```
catch(...){
printf(
"Assignment::SolveProblem() AAAA!");
}
```

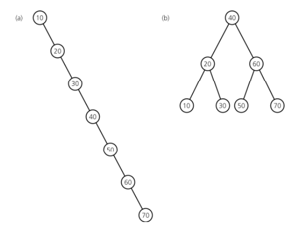
Outline

- Balanced Search Trees
  - 2-3 Trees
  - 2-3-4 Trees

ADD SLIDES ON DISJOINT SETS

Why care about advanced implementations?

Same entries, different insertion sequence:



→ Not good! Would like to keep tree balanced.

2-3 Tree

www.serc.iisc.ernet.in/~viren/Courses/2009/SE286/2-3Trees-mod.ppt

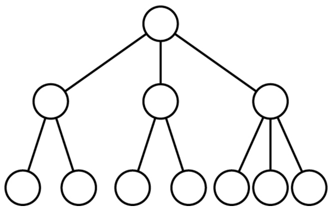
B-TREE

- B-tree keeps data sorted and allows searches, sequential access, insertions, and deletions in log(n).
- The B-tree is a generalization of a BST (node can have more than two children)
- Unlike balanced BST, the B-tree is optimized for systems that read and write.
- Used in databases and filesystems.

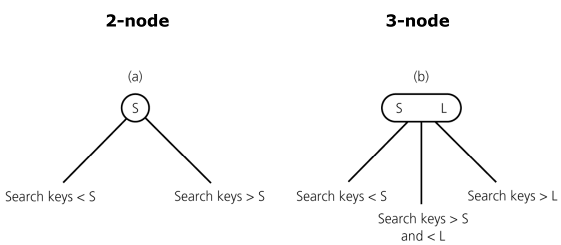
2-3 Trees

Features

- each internal node has either 2 or 3 children
- all leaves are at the same level

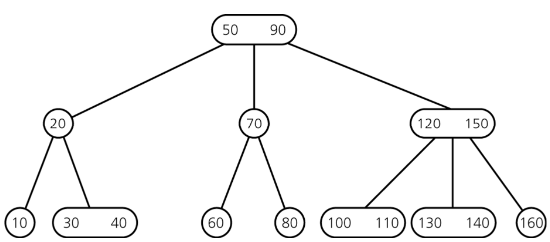


2-3 Trees with Ordered Nodes

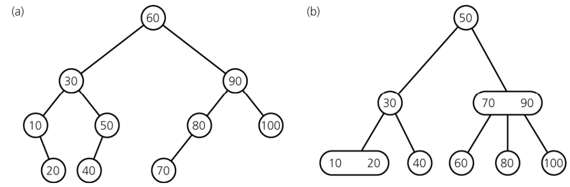


- leaf node can be either a 2-node or a 3-node

Example of 2-3 Tree

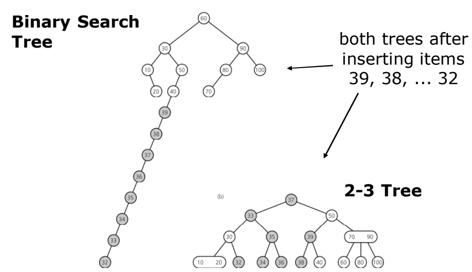


What did we gain?



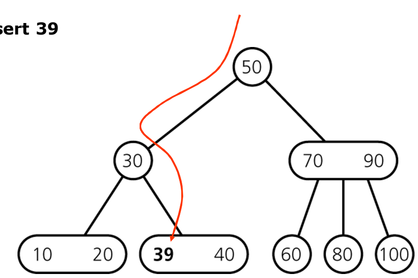
What is the time efficiency of searching for an item?

Gain: Ease of Keeping the Tree Balanced



Inserting Items

Insert 39



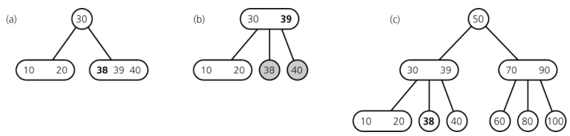
Inserting Items

Insert 38

insert in leaf

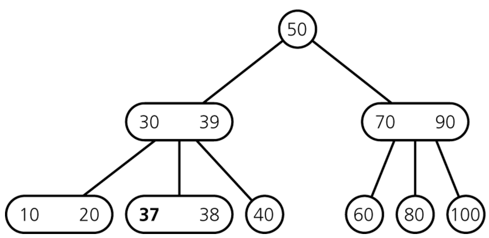
divide leaf
and move middle
value up to parent

result



Inserting Items

Insert 37

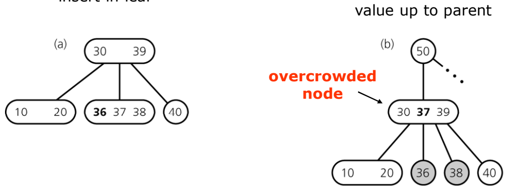


Inserting Items

Insert 36

insert in leaf

divide leaf
and move middle
value up to parent

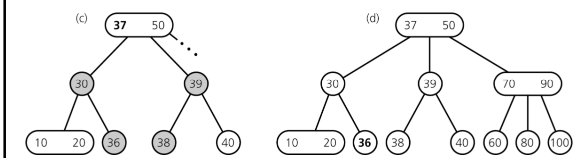


Inserting Items

... still inserting 36

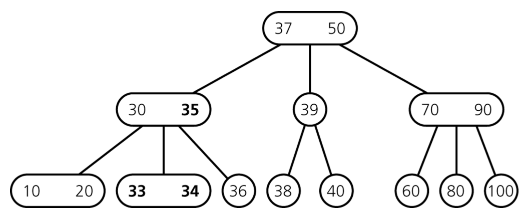
divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

result

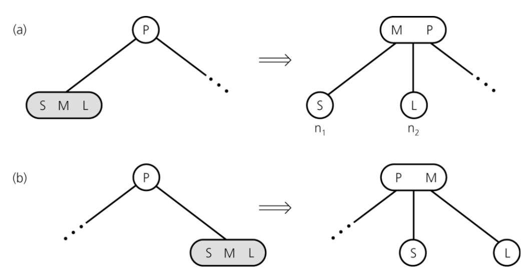


Inserting Items

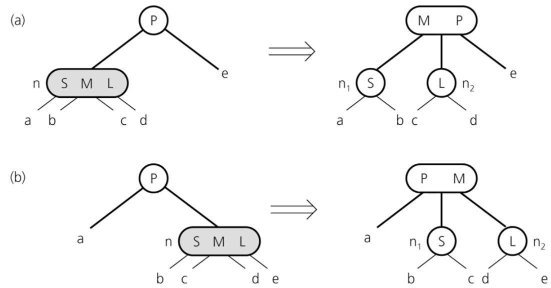
After Insertion of 35, 34, 33



Inserting so far

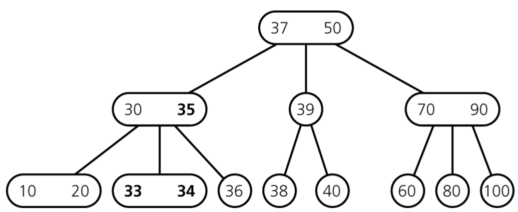


Inserting so far



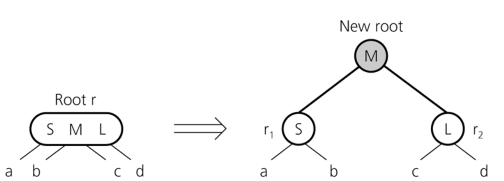
Inserting Items

How do we insert 32?



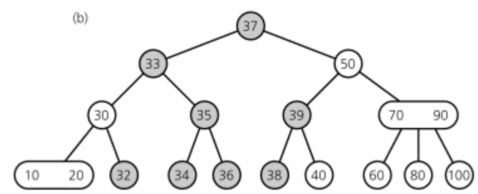
Inserting Items

- → **creating a new root if necessary**
- → **tree grows at the root**



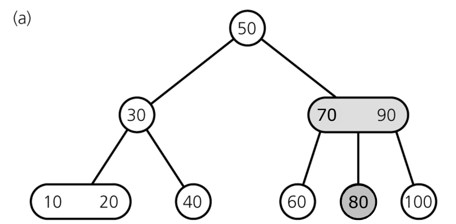
Inserting Items

Final Result



Deleting Items

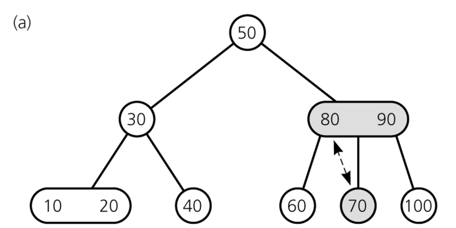
Delete 70



Swap with inorder successor

Deleting Items

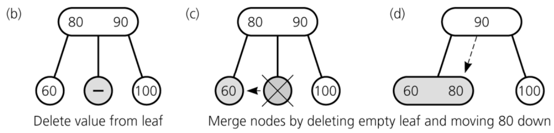
Deleting 70: swap 70 with inorder successor (80)



Swap with inorder successor

Deleting Items

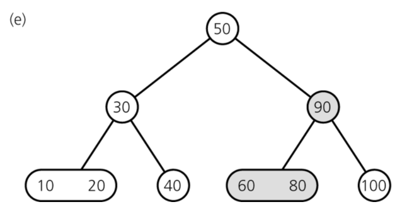
Deleting 70: … get rid of 70



Slide 25

Deleting Items

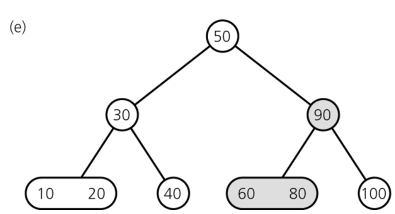
Result



Slide 26

Deleting Items

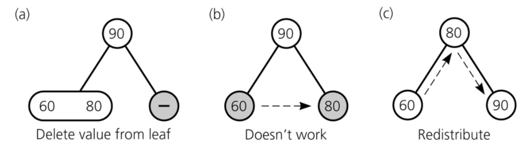
Delete 100



Slide 27

Deleting Items

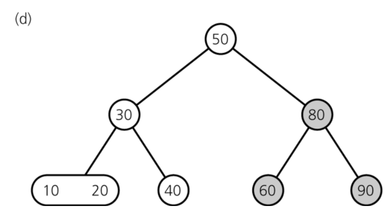
Deleting 100



Slide 28

Deleting Items

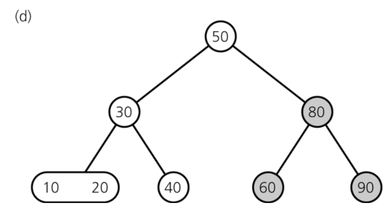
Result



Slide 29

Deleting Items

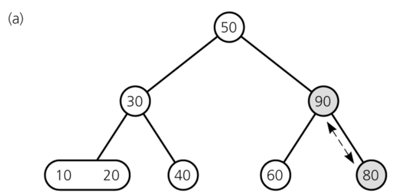
Delete 80



Slide 30

Deleting Items

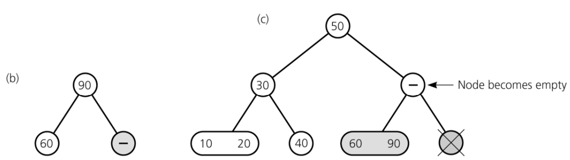
Deleting 80 ...



Swap with inorder successor

Deleting Items

Deleting 80 ...

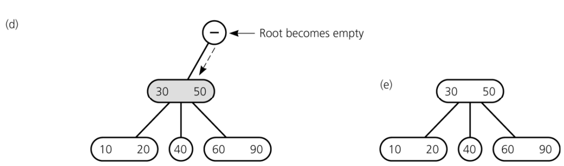


Delete value from leaf

Merge by moving 90 down and removing empty leaf

Deleting Items

Deleting 80 ...

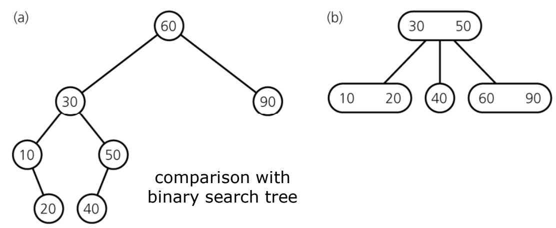


Merge: move 50 down, adopt empty leaf's child, remove empty node

Remove empty root

Deleting Items

Final Result



comparison with binary search tree

Deletion Algorithm I

Deleting item *I*:

1. Locate node *n*, which contains item *I*
2. If node *n* is not a leaf → swap *I* with inorder successor

→ deletion always begins at a leaf

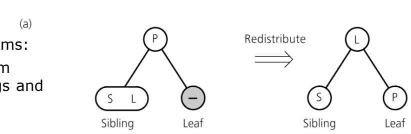
3. If leaf node *n* contains another item, just delete item *I*
   else
   try to redistribute nodes from siblings (see next slide)
   if not possible, merge node (see next slide)

Deletion Algorithm II

Redistribution

A sibling has 2 items:

→ redistribute item between siblings and parent

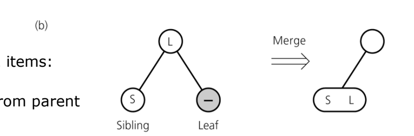


Merging

No sibling has 2 items:

→ merge node

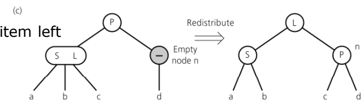
→ move item from parent to sibling



Deletion Algorithm III

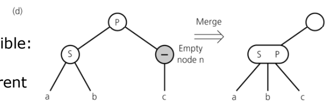
Redistribution

Internal node *n* has no item left
→ redistribute



Merging

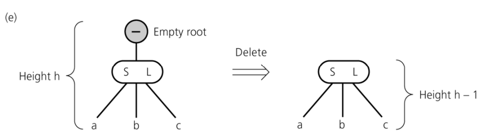
Redistribution not possible:
→ merge node
→ move item from parent to sibling
→ adopt child of *n*



If *n*'s parent ends up without item, apply process recursively

Deletion Algorithm IV

If merging process reaches the root and root is without item
→ delete root



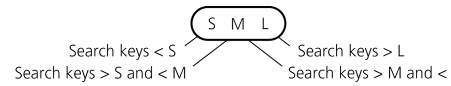
Operations of 2-3 Trees

all operations have time complexity of log n

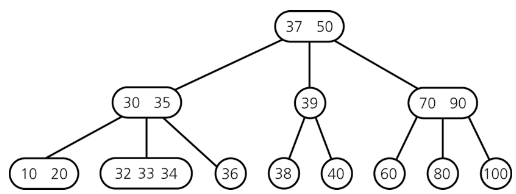
2-3-4 Trees

- **similar to 2-3 trees**
- **4-nodes can have 3 items and 4 children**

4-node



2-3-4 Tree Example



2–3–4 Trees and Red-Black Trees

- 2–3–4 trees are an isometry of red-black trees
  - for every 2–3–4 tree, there exists red–black tree with data elements in the same order.
  - operations on 2–3–4 trees that cause node expansions, splits and merges are equivalent to the color-flipping and rotations in red–black trees.
- 2–3–4 trees, difficult to implement in most programming languages so RB-trees tend to be used instead.

2-3-4 Tree: Insertion

Insertion procedure:
- similar to insertion in 2-3 trees
- items are inserted at the leafs
- since a 4-node cannot take another item, 4-nodes are split up during insertion process

Strategy
- on the way from the root down to the leaf: split up all 4-nodes "on the way"

→ insertion can be done in one pass
(remember: in 2-3 trees, a reverse pass might be necessary)

Slide 43

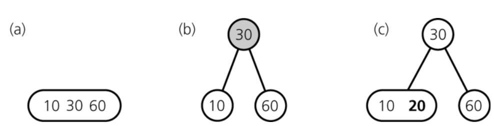
2-3-4 Tree: Insertion

Inserting 60, 30, 10, 20, 50, 40, 70, 80, 15, 90, 100

Slide 44

2-3-4 Tree: Insertion

Inserting 60, 30, 10, 20 ...

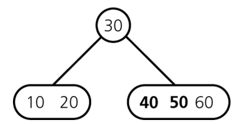


... 50, 40 ...

Slide 45

2-3-4 Tree: Insertion

Inserting 50, 40 ...

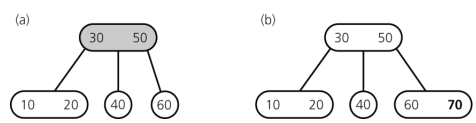


... 70, ...

Slide 46

2-3-4 Tree: Insertion

Inserting 70 ...

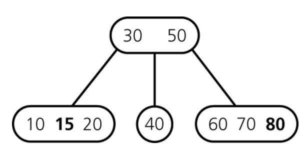


... 80, 15 ...

Slide 47

2-3-4 Tree: Insertion

Inserting 80, 15 ...

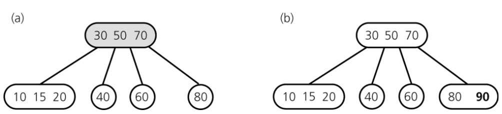


... 90 ...

Slide 48

2-3-4 Tree: Insertion

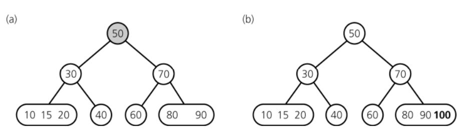
Inserting 90 …



… 100 …

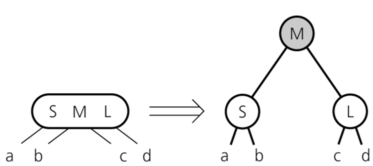
2-3-4 Tree: Insertion

Inserting 100 …



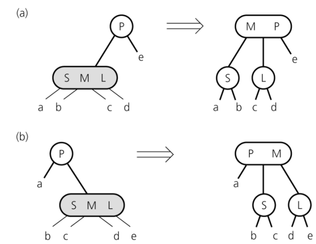
2-3-4 Tree: Insertion Procedure

Splitting 4-nodes during Insertion



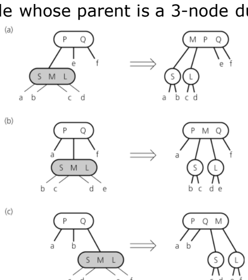
2-3-4 Tree: Insertion Procedure

Splitting a 4-node whose parent is a 2-node during insertion



2-3-4 Tree: Insertion Procedure

Splitting a 4-node whose parent is a 3-node during insertion



2-3-4 Tree: Deletion

Deletion procedure:

- similar to deletion in 2-3 trees
- items are deleted at the leafs
  → swap item of internal node with inorder successor
- note: a 2-node leaf creates a problem

Strategy (different strategies possible)

- on the way from the root down to the leaf:
  turn 2-nodes (except root) into 3-nodes

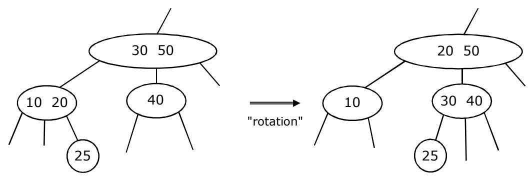
→ deletion can be done in one pass
(remember: in 2-3 trees, a reverse pass might be necessary)

2-3-4 Tree: Deletion

Turning a 2-node into a 3-node …

Case 1: an adjacent sibling has 2 or 3 items

→ "steal" item from sibling by rotating items and moving subtree



Slide 55

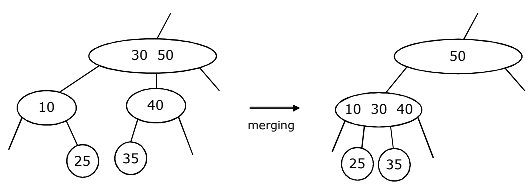
2-3-4 Tree: Deletion

Turning a 2-node into a 3-node …

Case 2: each adjacent sibling has only one item

→ "steal" item from parent and merge node with sibling

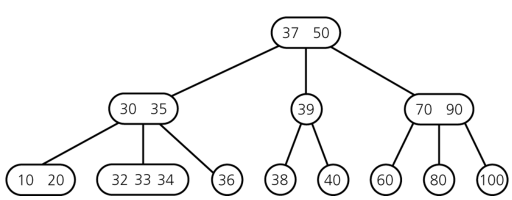
(note: parent has at least two items, unless it is the root)



Slide 56

2-3-4 Tree: Deletion Practice

Delete 32, 35, 40, 38, 39, 37, 60



Slide 57