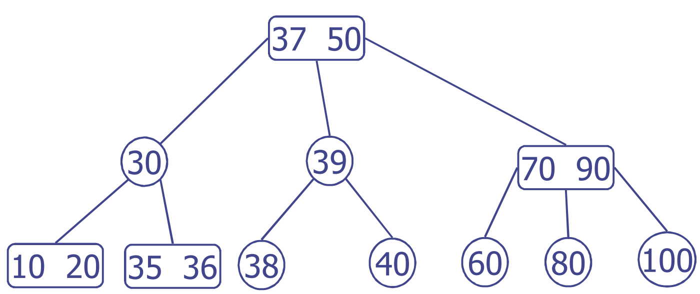
Insert 36

- Divide 4 children
  - Two leftmost become children of S
  - Two rightmost become children of L



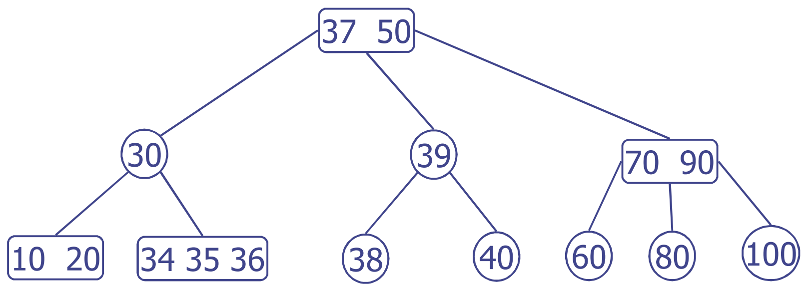
Insert 35

- Insert 35
  - Inserts into leaf



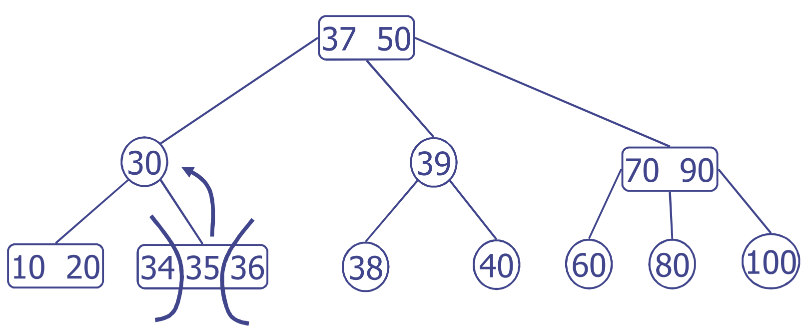
Insert 34

- Insert 34
  - Causes a split



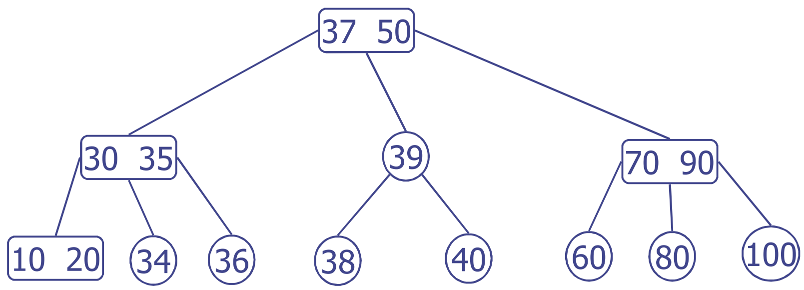
Insert 34

- Insert 34
  - Causes a split



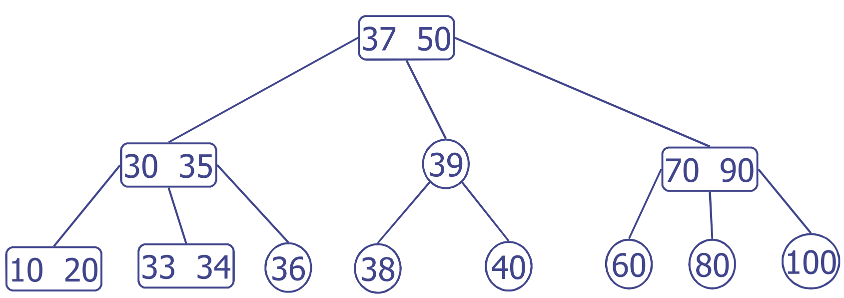
Insert 34

- Insert 34
  - Causes a split



Insert 33

- Insert 33
  - Inserts into leaf



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 n.  The insertion
// of item will cause n to have 3 items so n
// must be split
split ( Node *n, itemtype item, ... // you may need more )
        if ( n 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)

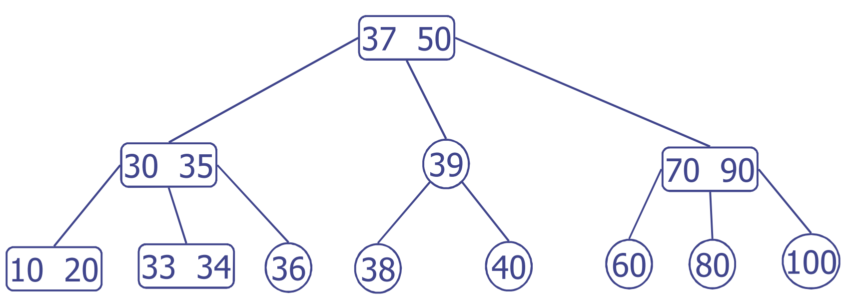
if (n is not a leaf)
n1 becomes the parent of n's two leftmost children
n2 becomes the parent of n's two rightmost children

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

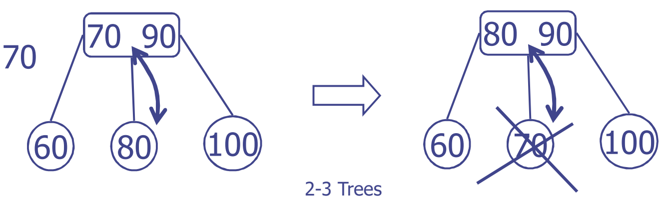
Insert 32

Insert 32

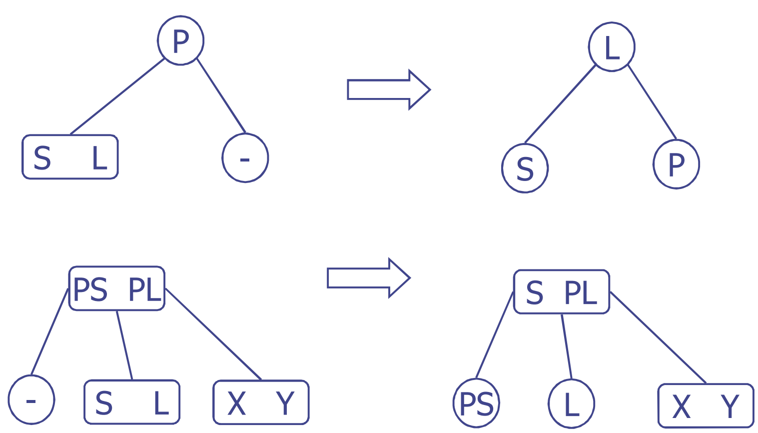
Remove

- 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

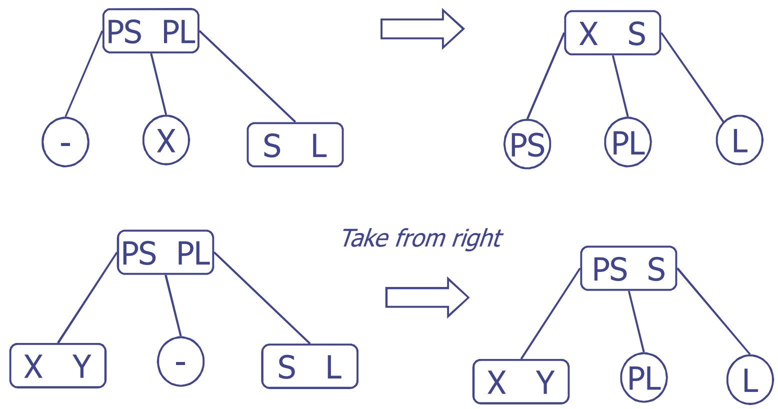
Delete 70



Remove - Redistribute

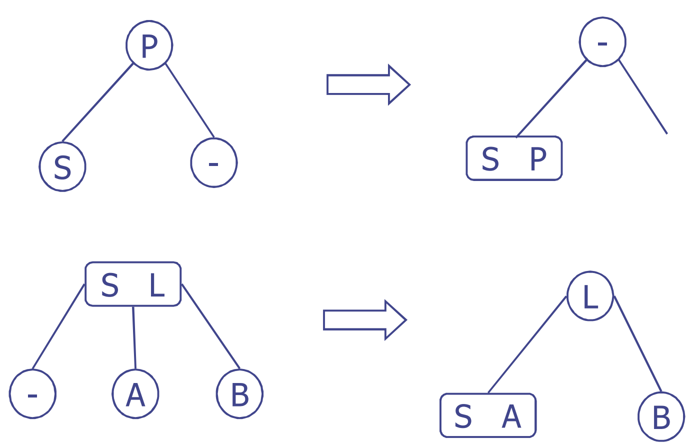


Remove - Redistribute



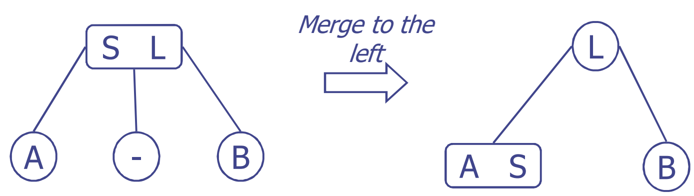
2-3 Trees 43

Remove - Merge

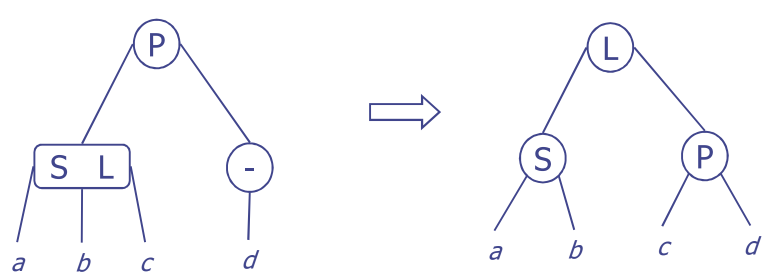


2-3 Trees 44

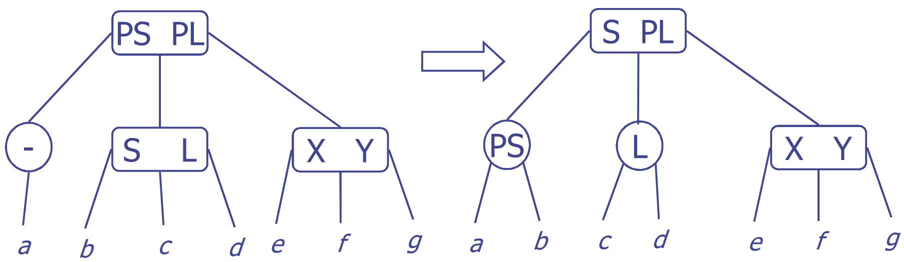
Remove - Merge



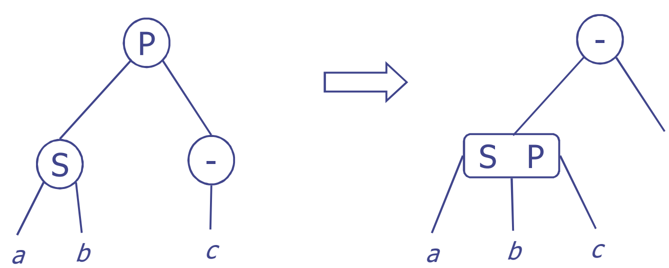
Remove - Redistribute



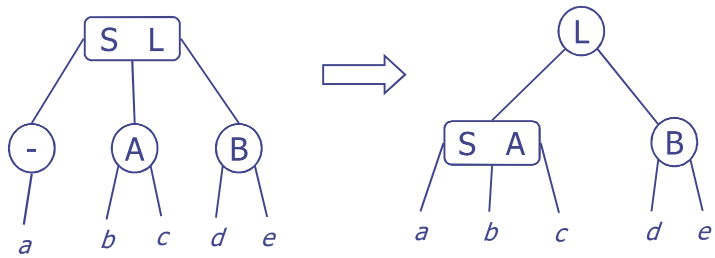
Remove - Redistribute



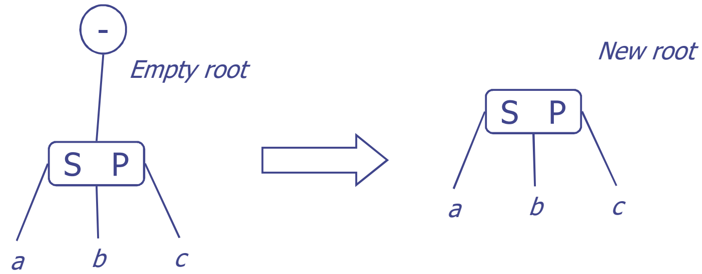
Remove - Merge



Remove - Merge

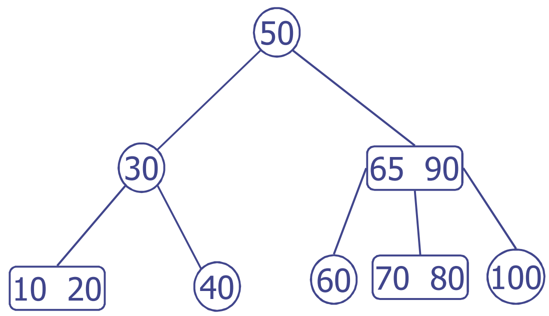


Deleting the Root



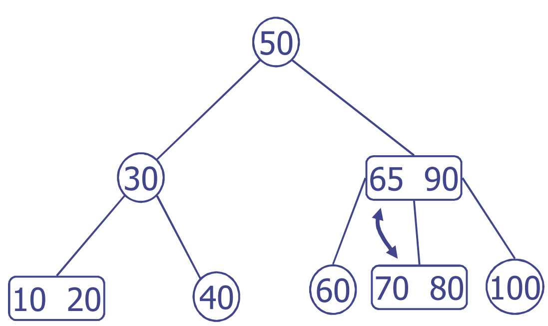
2-3 Tree

Start with this tree



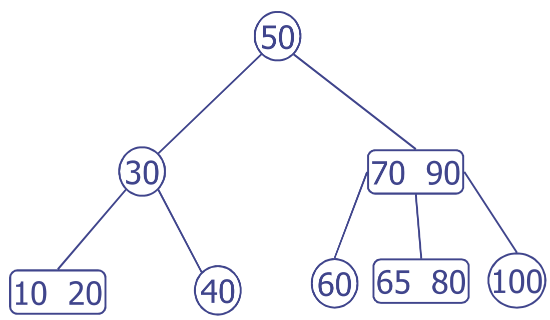
Remove 65

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



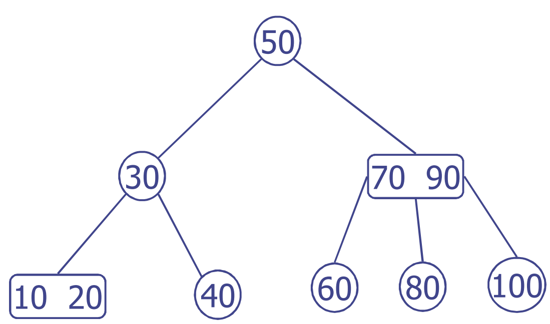
Remove 65

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



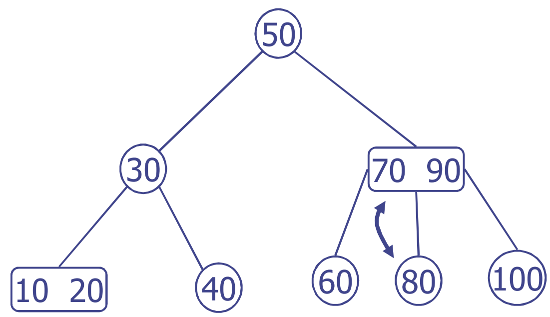
Remove 65

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



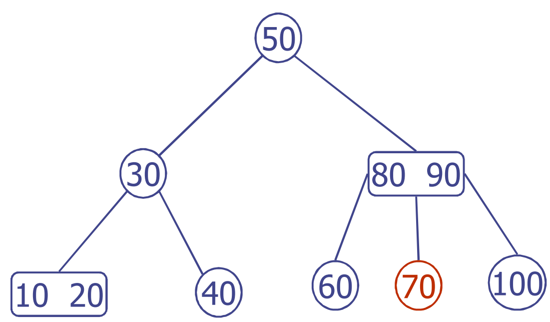
Delete 70

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



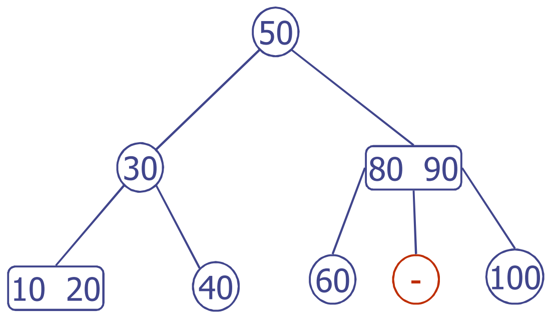
Delete 70

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



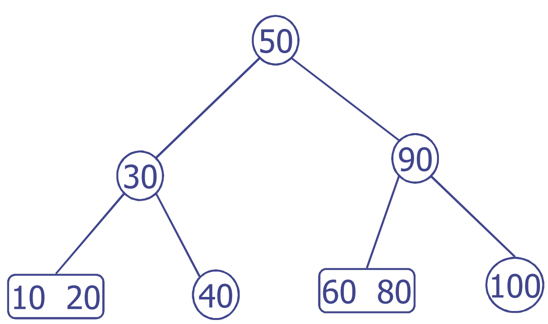
Delete 70

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



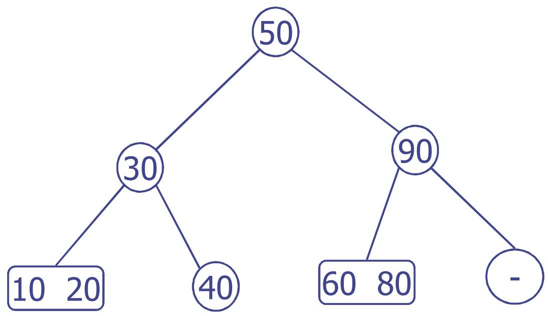
Delete 70

- Merge nodes to fix tree



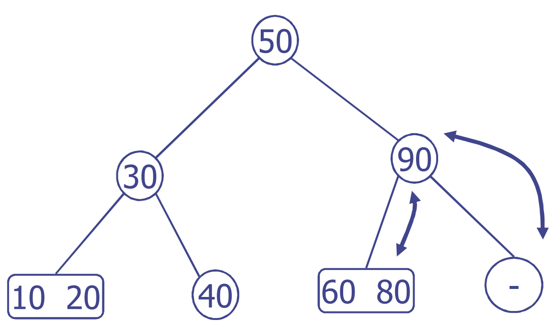
Delete 100

- 100 is already leaf, just remove leaf



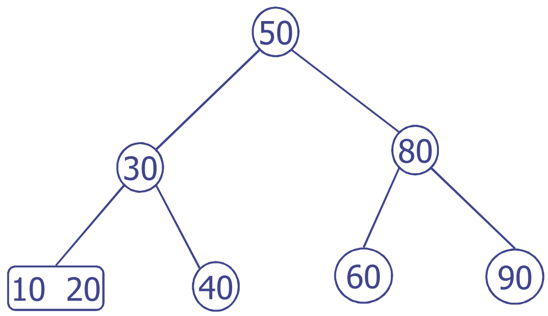
Delete 100

- Sibling has data item to spare, redistribute



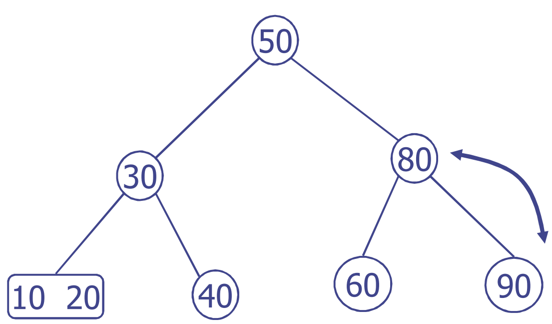
Delete 100

- Sibling has data item to spare, redistribute



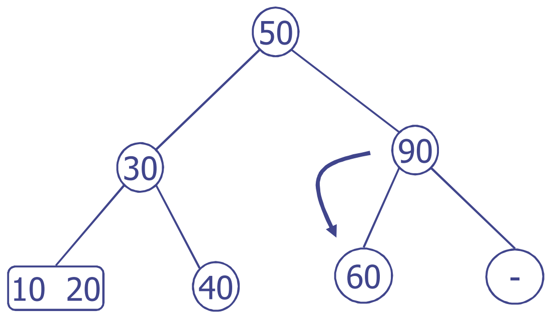
Delete 80

- Swap 80 with inorder successor



Delete 80

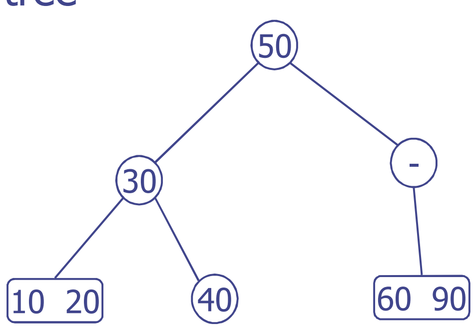
- Can't redistribute so merge nodes



2-3 Trees 63

Delete 80

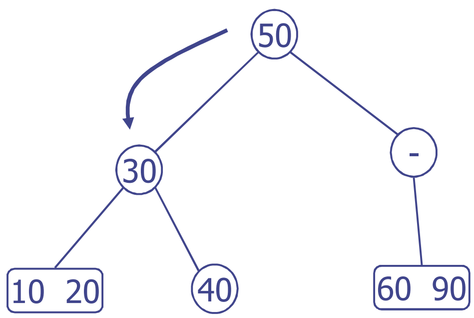
- Can't redistribute so merge nodes
- Invalid 2-3 tree, continue recursively up the tree



2-3 Trees 64

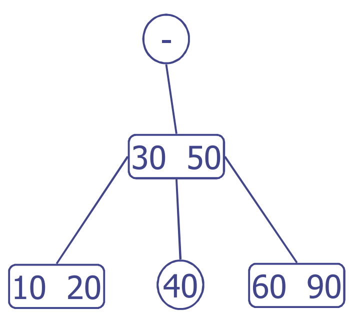
Delete 80

- Can't redistribute so merge nodes



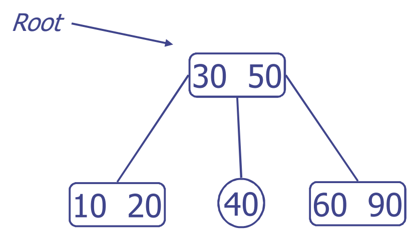
Delete 80

- Can't redistribute so merge nodes



Delete 80

- Root is now empty, set new root pointer



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
else
leafNode = node

delete item from leafNode

if (leafNode now contains no items)
fix (leafNode)

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 n's sibling
            (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

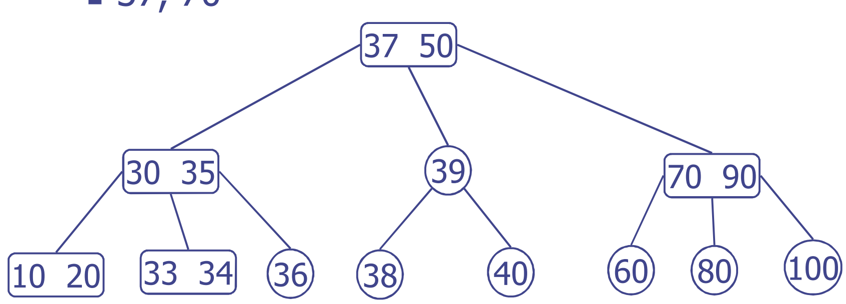
if (n is internal)
move n's child to s

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