

TOPIC 2B

DATA STRUCTURES: TREES

BINARY SEARCH TREES

Objective: To access elements in a sorted order

Operations:

Exists: Does an element exist in a set ?

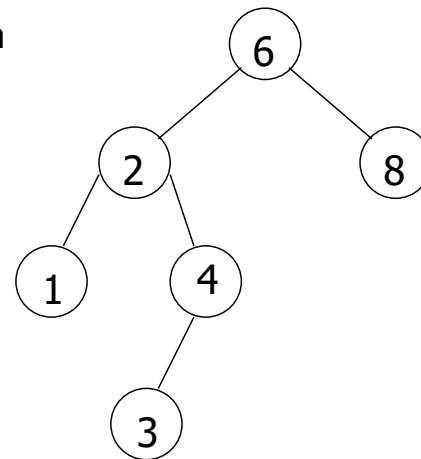
Maximum/Minimum: What is the largest (smallest) element

Insertion

Deletion

BINARY SEARCH TREES

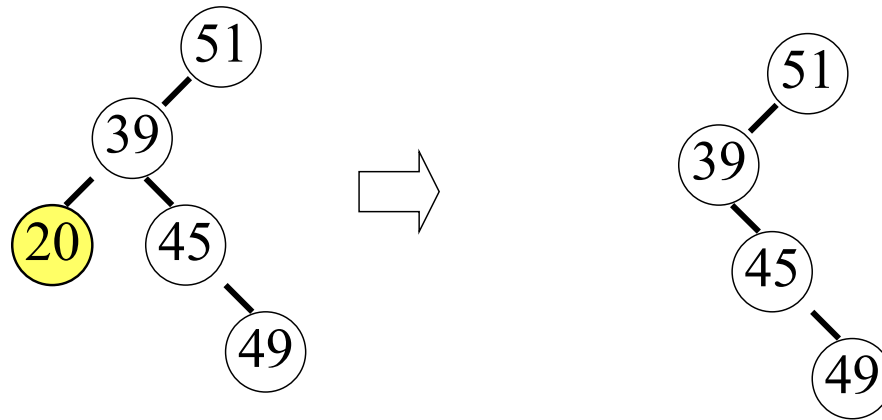
- For every node, X , in the tree, the values of all the items in its left subtree are smaller than the item in X , and the values of all the items in its right subtree are larger than the item in X .



HOW TO REMOVE A NODE?

What if the node is a leaf?

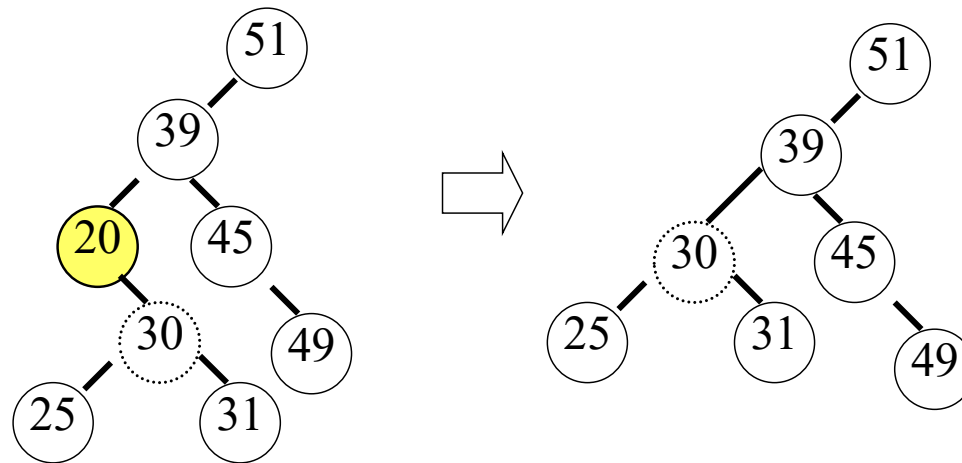
- It can be deleted immediately.



HOW TO REMOVE A NODE?

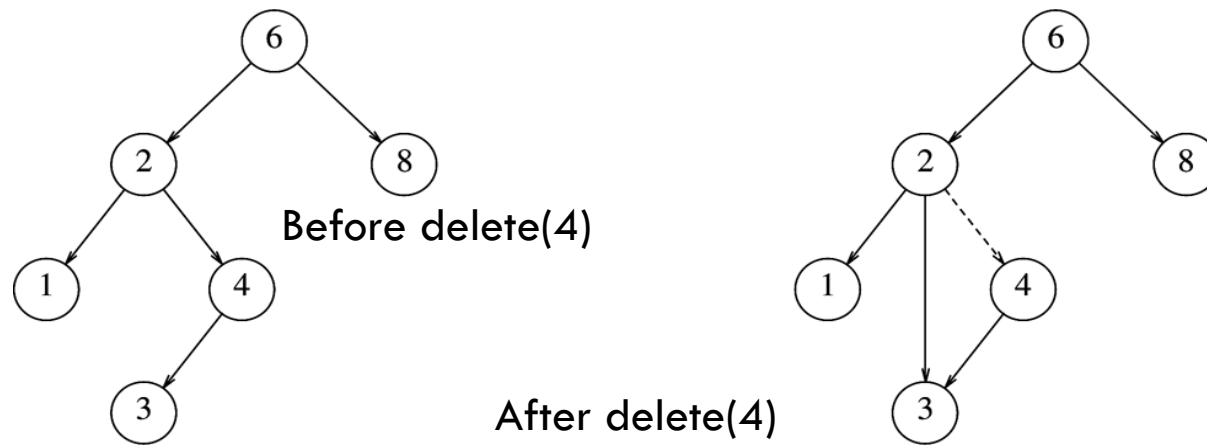
What if the node has one child?

- It can be deleted after its parent adjusts a pointer to bypass the node



BST: DELETION

Deleting a node with one child



Deletion Strategy: Bypass the node being deleted

HOW TO REMOVE A NODE?

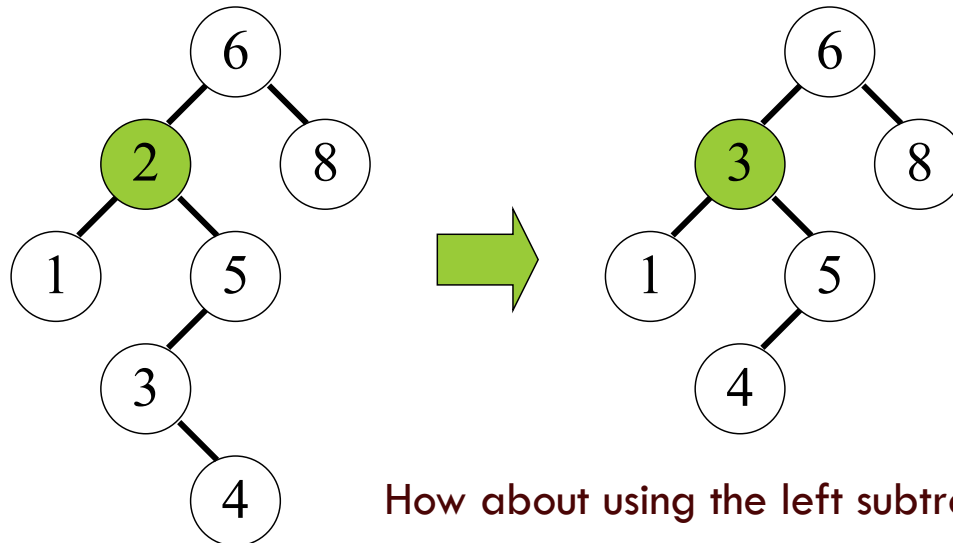
What if the node has two children?

- The general strategy is to replace the data of this node with the **smallest data** of the right subtree, and recursively delete the node.

Find smallest data
in right tree

Replace this value
in the node

Delete smallest
data, since it has
no child



How about using the left subtree

LAZY DELETION

Another deletion strategy

- Don't delete!
- Just mark the node as deleted.
- Wastes space
- But useful if deletions are rare or space is not a concern.
 - Even if the number of “deleted” nodes is the same as the number of real nodes, then the height of the tree is only expected to go up by a small constant
 - If a deleted item is reinserted, the overhead of allocating a new cell is avoided.

FULL BINARY TREE

Theorem: The number of leaves in a non-empty full binary tree is one more than the number of internal nodes.

Proof (by Mathematical Induction):

Base cases: A full, non-empty binary tree with 0 internal nodes must have 1 leaf node. A full binary tree with 1 internal node must have two leaf nodes.

Induction Hypothesis: Assume any full binary tree T containing n internal nodes has $n+1$ leaves.

FULL BINARY TREE

Induction Step: Given tree **T** with n internal nodes, pick leaf node L and make it internal (by adding two children). **T'** is still full. How many internal leaf nodes does it contain?

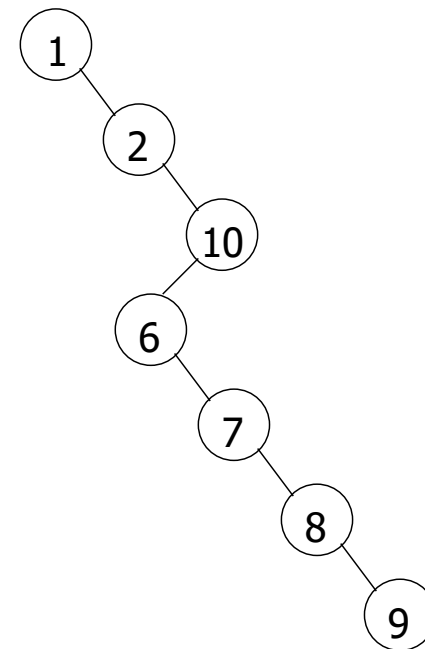
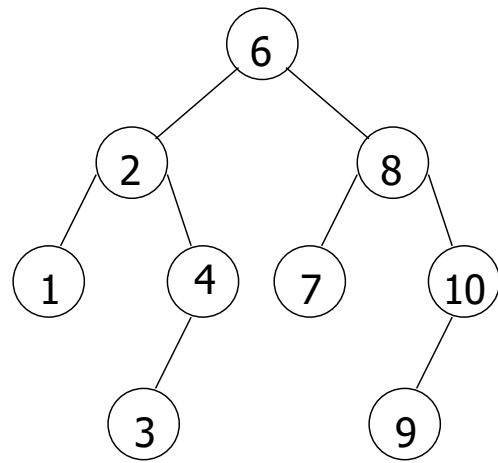
- We added two new leaf nodes, but lost one
 - $n + 1 + 2 - 1 = n + 2$
- We created one new internal node
 - $n + 1$
- T'** has $n+1$ internal nodes, and $n+2$ leaf nodes, or n' internal nodes and $n'+1$ leaf nodes
- By induction, the theorem holds for $n \geq 0$

BINARY SEARCH TREES

For every node, X , in the tree, the values of all the items in its left subtree are smaller than the item in X , and the values of all the items in its right subtree are larger than the item in X .

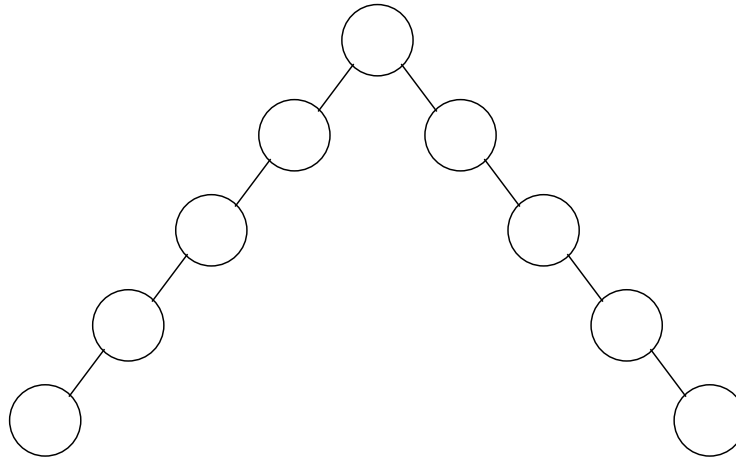
The complexity of most operations on binary search trees are bounded by their heights.

WHICH ONE IS A BINARY SEARCH TREE?



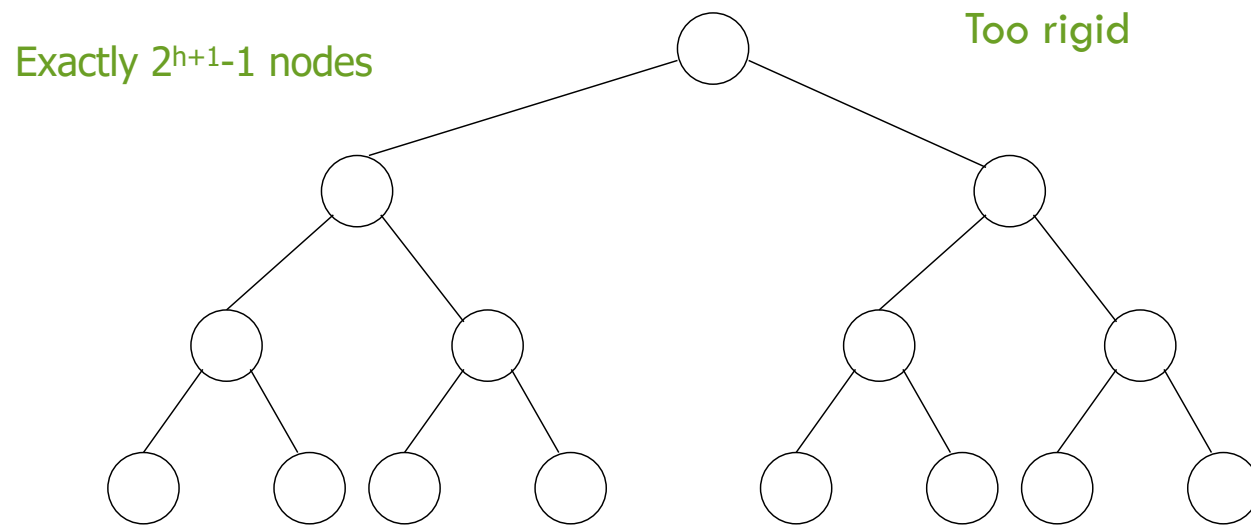
A BAD BINARY SEARCH TREE

Too weak



- One balance condition: the left and right subtrees of the root has the same height.

A PERFECT BINARY SEARCH TREE



- Another balance condition: every node must have left and right subtrees of the same height.

AN AVL TREE

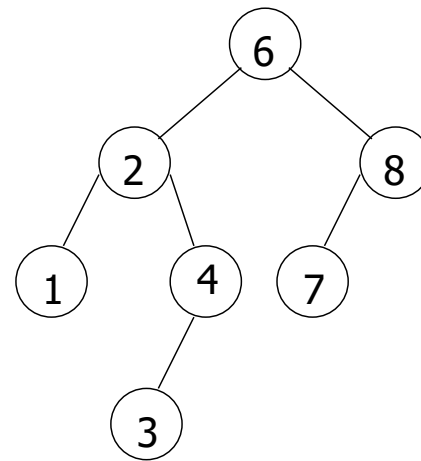
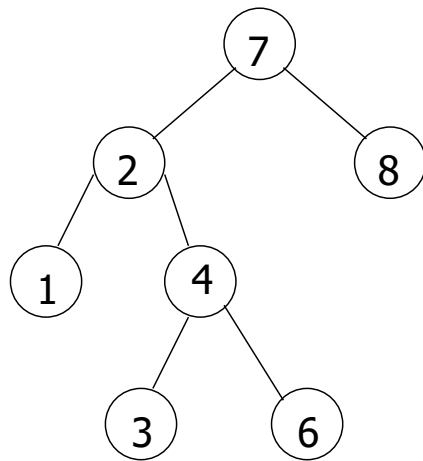
An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of its left and right subtrees can differ by at most 1.

The height of an empty tree is defined to be -1.

The height of an AVL tree is only slightly more than $\log N$.

- Therefore, most of the tree operations can be performed in $O(\log N)$ time.
- It can be shown that the height of an AVL tree is at most roughly $1.44\log(N+2) - .328$.

WHICH ONE IS AN AVL TREE?



WHAT HAPPENS AFTER AN INSERTION?

The balancing information for the nodes on the path from the inserted one back to the root may be changed.

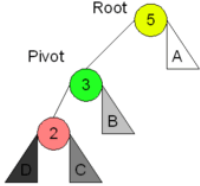
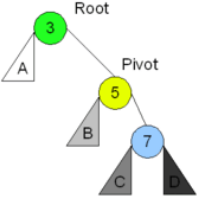
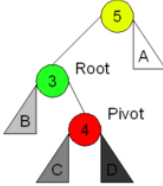
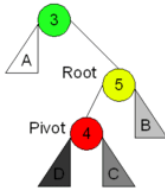
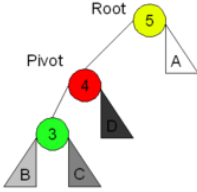
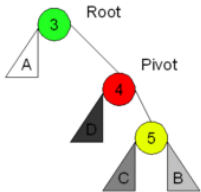
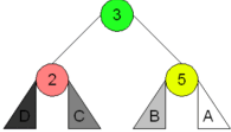
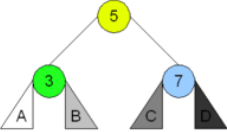
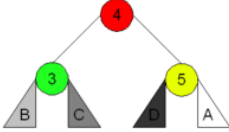
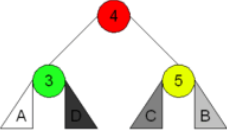
Because only those nodes have their subtrees altered.

We may find a node whose new balance violates the AVL condition.

AVL

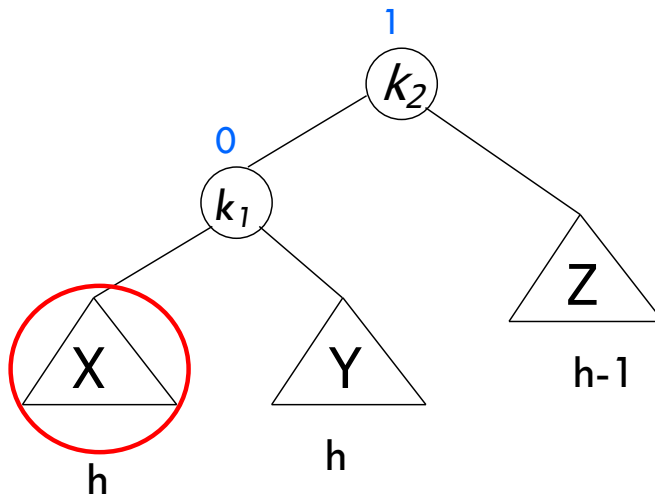
There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.

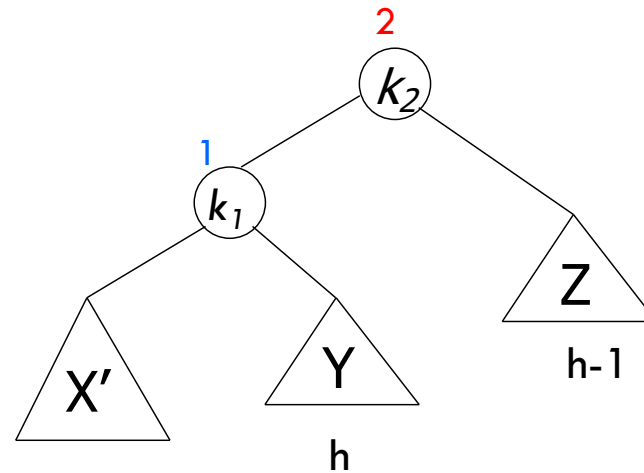
Left Left Case	Right Right Case	Left Right Case	Right Left Case
 <p>Root 5 (yellow) has left child 3 (green) and right child A (white). Node 3 has left child 2 (red) and right child B (gray). Node 2 has left child D (black) and right child C (gray).</p> <p>Right Rotation</p>	 <p>Root 3 (green) has left child A (white) and right child 5 (yellow). Node 5 has left child B (gray) and right child 7 (blue). Node 7 has left child C (gray) and right child D (black).</p> <p>Left Rotation</p>	 <p>Root 5 (yellow) has left child 3 (green) and right child A (white). Node 3 has left child B (gray) and right child 4 (red). Node 4 has left child C (gray) and right child D (black).</p> <p>Left Rotation</p>	 <p>Root 3 (green) has left child A (white) and right child 5 (yellow). Node 5 has left child 4 (red) and right child B (gray). Node 4 has left child D (black) and right child C (gray).</p> <p>Right Rotation</p>
		 <p>Root 5 (yellow) has left child 4 (red) and right child A (white). Node 4 has left child 3 (green) and right child D (black). Node 3 has left child B (gray) and right child C (gray).</p> <p>Right Rotation</p>	 <p>Root 3 (green) has left child A (white) and right child 4 (red). Node 4 has left child D (black) and right child 5 (yellow). Node 5 has left child C (gray) and right child B (gray).</p> <p>Left Rotation</p>
			

Source: <https://medium.com/@randerson112358/avl-trees-a7b4f1fa2d1a>
No copyright infringement is intended

AN INSERTION INTO THE LEFT SUBTREE OF THE LEFT CHILD OF α .

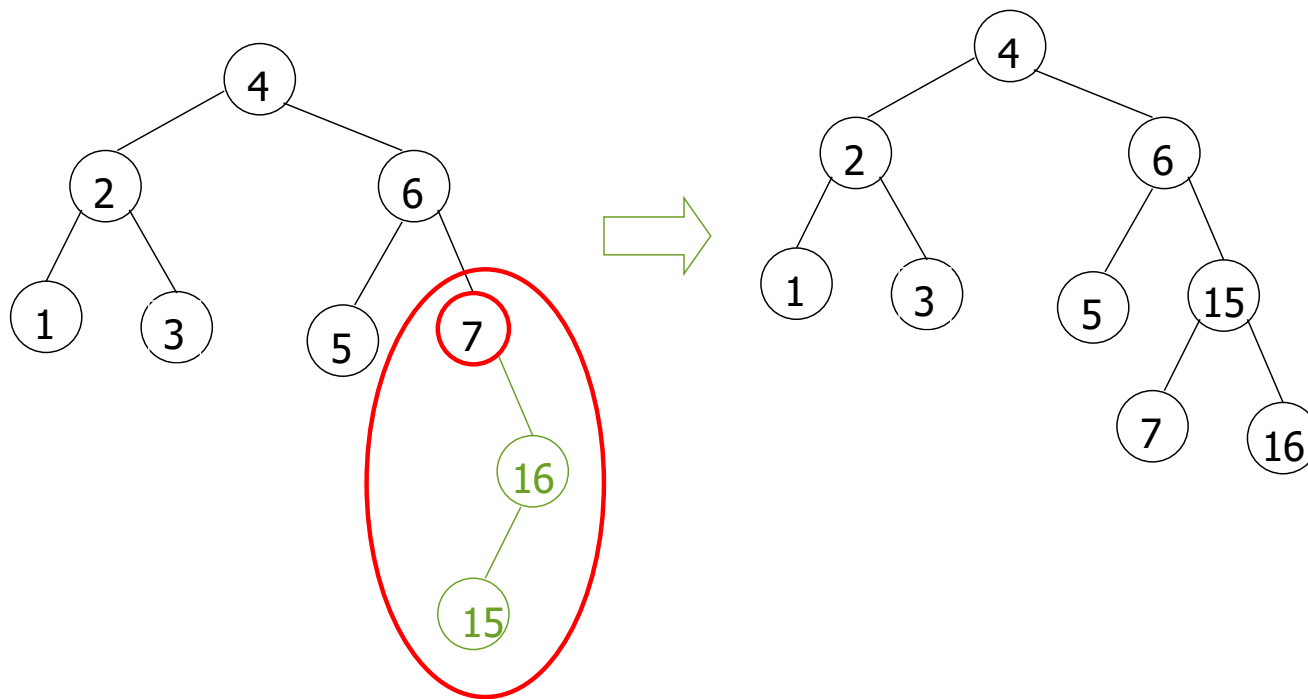


Before Insertion

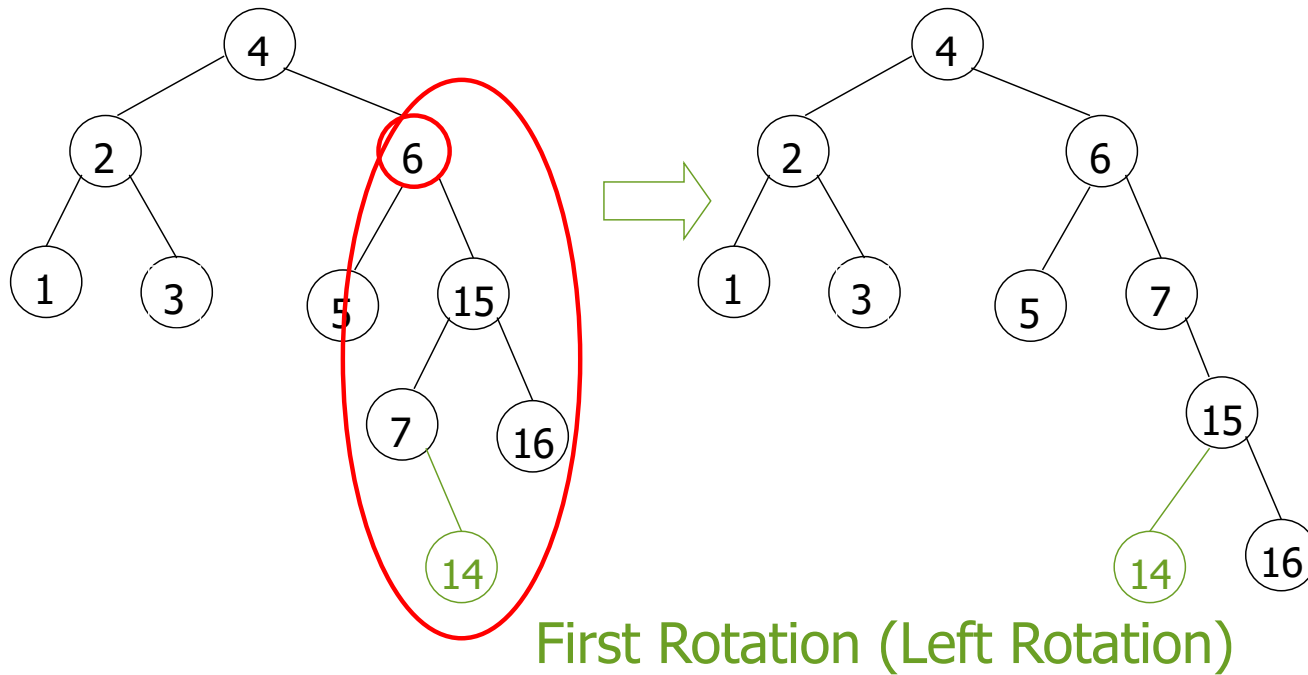


h+1 After Insertion

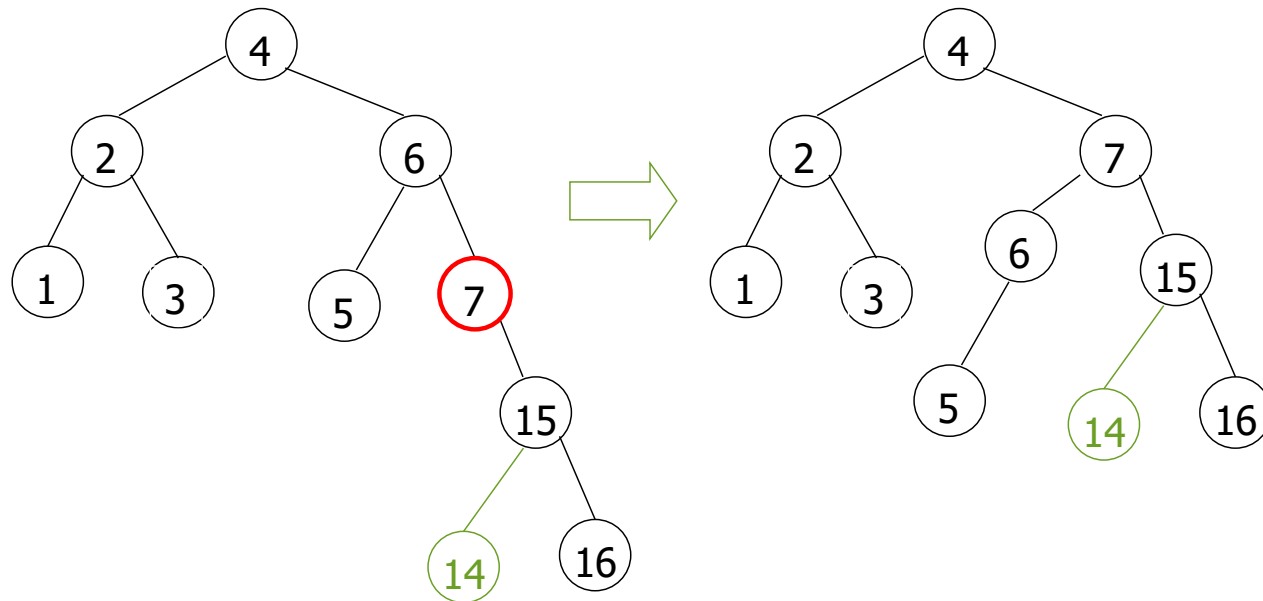
EXAMPLES (INSERT 16, 15)



EXAMPLES (INSERT 14)

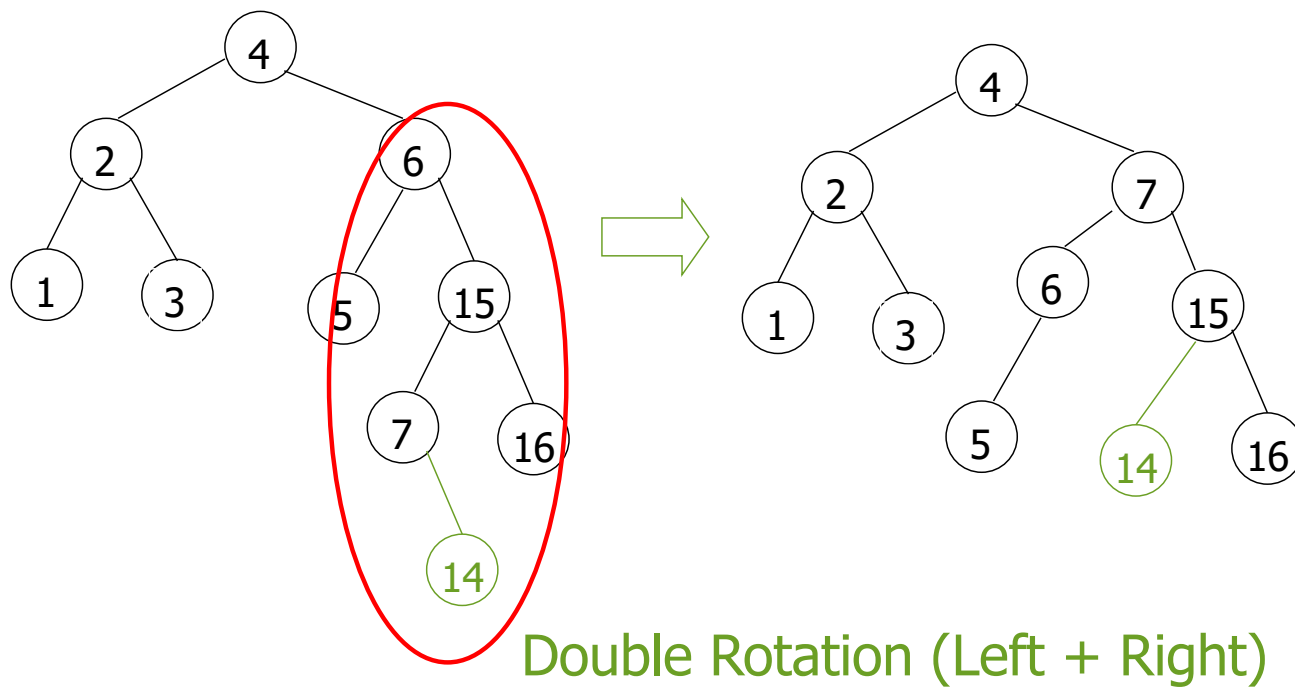


EXAMPLES (INSERT 14)



Second Rotation (Right Rotation)

EXAMPLES (INSERT 14)



SUMMARY: ROTATIONS

After inserting a new node with item X into an AVL Tree T , a height imbalance can appear in T .

- Left-Left or Right-Right Case
 - Single Rotation
- Left-Right or Right-Left Case
 - Double Rotation

Rotations are expensive and can occur frequently with insertions.

Can we get $O(\log N)$ complexity without balancing the trees ?

SPLAY TREES

Splay Trees: No bad input

Any M consecutive tree operations take $O(M \log N)$ operations.

However any **single** operation can take $O(N)$

Amortized Cost: When a sequence of M operations takes worst case time $O(M \log N)$

- then amortized (average) time is $O(M \log N) / M = O(\log N)$

Main Idea: $O(N)$ operations not bad if they occur infrequently

BASIC IDEAS

A single operation could take $\Theta(n)$ time.

A sequence of “Finds” to deep nodes causes problems.

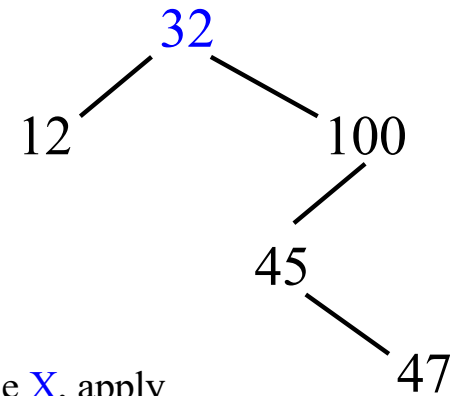
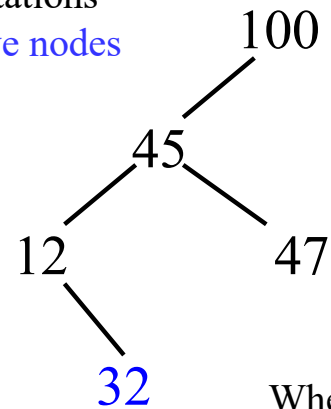
We change the tree structure after a (bad) operation.

- Self-adjusting

SPLAY TREE

Idea: Self Organizing Binary Search Tree

Use rotations
of **move nodes**
to root



When we access a node **X**, apply
a rotation to move the node **X**
closer to root

Repeat until **X** is the root

FIRST IDEA

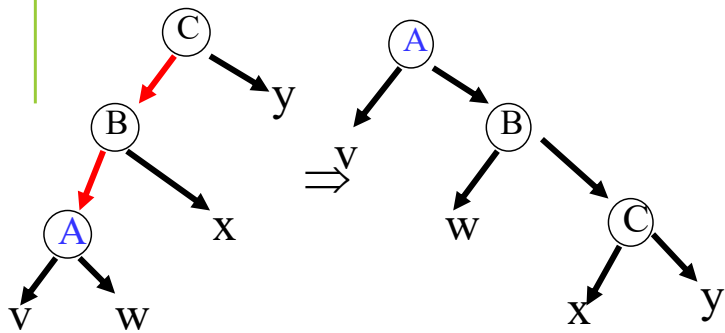
Bring frequently accessed node closer to the root

For each access bring the accessed node to the root

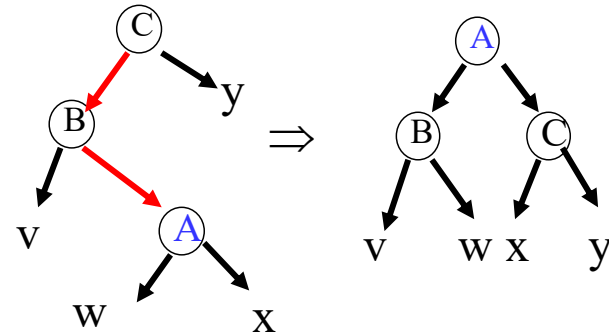
Perform single rotations until it is the root

Problem: Tree remains unbalanced

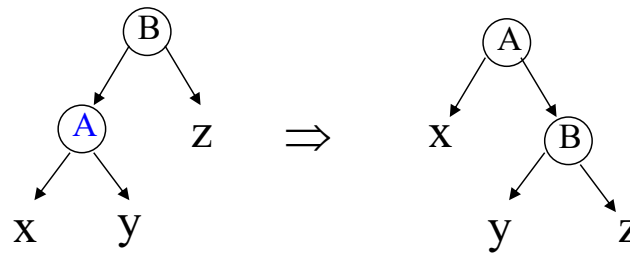
SPLAYING AT NODE **A**



1. One side



2. zig-zag



3. No grandparent

INSERTION

Insert new node as leaf in correct position

Splay new node until it becomes the root

EXAMPLE: INSERTING NODE 1

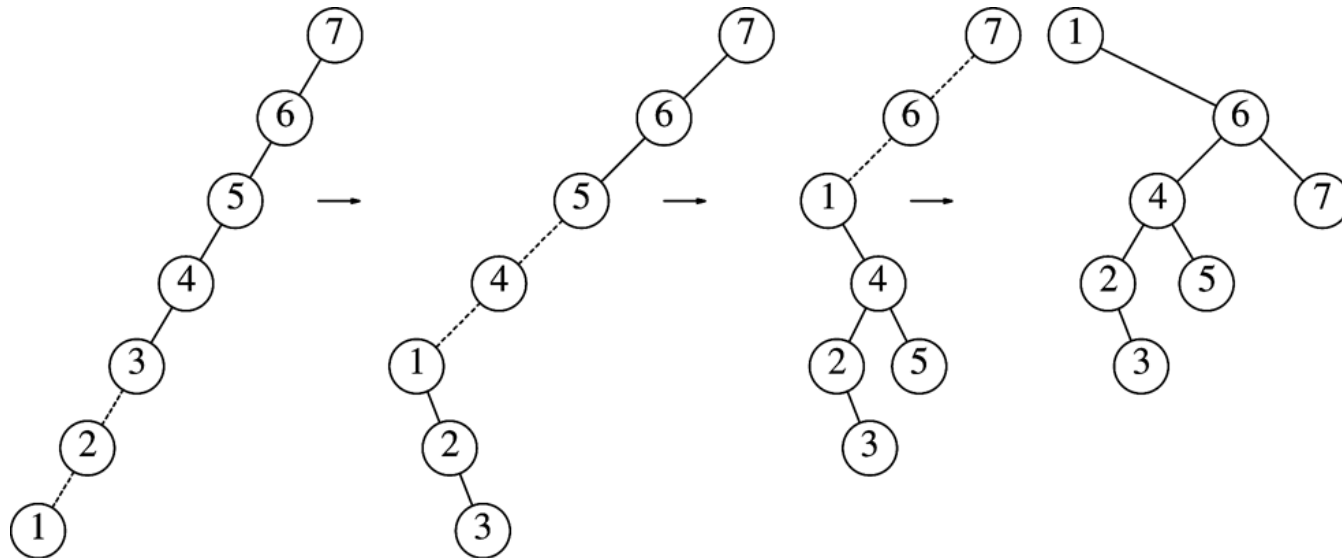
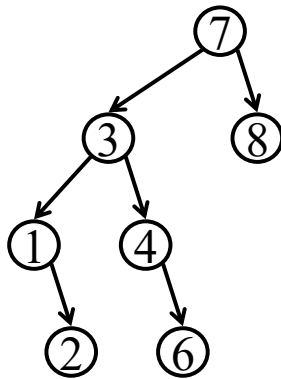


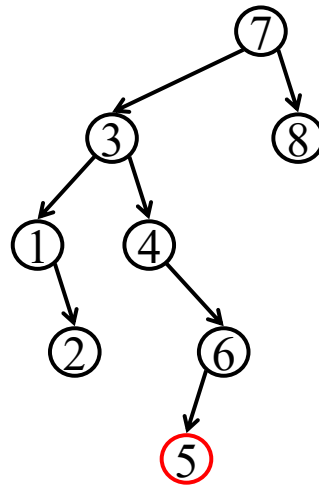
Figure 4.49: Result of splaying at node 1

For more Examples: <https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>

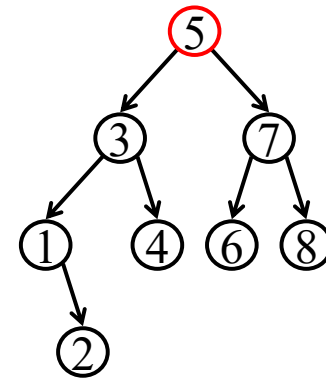
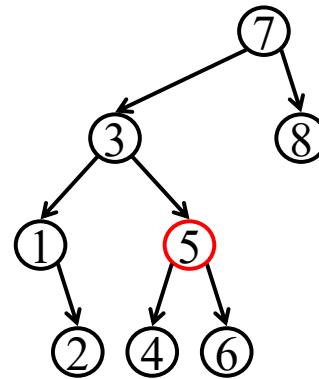
INSERT 5



Insert



Splay



DELETE

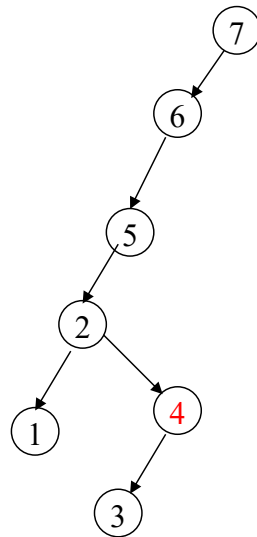
Depends on how often deletions are...

Delete X

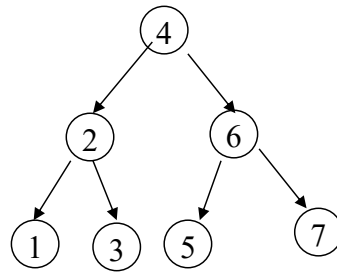
- 1. First Splay X
- 2. Splay the largest node A in the left subtree upto root of left subtree
- 3. Delete X
- 4. Make A new Root

DELETE

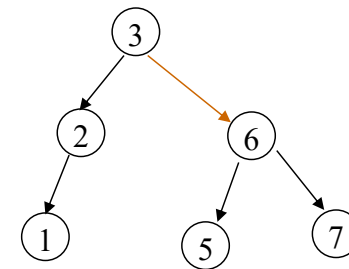
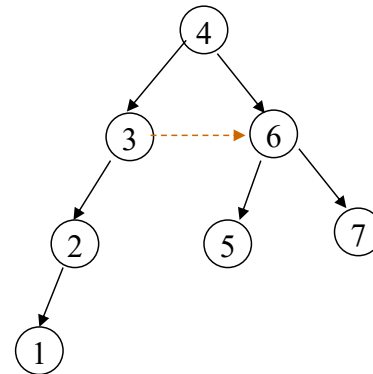
delete(4)
Start



After splay(4,T)



After splay(3)



After delete 4 and reset left link of 3

AMORTIZED COST FOR SPLAY TREE

For a splay tree, the worst case cost for a sequence of M operations starting from an empty tree is bounded by $O(M \log N)$

- Thus, amortized cost per operation is $\log N$
- Worst case running time for an operation is $O(N)$

Cost of $O(N)$ for an operation is not bad for many applications if this happens seldom. In binary search tree a sequence of worst case operations can happen

MEMORY ACCESS

Data Locality can affect performance time

Data is accessed from hard drive via cache

Data in cache can be accessed faster....

- Line of 64 bytes in 2.5 ns

...than Data in main memory

- First word (4 bytes) 50 ns and then 5ns for consecutive words

Unless data is carefully arranged, memory access will dominate running time.

USING BINARY SEARCH TREE

Binary search Trees

- The worst case: linear time 10,000,000 disk accesses.
- In a typical randomly constructed tree, a search would require about 100 disk accesses.

AVL Trees

- The worst case of $1.44 \log N$ (≈ 35 disk accesses,)
- The typical case using an AVL tree is very close to $\log 10,000,000 \approx 24$ disk accesses

HOW TO REDUCE THE NUMBER OF DISK ACCESSSES?

Consider each height as a disk access

Reduce the number of disk accesses

We cannot go below $\log N$ using a binary search tree (including AVL tree).

- A binary search tree allows only one key per node.
- The only way to reduce the number of disk accesses is to increase the number of keys in a node.

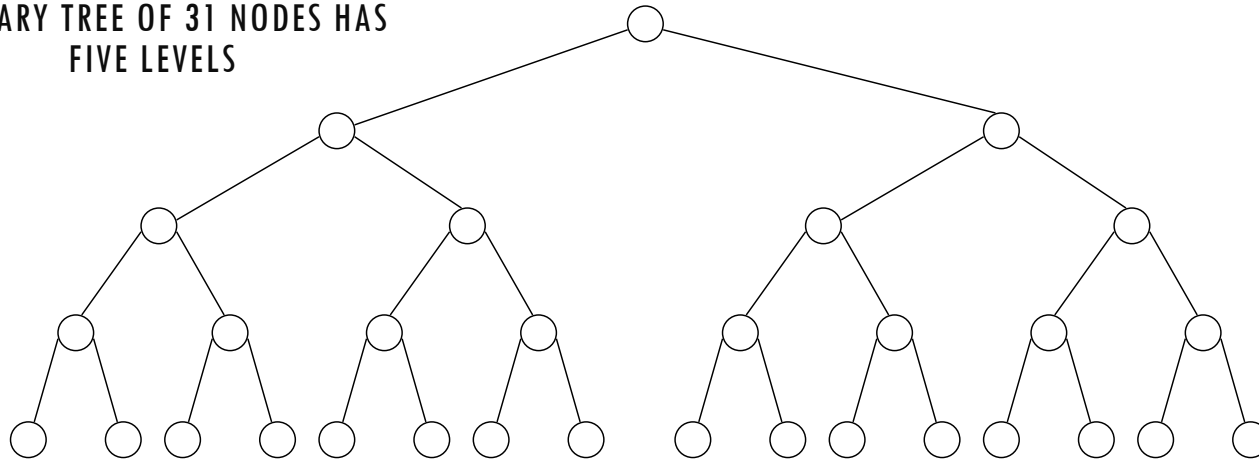
MORE BRANCHING=LESS HEIGHT

An M -ary search tree allows M -way branching.

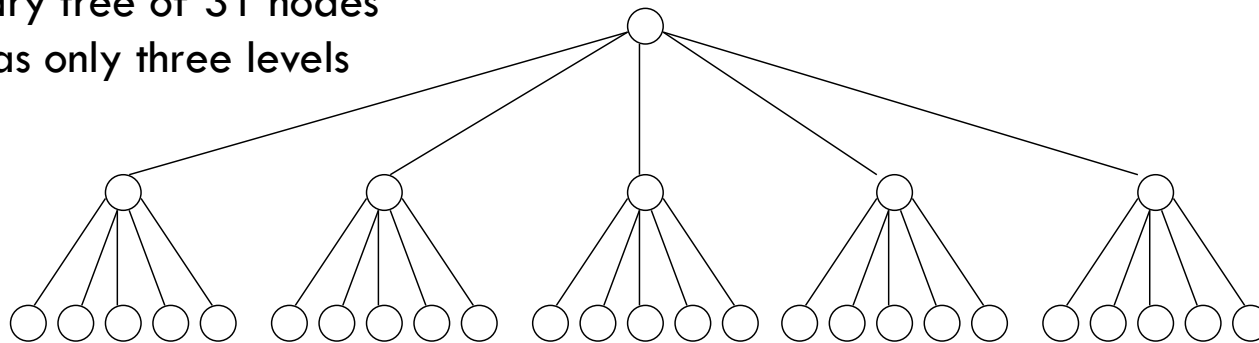
A perfect M -ary tree is an M -ary search tree with all the leaf nodes at the same depth.

A complete M -ary tree has height that is roughly $\log_M N$

**A BINARY TREE OF 31 NODES HAS
FIVE LEVELS**



**5-ary tree of 31 nodes
has only three levels**

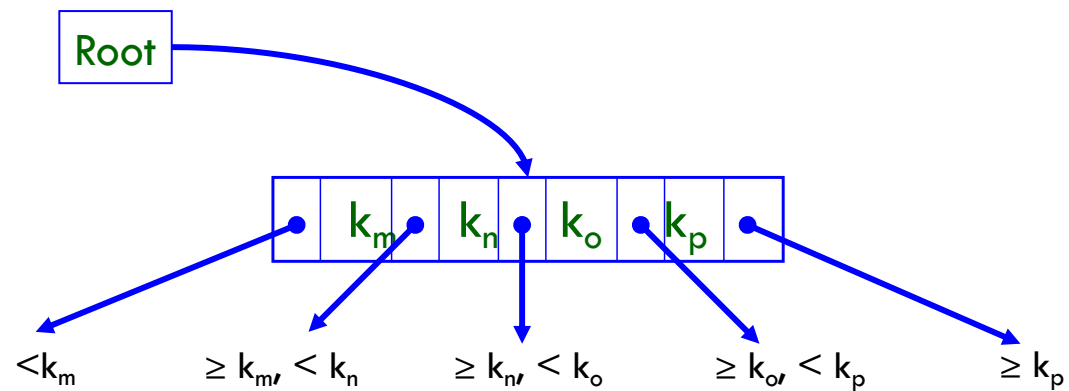


M-ARY SEARCH TREE

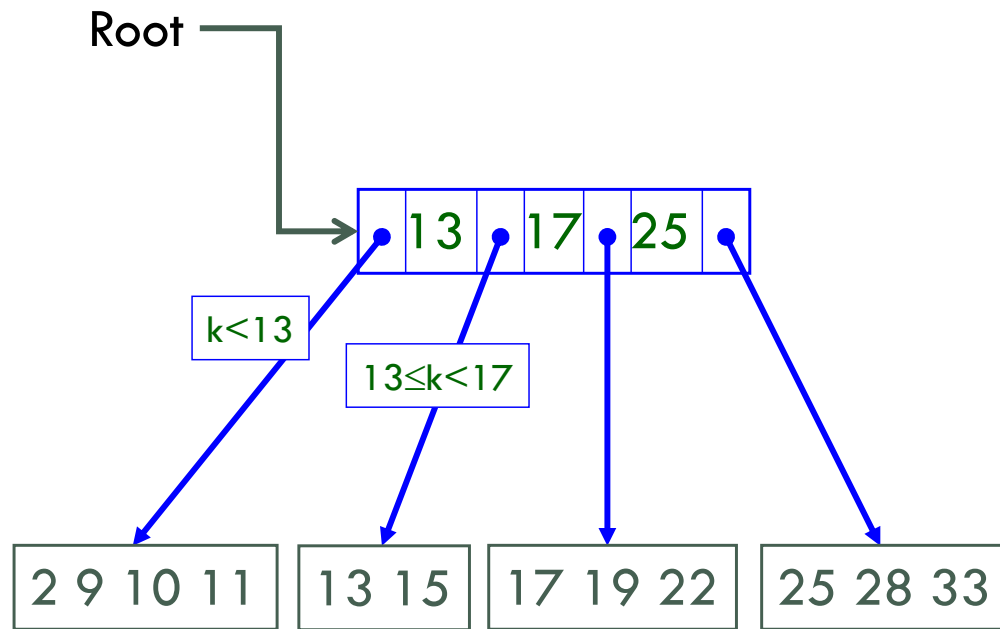
In an M -ary search tree, we need $M-1$ keys to decide which branch to take.

Should be balanced in some way.

We do not want an M -ary search tree to degenerate to even a binary search tree.



Example

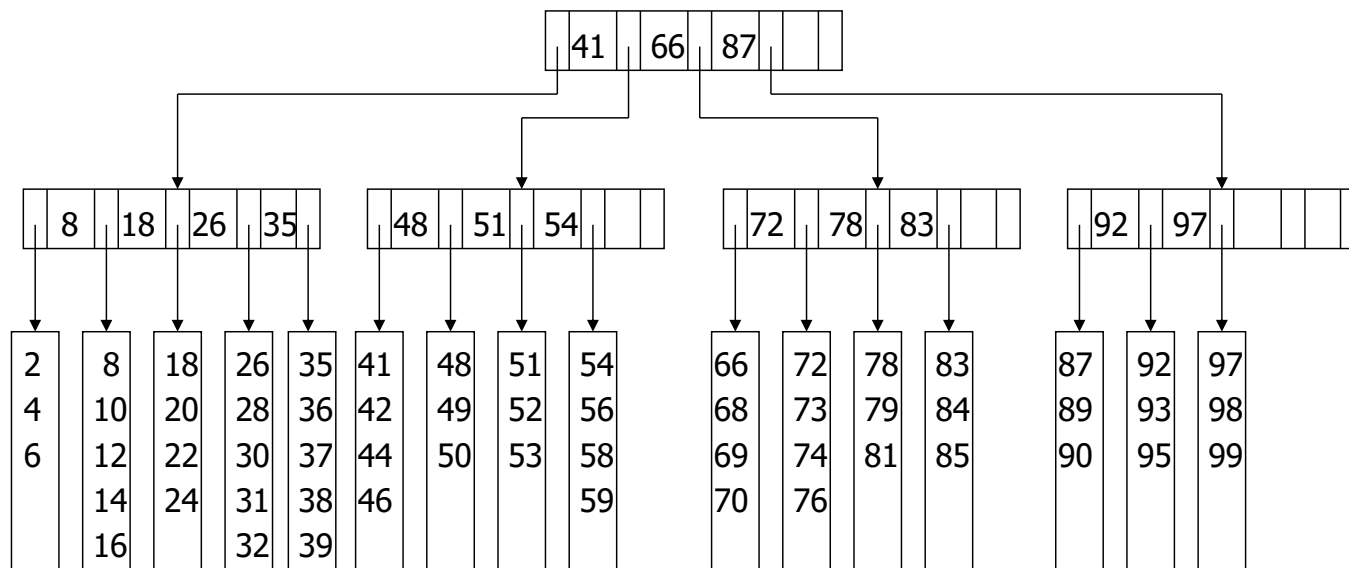


B-TREE

A B-tree of order M is an M -ary tree with the following properties:

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M-1$ keys to guide the searching; key i represents the smallest key in subtree $i+1$.
3. The **root** is either a leaf or has between two and M children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L children, for some L .

B-TREE OF ORDER 5

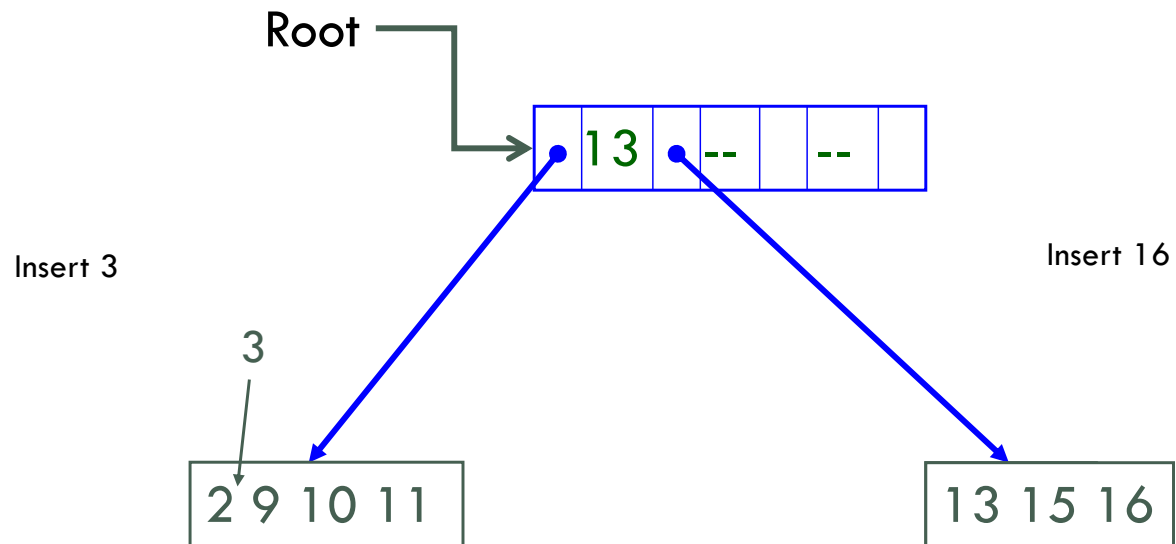


Leaf nodes are in sorted two-way linked list

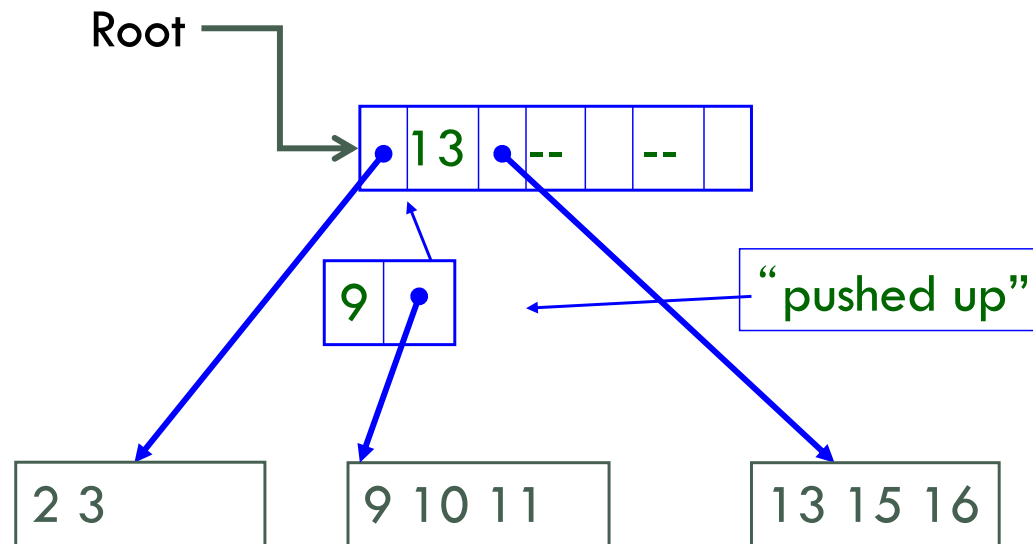
Insertion

- If empty we create one node that is a root node and insert there - otherwise -
- Find the node the key belongs in
 - Insert if enough room - otherwise -
 - Split in half into two nodes
 - Insert new key (median value from node that was split) in parent recursively

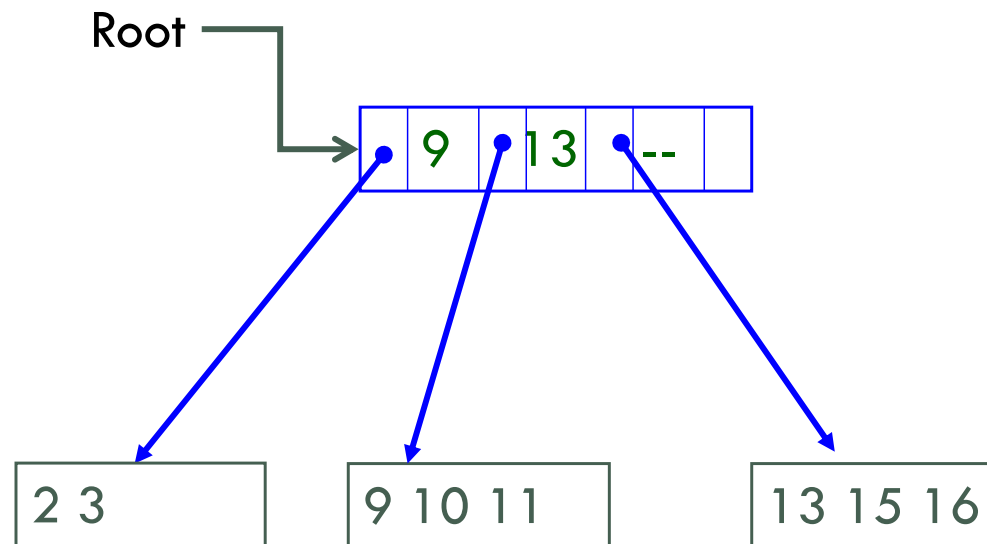
Example - Insert 16 & 3



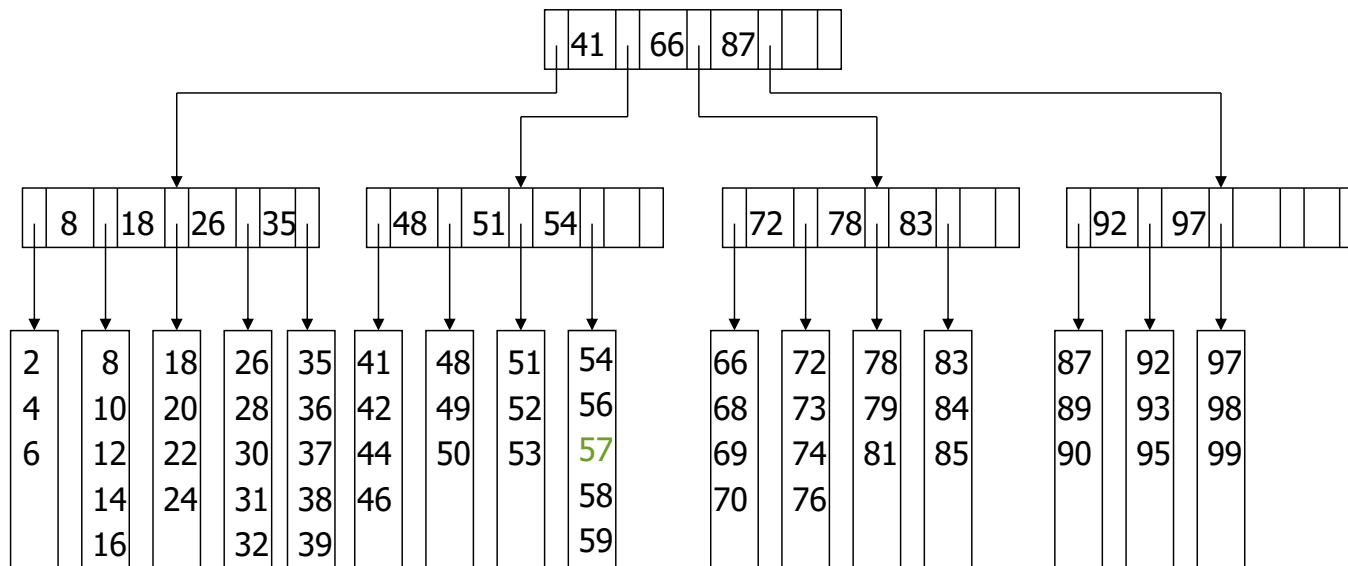
Split a node



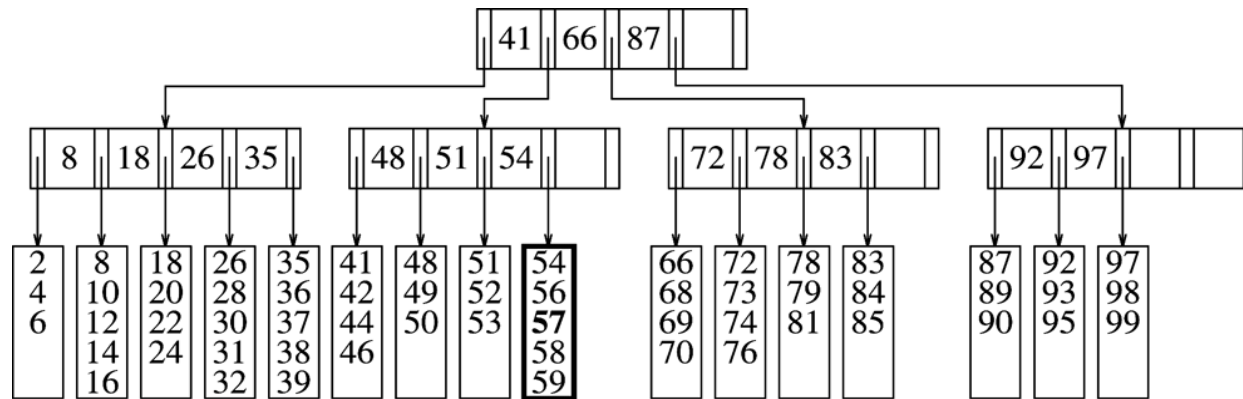
Insert continued



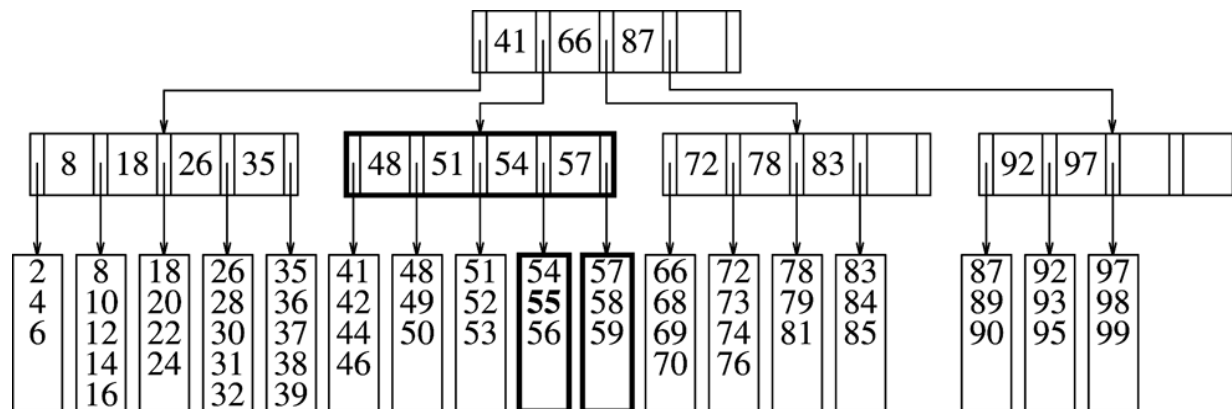
ADD ITEM 57 TO THE B-TREE



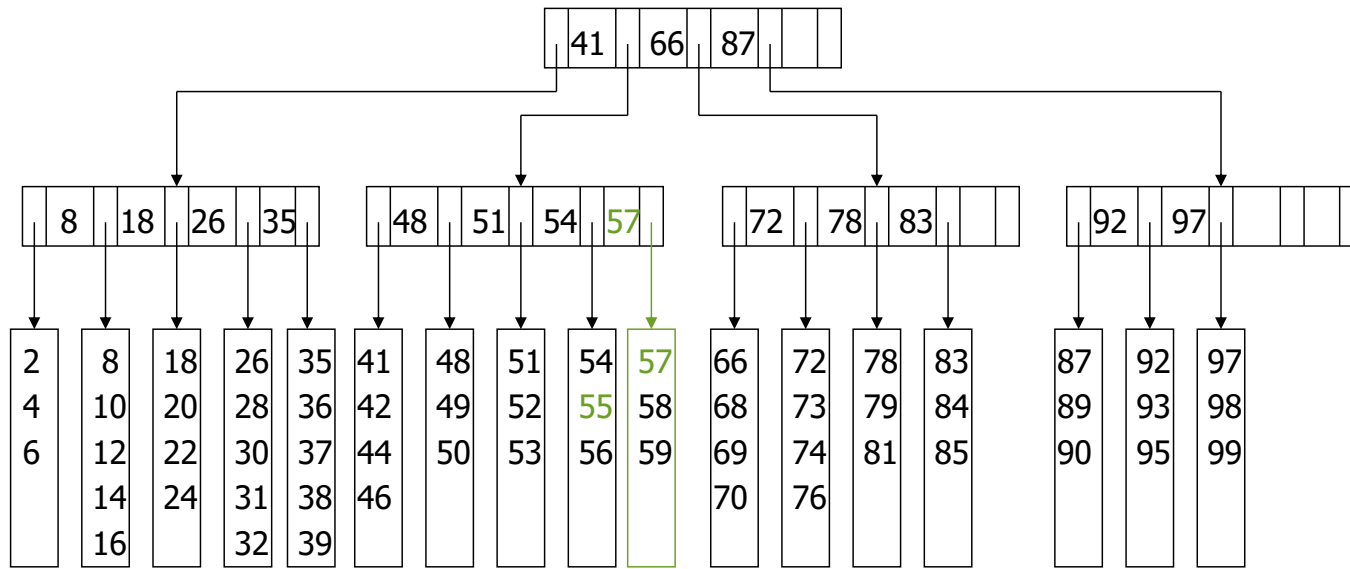
Add item 55

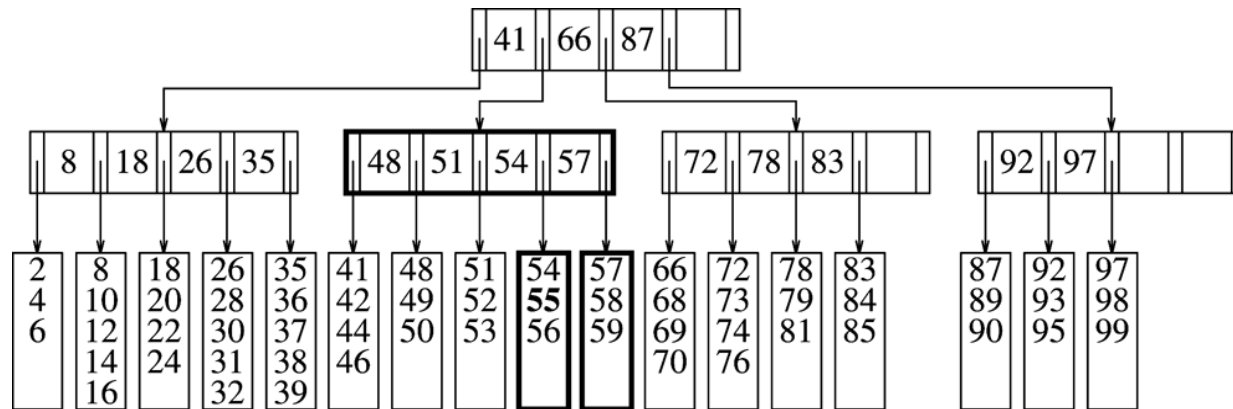


Add item 55

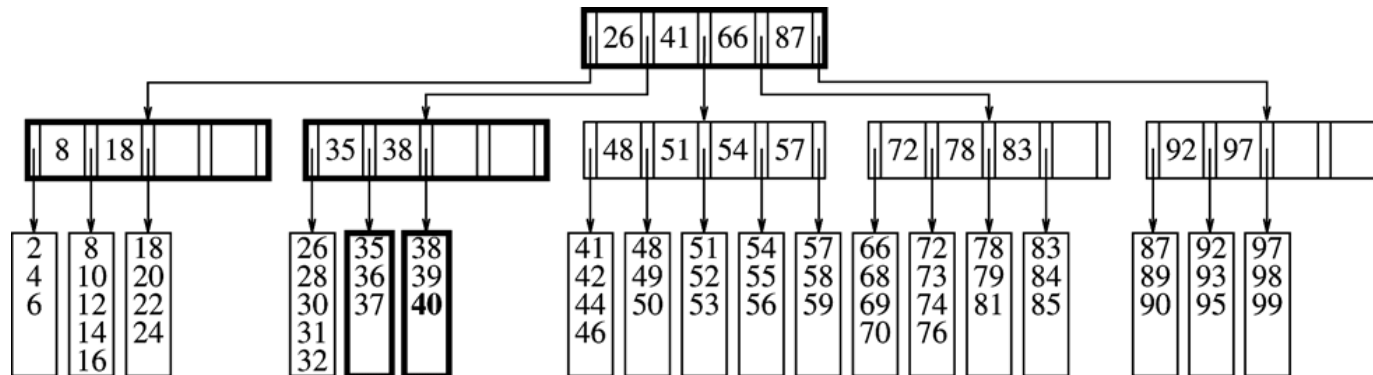


EXERCISE: ADD 40





Add item 40



REMOVE AN ITEM FROM B-TREE

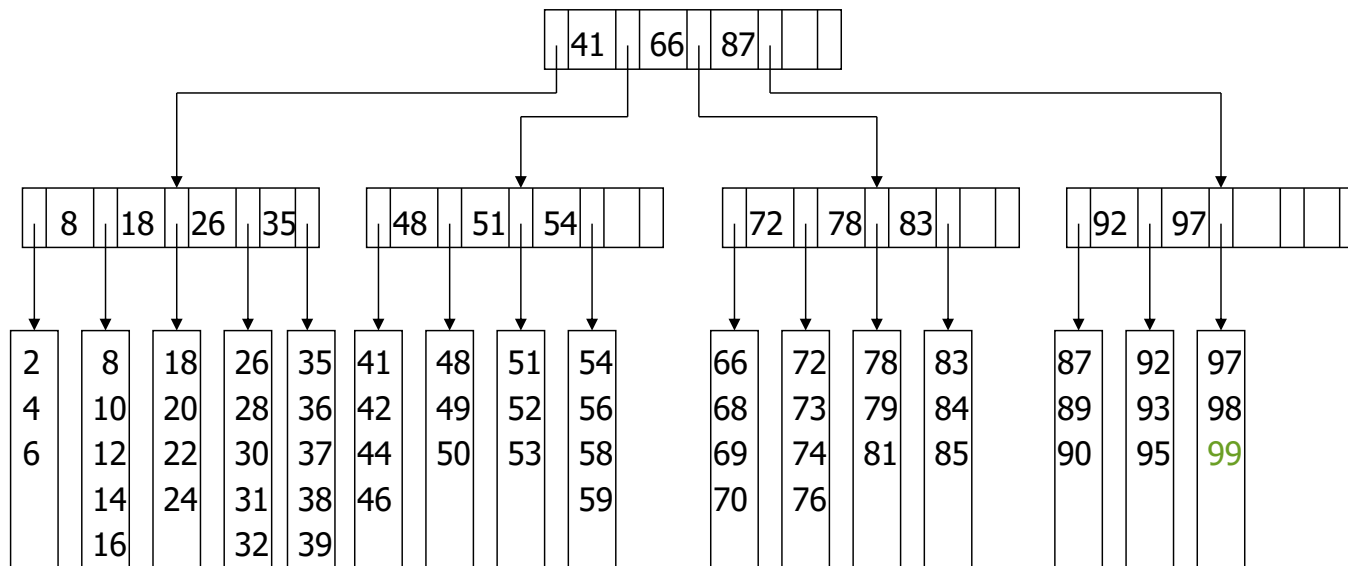
What if the leaf it was in has the minimum number of data items?

Adopting a neighboring item if the neighbor is not itself at its minimum.

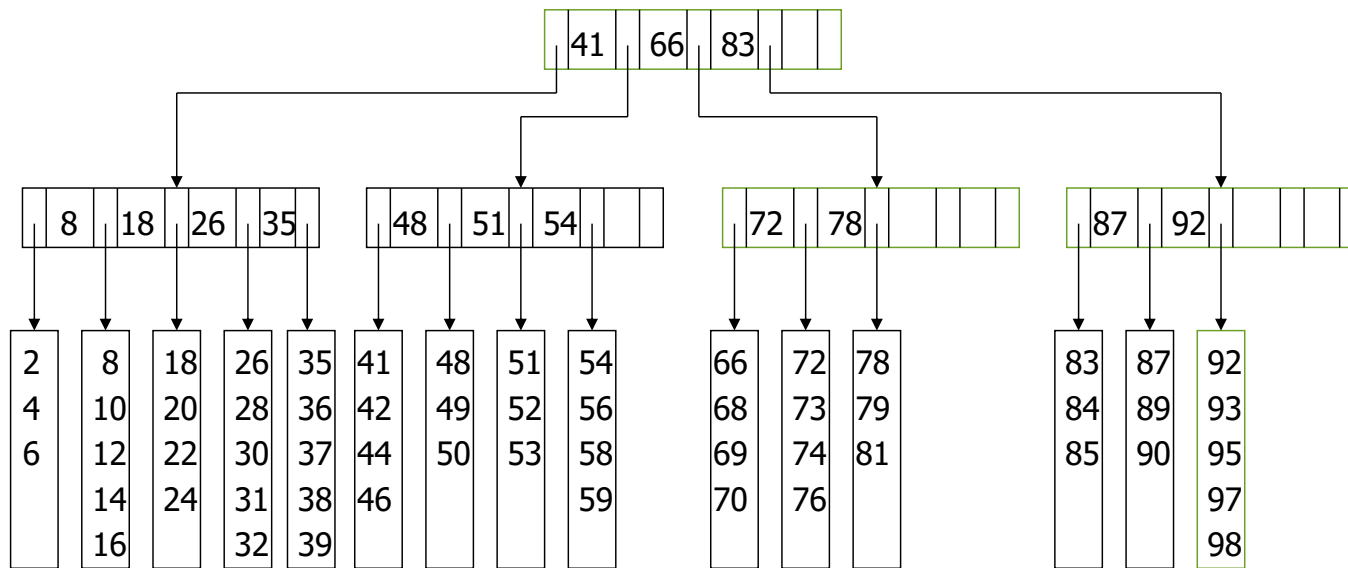
If neighbor at minimum combine with the neighbor to form a full leaf.

This means that the parent has lost a child. If parent to fall below its minimum, follow the same strategy.

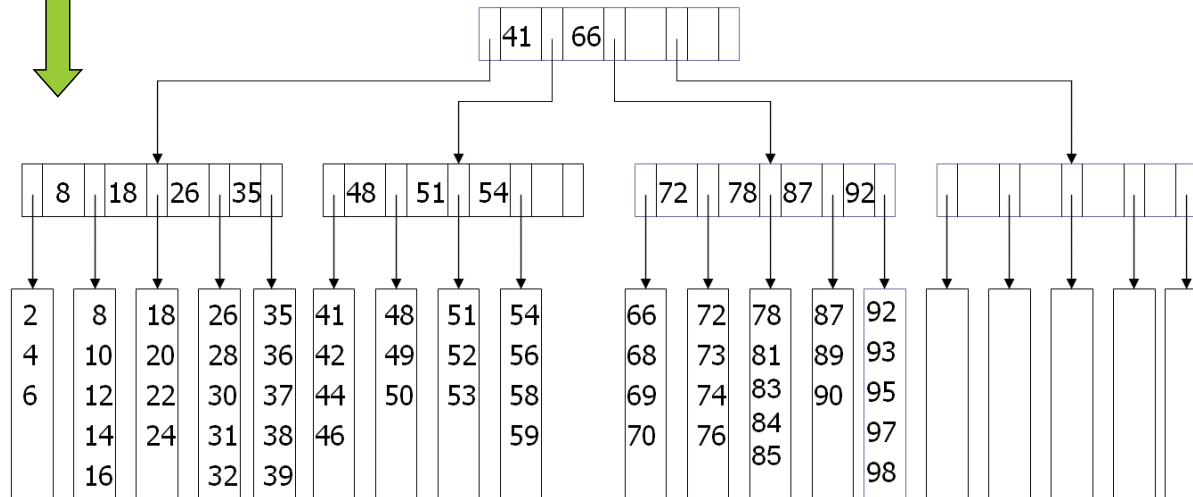
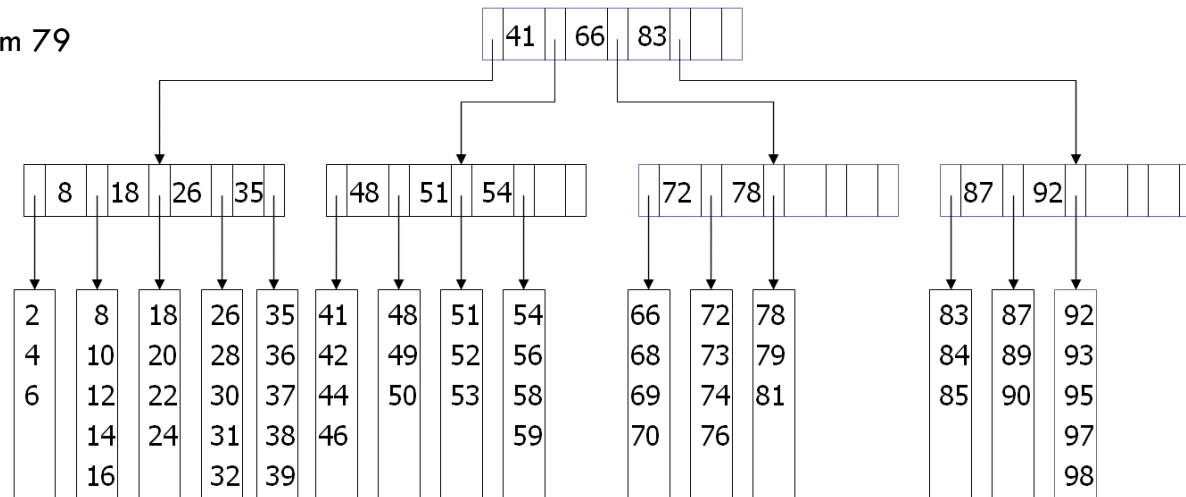
REMOVE ITEM 99 FROM THE B-TREE



AFTER REMOVING 99 FROM THE B-TREE



Exercise: Remove item 79



228-ARY B-TREE

228-Ary B-tree $M=228$

The size of the leaf-node is 32 records. $L=32$

Each leaf has between 16 and 32 data records, so there are, at most, $10,000,000/16 \approx 625,000$ leaves.

Each internal node (except the root) branches at least 114 ways.

In the worst case, leaves would be on level 4

Complete M-ary tree has height $\log_M N$

Average height = $\log_{M/2} N$

USING B-TREE

Each node represents a disk block.

Choose M and L based on the size of the items that are being stored.

Suppose one block holds 8,192 bytes.

In our example, each key uses 32 bytes.

Let each branch link is 4 bytes.

Total memory requirement for a nonleaf node is thus $(32(M-1) + 4M)$.

What is the maximum value of M ?

Each record is 256 bytes, what is the largest value of L ?

SUMMARY FOR B-TREES

Binary Search Trees

- Complexity $O(\log N)$ if balanced
- Balancing too often is expensive

Splay Trees

- Amortized cost is $O(M \log N)$
- Worst case $O(N)$

B-Trees

- B-trees are balanced M -way (as opposed to 2-way or binary) trees, which are well suited for disks.
- Tree height increases at top, not bottom
- Tree is inherently balanced.
- More complicated to implement