CS 234

Module 6

October 16, 2018

Idea for an ADT

The **ADT Dictionary** stores pairs (key, element), where keys are distinct and elements can be any data.

Notes:

- This is similar, but not identical, to the Python data type dictionary.
- This is similar to the textbook's ADT Map. ADT Map is restricted to the case where keys are orderable.
- This is inspired by, but not identical to, a language dictionary, where keys are words and elements are etymology, pronunciation, and definitions.
- The term Dictionary is used throughout the literature.

Restrictions to consider:

- Keys are orderable.

ADT Dictionary

Data: (key, element pairs) where

- keys are distinct but not necessarily orderable, and
- elements are any data.

Preconditions: For all D is a dictionary, k is a key, and e is an element.

Postconditions: Mutation by Add (add/replace item with key k) and Delete (delete item with key k, if any).

Name	Returns
Create()	a new empty dictionary
IsEmpty(D)	true if empty, else false
LookUp(D, k)	item with key k if any, else false
Add(D, k, e)	
Delete(D, k)	

Array and linked implementations

Customizing

Array or linked list, items not ordered

- Add, Delete require LookUp (for add, to see key not already there)
- LookUp $O(n)$ worst case

Special case of orderable data

- Store in increasing order of key
- Costs still depend on LookUp
- Recall binary search from CS 116

Data structure: binary search tree (BST)

A **binary search tree** (or **BST**) is a binary tree that satisfies the **binary search tree property**, that is, for every node n in the tree, for k the key stored in the node:

- The values stored in the left subtree of n are smaller than k.
- The values stored in the right subtree of n are greater than k.

Note: We can generalize this to allow repeated values, using $\leq$ and $>$ or $<$ and $\geq$.

Customizing ADT Binary Tree to implement BST

Additional data: Store both keys and elements at nodes.

Notes:

- Only keys are ordered using the binary search tree property.
- Our illustrations show keys only.

Modified ADT Binary Tree operations:

- AddNode(B, p, k, e, side)

Additional ADT Binary Tree operations:

- KeyAtNode(B, n)
- ElementAtNode(B, n)
- StoreInNode(B, n, k, e) - changes what is stored in a node

Implementing ADT Dictionary operations

LookUp(D, k)

- Compare, choose direction (use LeftChild, RightChild)
- Find element or failure (hits empty pointer)

Add(D, k, e)

- First execute LookUp(D, k)
- If ends in empty (side = left or right) pointer of p, then AddNode(B, p, node, side)

Analysis of operations

- $\Theta(1)$ for each node examined: how many in total?
- Θ(height of tree)
- $\Theta(n)$ in worst case
- If nice tree, worst case is $\Theta(\log n)$
- Lower bound of $\Omega(\log n)$ (in n node tree, some node must be at least log n away from the root)

Implementing Delete(D, k)

Find the node n containing the key

Case 1: n has no children: delete n

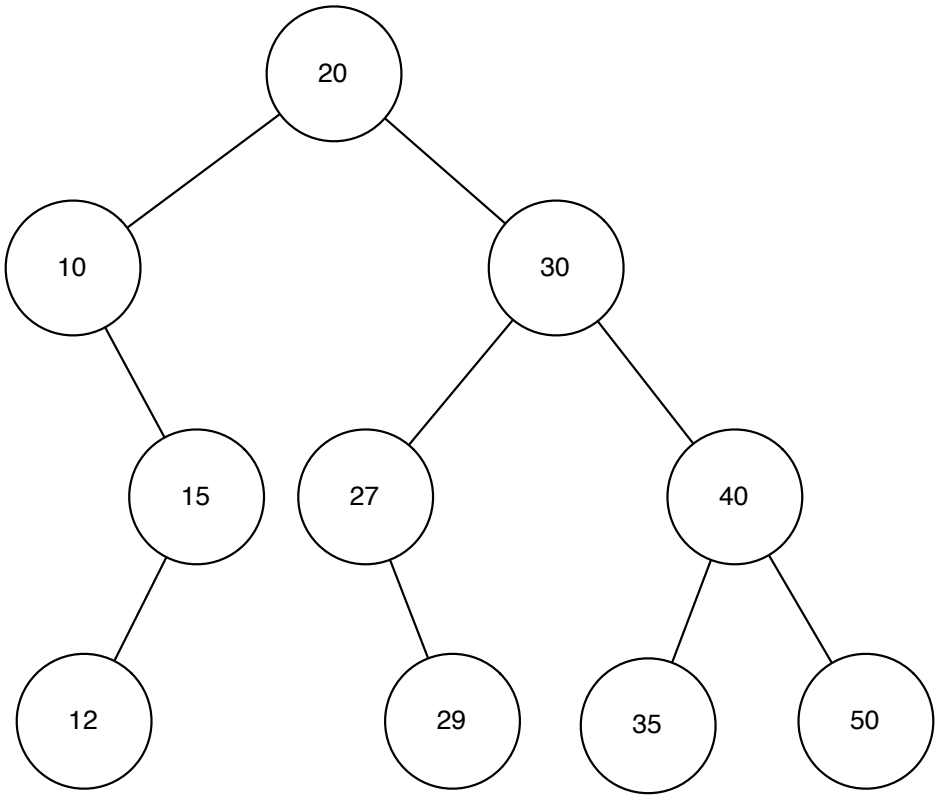
Case 2: n has only one child c: make c into the child of n's parent, then delete n

Case 3: n has two children:

- Find the inorder successor s of n.
- Replace n's (key, element) pair with s's (key, element) pair.
- Delete the node containing n (Case 1 or 2).

Binary search tree deletion, illustrated

Case 1: 12, 29, 35, 50
Case 2: 10, 15, 27
Case 3: 20, then 27 (Case 2)
Case 3: 30, then 35 (Case 1)
Case 3: 40, then 50 (Case 1)



Finding an inorder successor

Guide students:

- Where is it? (Answer, leftmost node in right subtree)
- How can you find it? (Go right, then always to the left)
- Cost of finding it? (Worst case: height of tree)
- Can you use the inorder predecessor? (Yes.)
- Where is it? (Rightmost node in left subtree)
- Will it always have one child? (Yes, same argument, mirror image.)

Height of a binary search tree

Everything depends on the height

- worst case $\Theta(n)$
- best case $\Theta(\log n)$

Goals:

- Maintain $\Theta(\log n)$ height
- Implement $\Theta(\log n)$ worst case time for LookUp(D, k), Add(D, k, e), and Delete(D, k)

Can we maintain the minimum height at all times?

Too costly to maintain, too inflexible.

Idea: Find "close enough" to minimum to work.

Data structure: AVL tree

Key ideas:

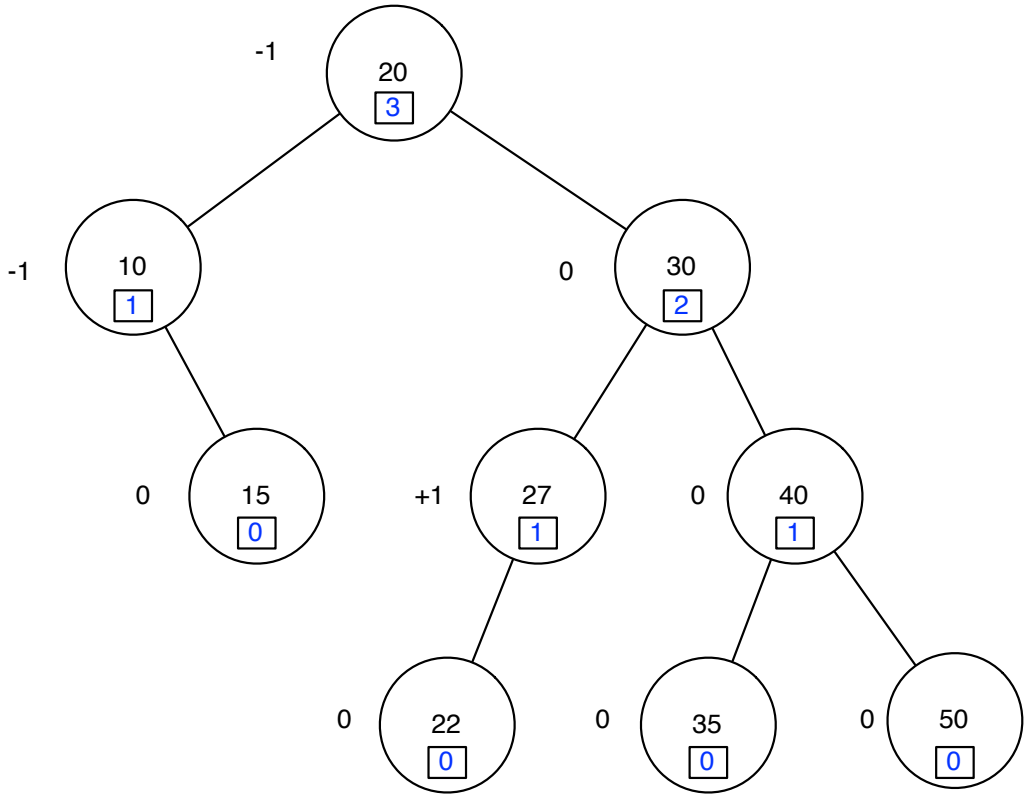
- A node is **balanced** if the difference in height of left and right subtrees is at most 1.
- A tree satisfies the **height-balance property** if every node is balanced.
- An **AVL tree** is a height-balanced BST.
- To determine balance, store the height of each node.
- For calculations, consider an absent subtree to have height -1.

Maintain balance and running times as follows:

- Any AVL tree with n nodes has height $\Theta(\log n)$. (Proof omitted.)
- Add preserves the height-balance property, worst case in $\Theta(\log n)$.
- Deletion preserves the height-balance property, worst case in $\Theta(\log n)$.

AVL tree example

Heights are stored in nodes.
Balance: 0 if left and right subtrees have the same height, 1 if the left subtree is one higher, -1 if the right subtree is one higher



Customizing ADT Binary Tree to implement AVL tree

Since an AVL tree is a special type of BST, make all the same modifications as needed to implement a BST using the ADT Binary Tree, and then also the ones below.

Additional ADT Binary Tree operations:

- Height(B, n) - looks up height at node n
- ReplaceHeight(B, n, h) - resets the height at node n to h

Customize each data structure for ADT Binary Tree:

- Add a field for each node storing the height of the node.

Implementing Add(D, k, e) in an AVL tree

The **pivot node** is the lowest unbalanced node in the tree after a new node has been inserted.
A **rotation on the pivot node** is a rearrangement of subtrees that rebalances the tree without violating binary search order.

Algorithm:

- Find leaf to insert as in BST.
- Trace path from leaf to root, updating heights and checking balance.
- The first imbalanced node encountered (if any) is the pivot node.
- Rebalance by executing a rotation on the pivot node.

Observations:

- A rotation entails updating a constant number of pointers.
- At most one rotation is needed to rebalance the tree.

To check:

- Binary search order is preserved
- Height balance property is preserved.

Rebalancing

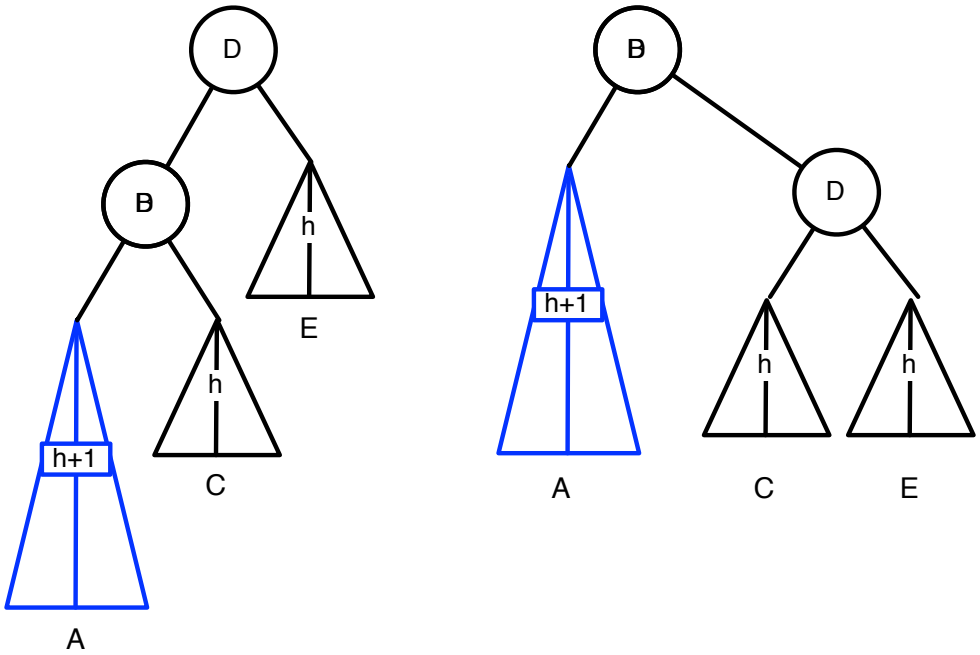
Possible cases for imbalance at a pivot node:

1. Left child's left subtree is too high
2. Left child's right subtree is too high
3. Right child's right subtree is too high (symmetrical to 1)
4. Right child's left subtree is too high (symmetrical to 2)

In all upcoming slides:

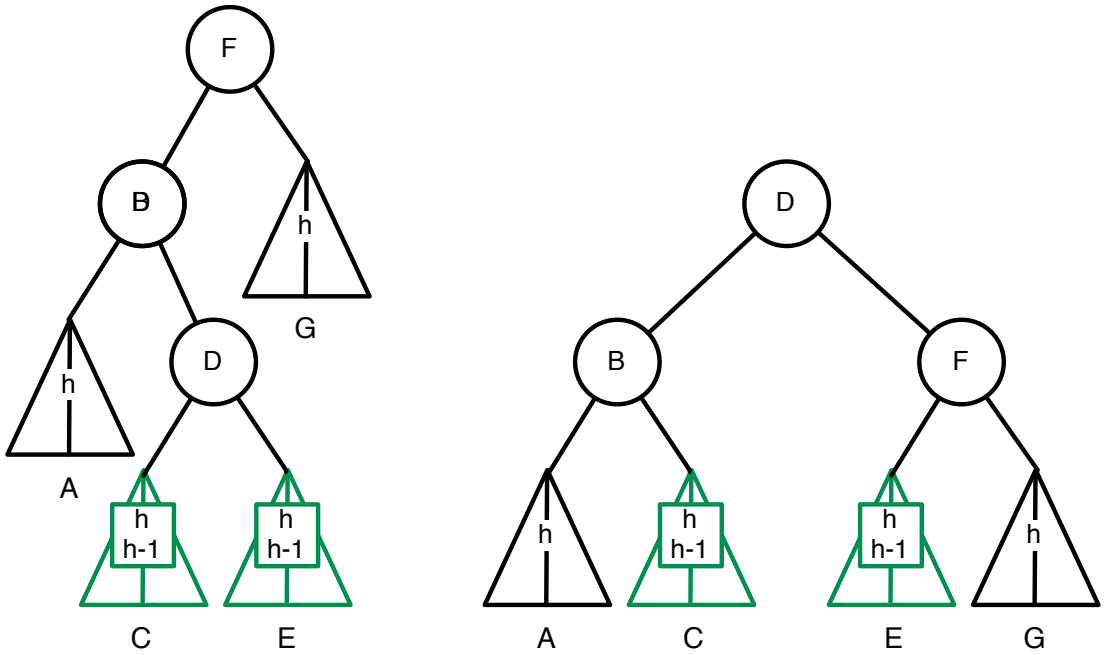
- Letter labels show order of values.
- Left image shows the portion of the tree rooted at the pivot node before rotation.
- Right image shows the replacement subtree after rotation.
- The comments explain why the heights in the left image must be the way they are.

Implementing Add(D, k, e), Case 1



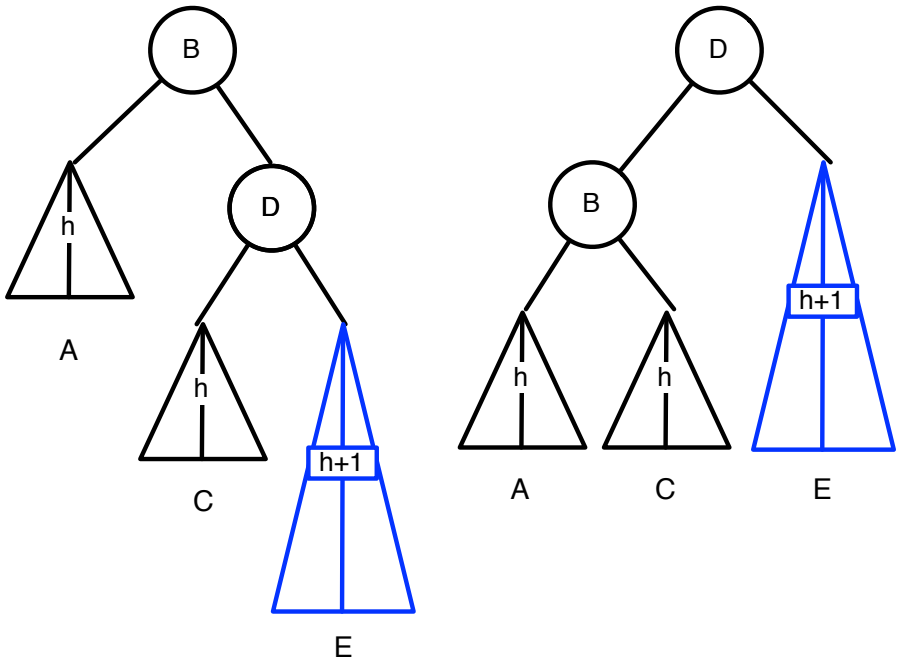
1. New node added in A $\Rightarrow$ height of A increased to make D the pivot node.
2. E is height h + D is pivot node $\Rightarrow$ subtree rooted at B is height $h+2$.
3. (1) + (2) $\Rightarrow$ A is height $h+1$.
4. B is balanced $\Rightarrow$ C is height h.

Implementing Add(D, k, e), Case 2



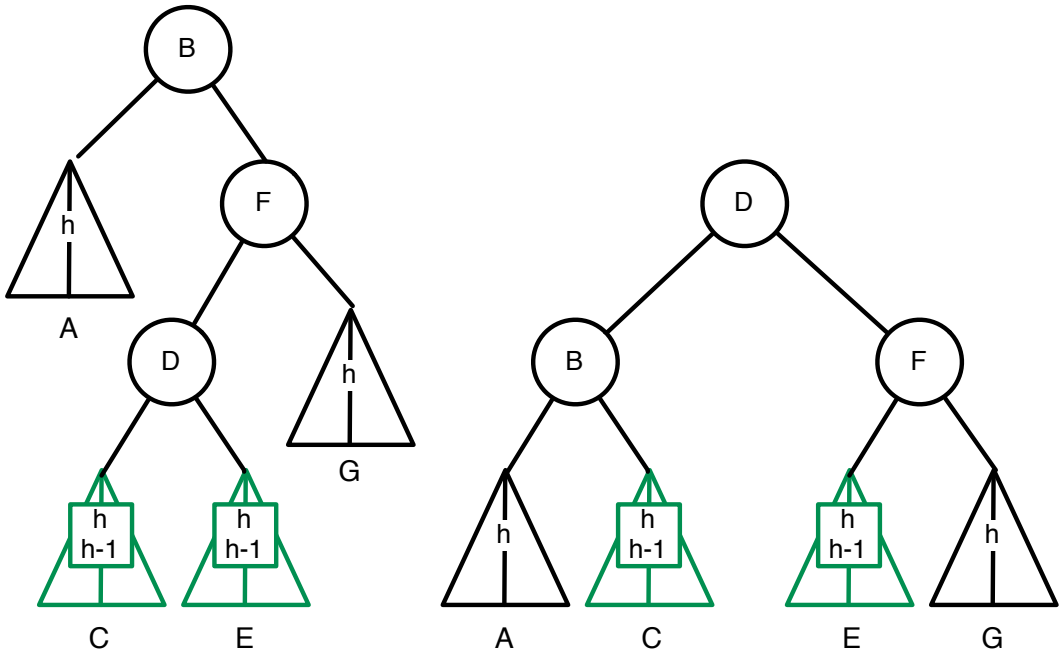
1. New node added in right subtree of B $\Rightarrow$ height of either C or E increased to make F the pivot node.
2. G is height h + F is pivot node $\Rightarrow$ subtree rooted at B is height $h+2$.
3. (1) + (2) $\Rightarrow$ exactly one of C and E is height h.
4. (3) + D is balanced $\Rightarrow$ exactly one of C and E is height $h-1$.
5. B is balanced $\Rightarrow$ A is height h.

Implementing Add(D, k, e), Case 3



1. New node added in E $\Rightarrow$ height of E increased to make B the pivot node.
2. A is height h + B is pivot node $\Rightarrow$ subtree rooted at D is height $h+2$.
3. (1) + (2) $\Rightarrow$ E is height $h+1$.
4. D is balanced $\Rightarrow$ C is height h.

Implementing Add(D, k, e), Case 4



1. New node added in left subtree of D $\Rightarrow$ height of either C or E increased to make B the pivot node.
2. A is height h + B is pivot node $\Rightarrow$ subtree rooted at F is height $h+2$.
3. (1) + (2) $\Rightarrow$ exactly one of C and E is height h.
4. (3) + D is balanced $\Rightarrow$ exactly one of C and E is height $h-1$.
5. F is balanced $\Rightarrow$ G is height h.

Implementing Delete(D, k) in an AVL tree

Algorithm:

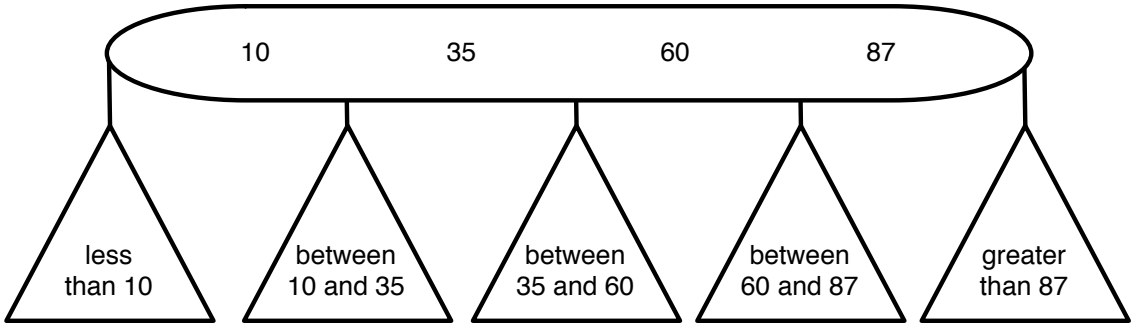
- Delete as in BST.
- Trace path from site of deletion to root, updating heights and checking balance.
- Rebalance by executing a rotation on the first imbalanced node encountered (if any).
- **Continue tracing the path to the root, updating heights and checking balance, pivoting again as often as needed.**

Cost of Delete(D, k):

- Rotation may result in a subtree which is shorter.
- Repeated rotation may be necessary.
- At worst one rotation per level in the tree.
- Worst case $\Theta(\log n)$ rotations at $\Theta(1)$ cost each.

Generalizing BSTs

Idea: To reduce height, why not have nodes with more keys and more children?



Multiway search tree

- Analogy of "perfect" tree now has height $\log_d n$.
- For d a constant, still $\Theta(\log n)$.
- For d not a constant, non-constant processing of a node.

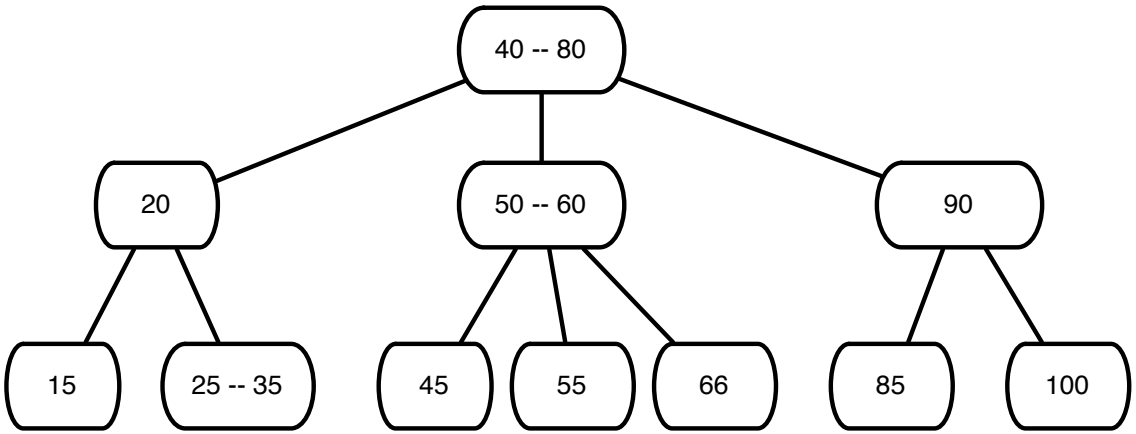
New idea:

- Make all leaves be the same distance from the root, but allow variation in the number of keys per node.
- Slack allows some flexibility and some ease of updates.

Data structure: (2,3) tree

(2,3) tree definition

- Each internal node has either one key and two children or two keys and three children.
- All leaves are at the same depth and have one or two keys.
- Keys in the left subtree are smaller than the first key, in the middle subtree are between the first and second keys, and in the right subtree are greater than the second key.



Any (2,3) tree storing n data items has height $\Theta(\log n)$.

Customizing ADT Ordered Tree to implement BST

Additional data: Store up to two (key, element) pairs and up to three children at nodes.

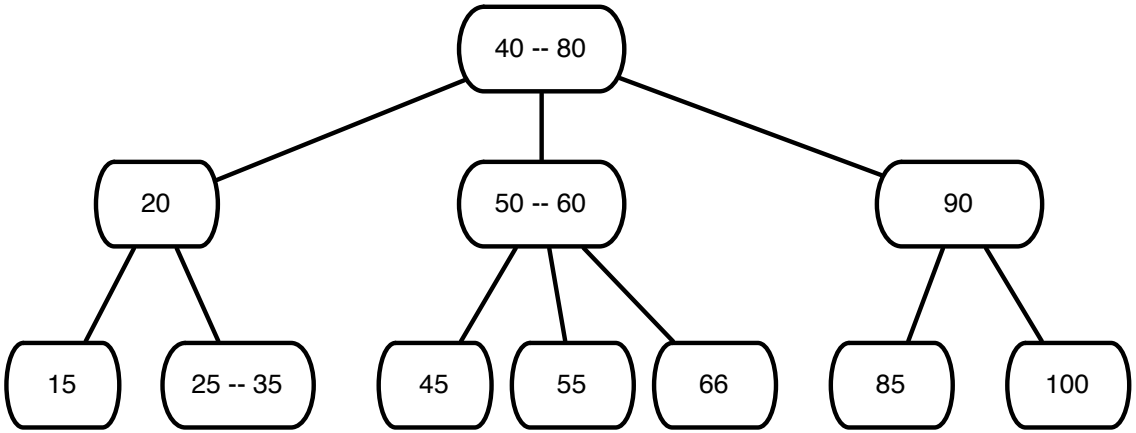
Modified ADT Ordered Tree operations:

- AddNode(O, p, key, element, s)

Additional ADT Ordered Tree operations:

- KeyOneAtNode(O, n)
- ElementOneAtNode(O, n)
- KeyTwoAtNode(O, n)
- ElementTwoAtNode(O, n)
- StoreInNode(O, n, key1, e1, key2, e2)

Implementing LookUp(D, k) in a (2,3) tree



Notes:

- Similar to search in a binary search tree, comparing search key to keys stored in a node and choosing a subtree to search.
- Each unsuccessful search leads to a leaf.
- Worst-case cost is in $\Theta(\log n)$ (height of tree).

Implementing Add(D, k, e,) in a (2,3) tree

If there are too many values in a node, then **overflow** has occurred.

The **split** operation splits a node into two and rearranges values as needed.

Basic idea of operation:

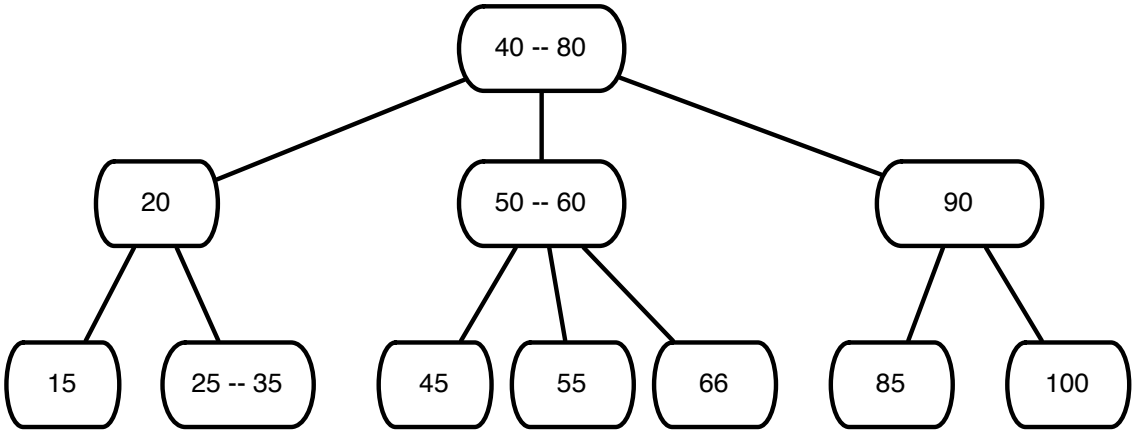
- Search leads to a leaf.
- Add item to leaf.
- If overflow, split leaf.
- If splitting the leaf leads to overflow in the parent, then the parent may need to be split as well.
- Splitting can propagate up to the root.

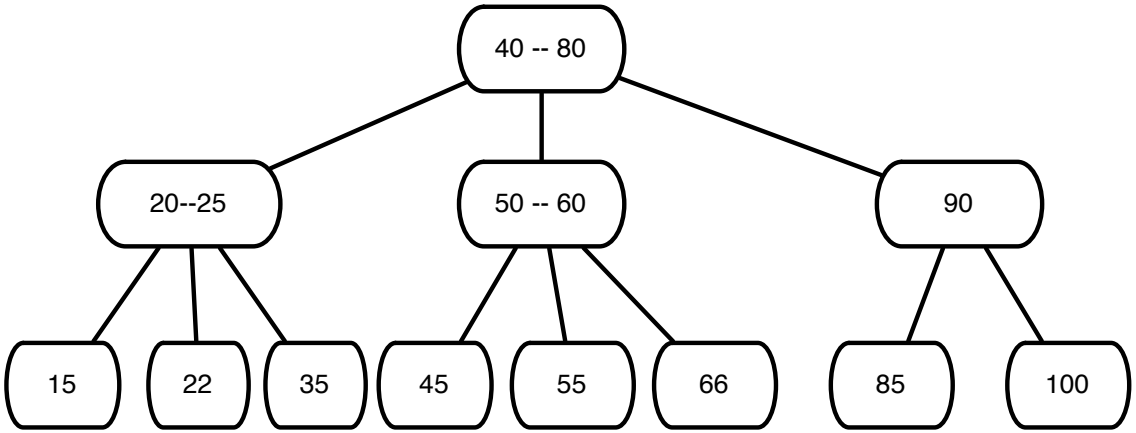
Basic idea of split:

- Node with three keys becomes two nodes, one with smallest key and one with largest key.
- Middle key is sent up to the parent.
- If the node being split was the root, the tree now has a new root.

Implementing Add(D, k, e), single split

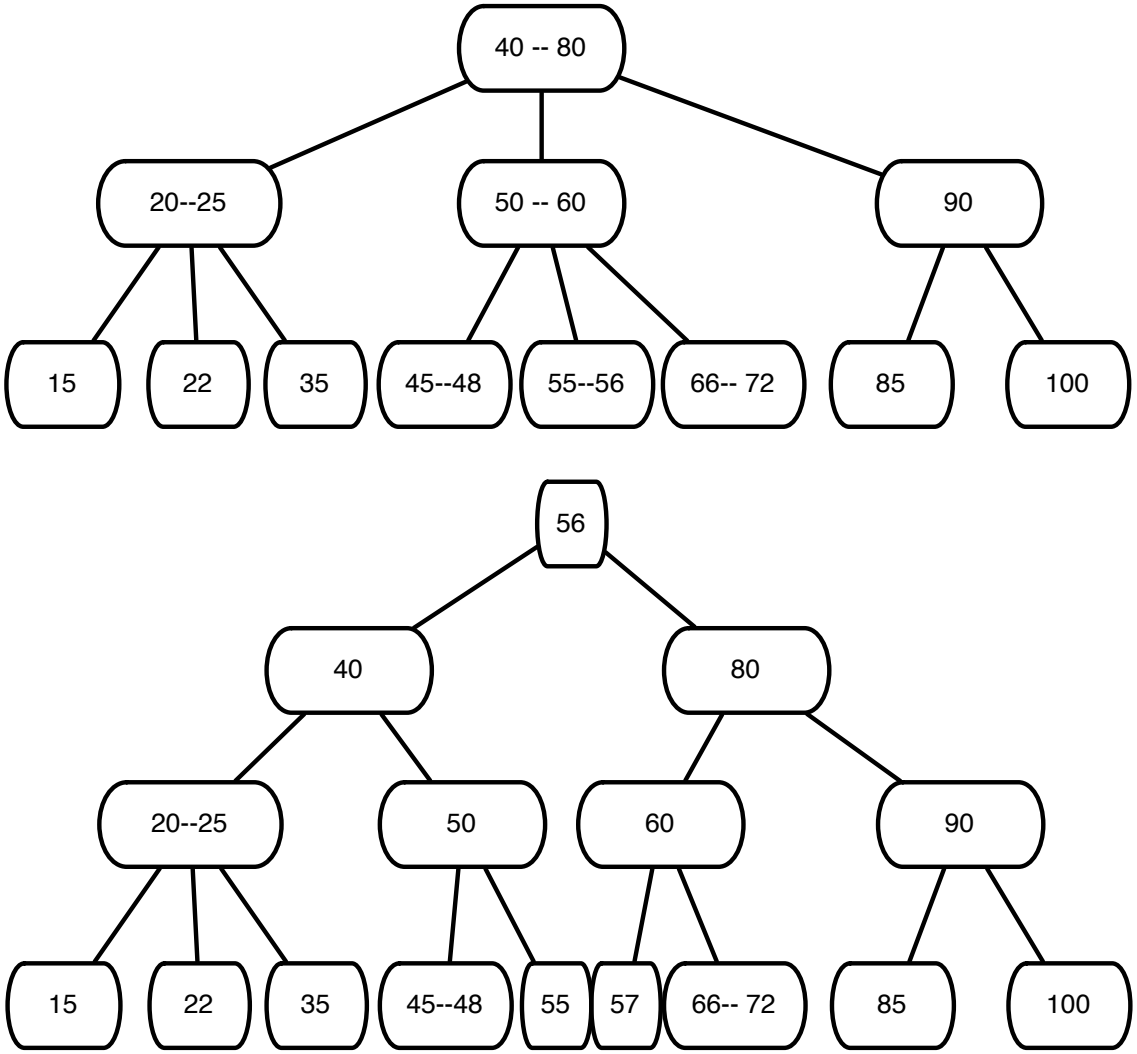
Add 22.





Implementing Add(D, k, e), cascading split

Add 57.



Implementing Delete(D, k) in a (2,3) tree

If there are too few values in a node, then **underflow** has occurred.

The **fuse** operation fuses two nodes into one and rearranges values as needed.

Basic idea of operation:

- Search leads to a leaf or internal node.
- If internal node, like in BST swap with inorder successor.
- Remove item from leaf.
- If underflow, need to fuse nodes.
- If fusing leads to underflow in the parent, then the parent and its sibling may need to be fused as well.
- Fusing can propagate up to the root.

More data structures for ADT Dictionary

Customizing trees:

- Use internal nodes for search; store all values in the leaves.

Other examples include:

- red-black trees
- splay trees
- skip lists (combining tree and list ideas)

A little more information about memory

The **memory hierarchy** consists of different type of memory, where the size of each type increases as the access speed decreases:

- registers
- cache (multiple levels)
- main memory
- secondary storage
- tertiary storage

All you need to know for this course is that there is a hierarchy.
Simplify to two levels:

- **Main memory**
- **External memory** (e.g. disks, cloud)

Impact on data structure design

Moving data between levels:

- Processing takes place in main memory.
- Data to be processed is moved into main memory as it is needed, and moved out to make room for other data to be processed.
- Data is moved between levels in fixed-sized chunks (**pages**).
- There are various **paging algorithms** used to determine which page(s) to move out when a new page is moved in.

Bottom line:

- For small amounts of data, only main memory needs to be considered.
- For large amounts of data, take into account page size.
- The number of accesses to external memory may dominate the cost of an algorithm.

Data structure: B-tree

B-tree of order d:

- Generalization of a (2,3) tree.
- Choose d so that d data items fill a page.
- The root has at most d children.
- Other nodes have at least $\lceil d/2 \rceil$ and at most d children.

Operations:

- $d+1$ children split into nodes with $\lceil (d+1)/2 \rceil$ and $\lfloor (d+1)/2 \rfloor$ items.
- $\lceil d/2 \rceil - 1$ children joined to become a node of size at most d.

Variants:

- B+-trees: all data appears in the leaves, threaded; increase space use by being judicious about splits.
- A B+-tree of order 256 holding 4 million keys uses at most 3 disk accesses.