CS 234: Data Types and Structures

Naomi Nishimura

Module 7

Date of this version: November 5, 2019

WARNING: Drafts of slides are made available prior to lecture for your convenience. After lecture, slides will be updated to reflect material taught. Check the date on this page to make sure you have the correct, updated version.

WARNING: Slides do not include all class material; if you have missed a lecture, make sure to find out from a classmate what material was presented verbally or on the board.

Case study

Problem: Find a way to store student records for a course, with unique IDs for each student, where records can be accessed, added, or deleted.

Recipe for user/plan

1. Determine types of data and operations.
2. For each type, choose/modify/create an ADT.
3. Develop pseudocode algorithm using ADT operations.
4. Calculate cost of algorithm with respect to costs of operations.
5. Using information from provider, choose best option.

Idea for an ADT

The **ADT Dictionary** stores (key, element) pairs, 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 Necaise textbook's ADT Map. ADT Map is restricted to the case where keys are orderable.
- This is inspired by, but not identical to, a dictionary for a language, 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, *Key* is a key, and *Element* is an element; for *DELETE*, *Key* is a key of a pair stored in *D*.

Postconditions: Mutation by *ADD* (adds a data item (*Key*, *Element*), replacing an old data item with key *Key*, if any) and *DELETE* (deletes data item with key *Key*).

Name	Returns
CREATE()	a new empty dictionary
IS_EMPTY(D)	*True* if empty, else *False*
LOOK_UP(D, Key)	element stored with key *Key* if any, else *False*
ADD(D, Key, Element)	
DELETE(D, Key)	

Array and linked implementations

Similar to ADT Set, but:

- Additional operation *Is_Empty*
- *Is_In*, which produces a Boolean, is replaced by *Look_Up*, which produces an element
- Data items are compound data

Options for handling compound data

Main options for compound data include:

- Create an ADT for the compound data to be stored as data items in a contiguous or linked implementation.
- Store each type of data separately (e.g. one array or one linked structure for each type of data).
- Modify a contiguous or linked structure to allow two or more values to be stored in each slot or node.

We will typically use the third option.

Note: For clarity, in many data structures we will only show the keys; you can assume that elements are also stored using one of these methods.

Special case: Orderable keys

New data structures:

- Sorted array: Data items are stored in increasing order by key.
- Sorted linked list: Data items are stored in increasing order by key.

Modifications:

- Shuffling data items left or right in an array.
- Splicing linked nodes in or out of a linked list.

Key idea: Shuffle data items in contiguous memory to maintain order upon modification.

Searching in sorted data structures

In **successful search** the item is present; in **unsuccessful search** the item is not.

Searching algorithms (n is the number of items stored):

- **Binary search** can be used in a sorted array. The cost is in $\Theta(\log n)$ in the worst case for both successful and unsuccessful search.
- **Linear search** can be used in a sorted linked list. Worst-case is still in $\Theta(n)$, but unsuccessful search can stop as soon as a value larger than the search item is found.

Key idea: Contiguous memory permits immediate access to any slot by **random access**.

Ordered array implementation of ADT Dictionary

Idea: Use order on keys to structure the data; store a (key, element) pair in each slot.

Data structures:

- Array with all pairs stored contiguously in order, starting at index 0
- Variable *First* storing index of first empty slot

Selected algorithms:

- *LOOK_UP*, *ADD*, and *DELETE* use binary search
- *ADD* requires larger data items to be shuffled one to the right to make space for the new item; *First* is then updated
- *DELETE* requires larger data items to be shuffled one to the left to use up the space emptied by the deleted item; *First* is then updated

Ordered linked implementation of ADT Dictionary

Idea: Use order on keys to structure the data; store a (key, element) pair in each node.

Data structures:

- Variable *Head* storing a pointer to a linked list of nodes, each storing a key, an element, and a pointer to the next node in the list where keys are in increasing order

Selected algorithms:

- *LOOK_UP*, *ADD*, and *DELETE* use linear search, keeping track of *Previous* and *Current* pointers
- *ADD* requires another $\Theta(1)$ time to splice in the new node in the right location
- *DELETE* requires another $\Theta(1)$ time to splice out the old node

Searching as a fundamental operation

In many data structures, the cost of other operations depends on the cost of searching.

Types of algorithms and analysis to consider (later):

- An algorithm is **comparison-based** if the only types of actions performed on data items are comparisons ($=$, $\neq$, $<$, $>$, $\leq$, $\geq$).
- It is possible to perform actions other than comparisons on **digital data**.
- Data structures may be **internal** (stored entirely in memory) or **external** (making use of external storage).
- When we know (or can guess) something about the frequency with which we search for various data items, we might consider average-case running time instead of worst-case running time.

Creating an ADT Dictionary filled with n items

Options for creating an ADT Dictionary filled with n items:

- Write an algorithm using existing ADT operations.
- Augment the ADT by adding a new operation.

Using existing ADT operations:

- Use *CREATE* to create an empty ADT.
- Use *ADD* to add each item in turn.
- Cost of adding item i is in $\Theta(i)$.
- Total worst-case cost is in $\Theta(n^2)$.

Improving the cost of adding n items

Using preprocessing:

- Sort the items to be added at a cost of $\Theta(n \log n)$ (to be shown later).
- Use *CREATE* to create an empty ADT, using a data structure in which the cost of adding an item smaller than those stored can be accomplished in time $\Theta(1)$ (e.g. sorted linked list).
- Use *ADD* on each item in decreasing order for a cost in $\Theta(n)$.
- Total worst-case cost is in $\Theta(n \log n)$.

Modifying the implementation for the augmented ADT:

- Using sorting for preprocessing.
- Sort data items as part of the implementation of the operation.

Sorting as a fundamental operation

Sorting and ADTs:

- User view: Use ADT operations to sort data.
- Provider view: Sort data by direct access to data structure.

A sorting algorithm is **stable** if equal-valued items are in the same order before and after sorting.

Revisiting comparison-based sorting algorithms

Selection sort

Sort by repeatedly extracting the smallest remaining value (CS 116).

User view: Like repeatedly using an ADT operation that removes the smallest data item.

Insertion sort

Sort by repeatedly inserting items into a sorted list (CS 116).

Provider view: Like adding to a sorted array or linked list data structure.

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 *Node* in the tree, for *Key* the key stored in the node:

- The key values stored in the left subtree of *Node* are smaller than *Key*.
- The key values stored in the right subtree of *Node* are greater than *Key*.

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

Augmenting the ADT Binary Tree

We will augment the ADT Binary Tree in order to support:

- Storing of both keys and elements at each node
- Access to either key or element

Modified ADT Binary Tree operations:

- *ADD_LEAF(B, Par, Side, Key, Element)*
- *SET_VALUE(B, Node, Key, Element)*

New operations:

- *MODIFY_LINK(B, Old_Par, Old_Side, New_Par, New_Side, Node)*
- *KEY(B, Node)*
- *ELEMENT(B, Node)*

Notes:

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

Implementing *DELETE*

Find the node *Node* containing *Key*

Case 1: *Node* has no children: delete *Node*

Case 2: *Node* has only one child *Child*: Make *Child* into the child of *Node*'s parent using *MODIFY_LINK*, then delete *Node*

Case 3: *Node* has two children:

- Find the inorder successor *Succ* of *Node* (the node that follows *Node* in an inorder traversal).
- Swap *Node*'s (key, element) pair with *Succ*'s (key, element) pair.
- Delete *Succ* (the node containing *Key*) using Case 1 or 2.

Binary search tree deletion, illustrated

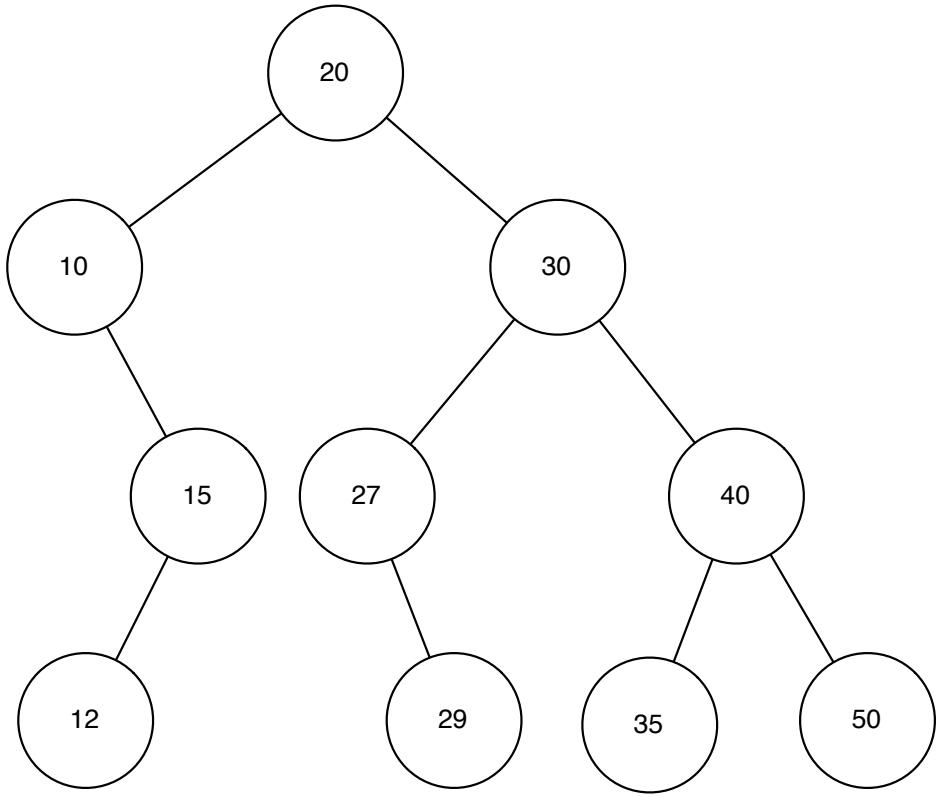
Case 1: 12, 29, 35, 50

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)



Height of a binary search tree

Everything depends on the height (including finding an inorder successor)

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

Goals:

- Maintain $\Theta(\log n)$ height
- Implement $\Theta(\log n)$ worst case time for *LOOK_UP*, *ADD*, and *DELETE*

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, and **imbalanced** otherwise.
- 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.

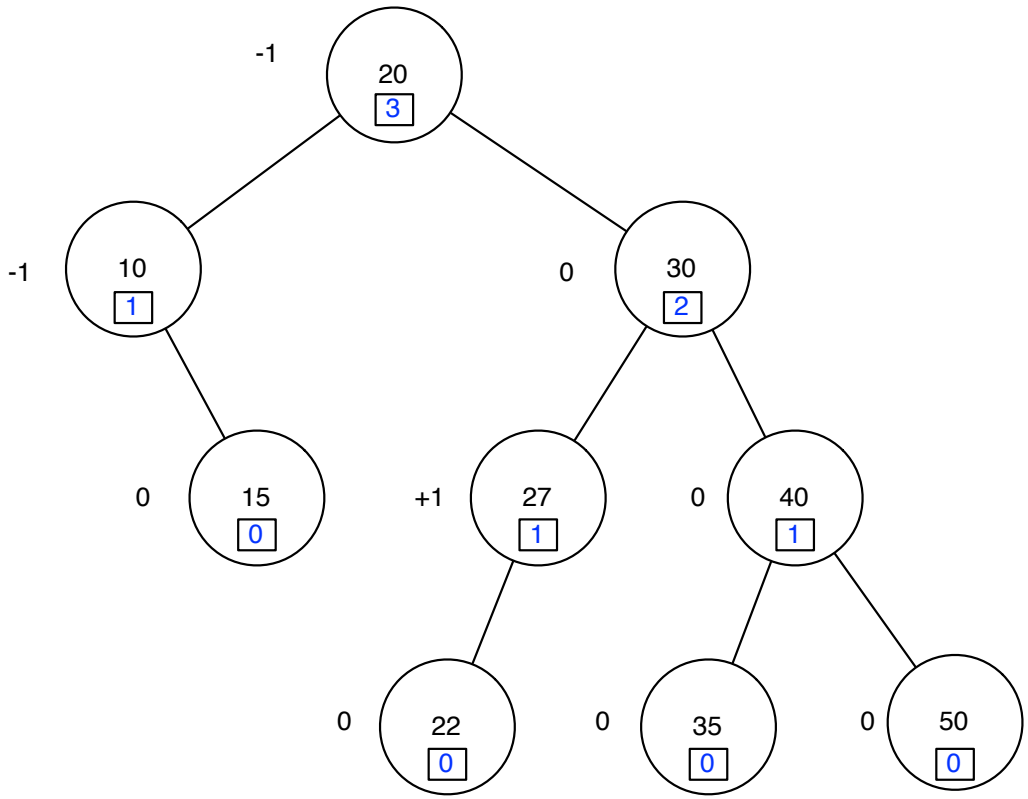
Maintain balance and running times as follows:

- Any AVL tree with n nodes has height $\Theta(\log n)$. (Proof omitted.)
- *ADD* preserves the height-balance property, worst case in $\Theta(\log n)$.
- *DELETE* preserves the height-balance property, worst case in $\Theta(\log n)$.
- *LOOK_UP* is linear in height, 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



Augmenting the ADT Binary 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.

New operations:

- *HEIGHT(B, Node)* - looks up height at node *Node*
- *SET_HEIGHT(B, Node, Height)* - sets the height at *Node* to *Height*

For each data structure for ADT Binary Tree, add a field for each tree node storing the height of the tree node.

Rebalancing a tree

Implementing an *ADD* or a *DELETE* operation like in a BST will preserve the binary seach tree property but might violate the height-balance property.

As part of each operation, rebalance the tree.

The **pivot node** is the lowest imbalanced node in the tree after an insertion or deletion.

A **rotation on the pivot node** is a rearrangement of subtrees that rebalances the tree without violating binary search order.

Implementing *ADD* in an AVL tree

Algorithm:

- Search for the value in the tree as in a BST.
- Insert the new value in a new leaf.
- Visit the path from the leaf to the root, updating heights and checking balance.
- If an imbalanced node is discovered, execute a rotation.

Implementing *DELETE* in an AVL tree

Algorithm:

- Search for the value in the tree as in a BST.
- Delete the node containing the value as in a BST.
- Visit the path from the deleted node to the root, updating heights and checking balance.
- If an imbalanced node is discovered, execute a rotation.
- **Continue tracing the path to the root, updating heights and checking balance, pivoting again as often as needed.**

Imbalanced nodes

Properties of the tree before and after the operation:

- All nodes were balanced before the operation.
- The only nodes that can be imbalanced after the operation are the pivot node and its ancestors.
- Since the operation changed a height by at most one, the balance must have gone from 1 to 2 (left subtree 2 higher) or from -1 to -2 (right subtree 2 higher).

Rotations

Ideas for rotations:

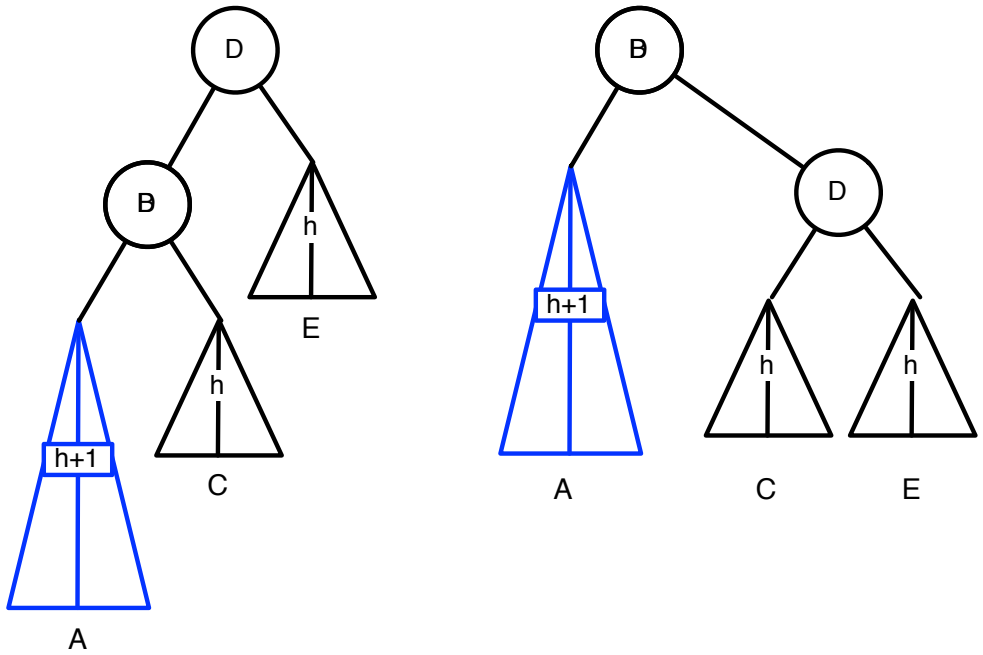
- Make a child or grandchild of the pivot node into the new root.
- Move around the subtrees rooted at the children and grandchildren of the pivot node to ensure that the binary search property holds.
- Specifics of the rotation depend on relative heights of subtrees rooted at children and grandchildren of the pivot node.

In all upcoming slides:

- Letter labels show order of values
- Left image: Before rotation, subtree rooted at the pivot node
- Right image: After rotation, replacement subtree

Rotation Case 1

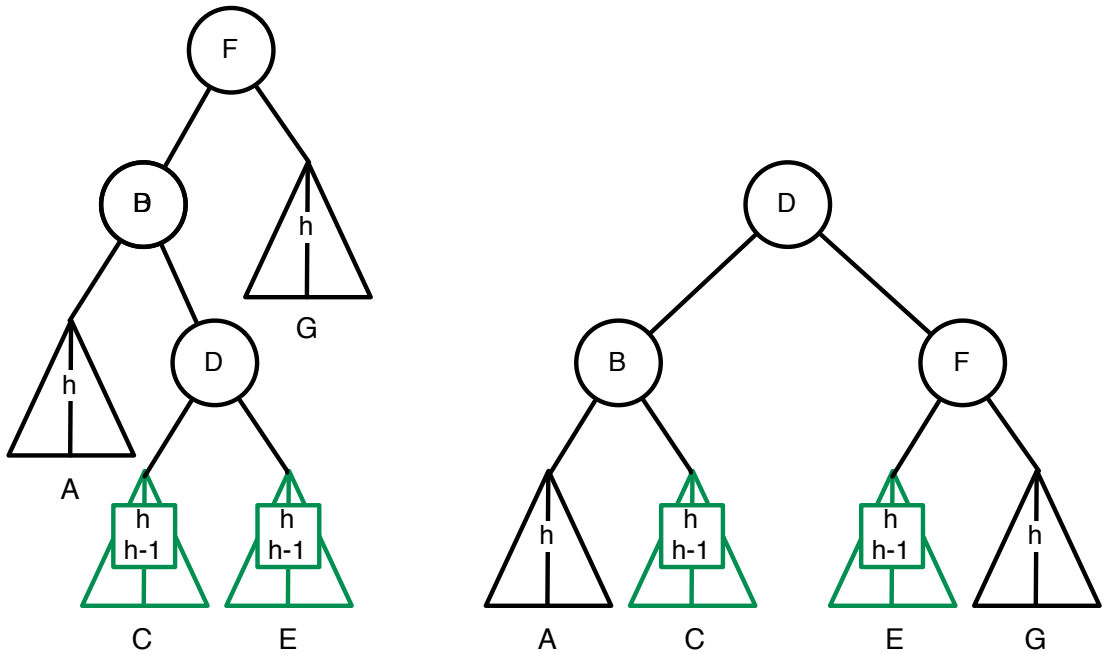
- At pivot, height of left subtree is 2 + height of right subtree
- At left child, height of left subtree is greater than height of right subtree



Change in height of subtree from $h+3$ to $h+2$.

Rotation Case 2

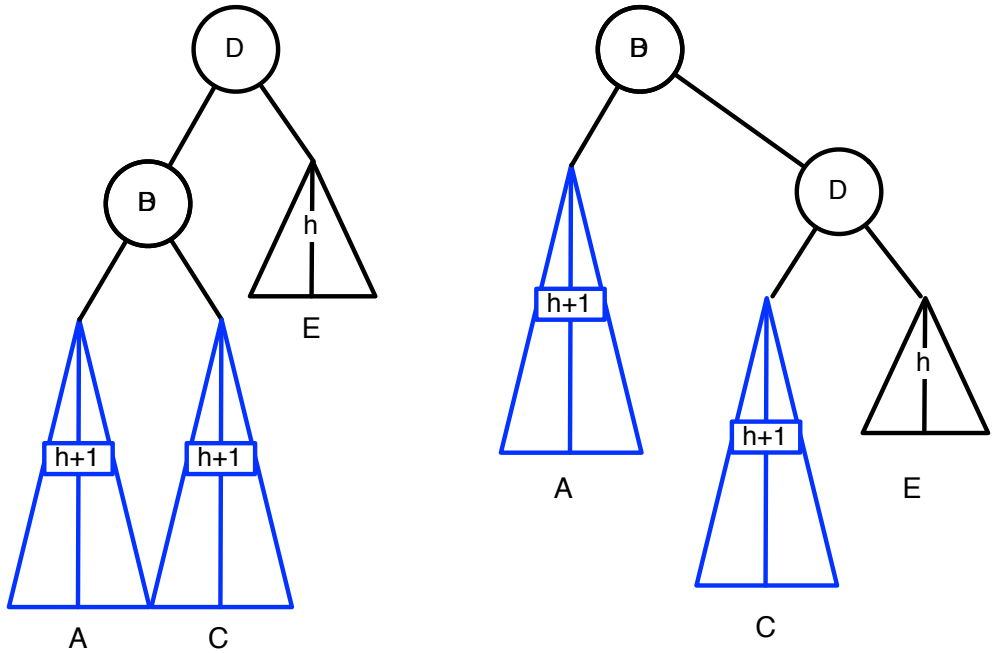
- At pivot, height of left subtree is 2 + height of right subtree
- At left child, height of right subtree is greater than height of left subtree



Change in height from $h+3$ to $h+2$.

Rotation Case 3

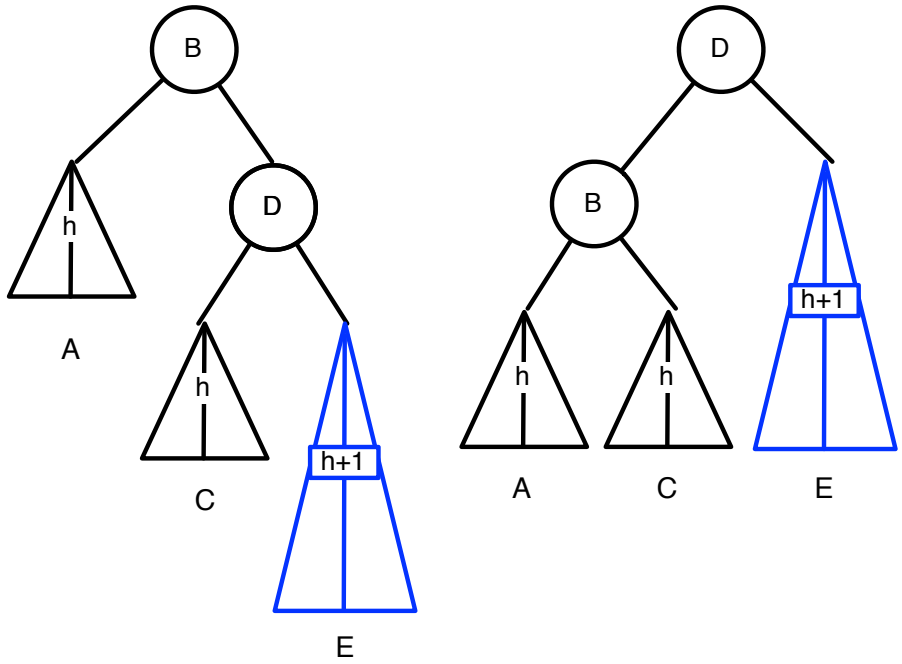
- At pivot, height of left subtree is 2 + height of right subtree
- At left child, heights of left and right subtrees are equal



Change in height of subtree from $h+3$ to $h+3$.

Rotation Case 4

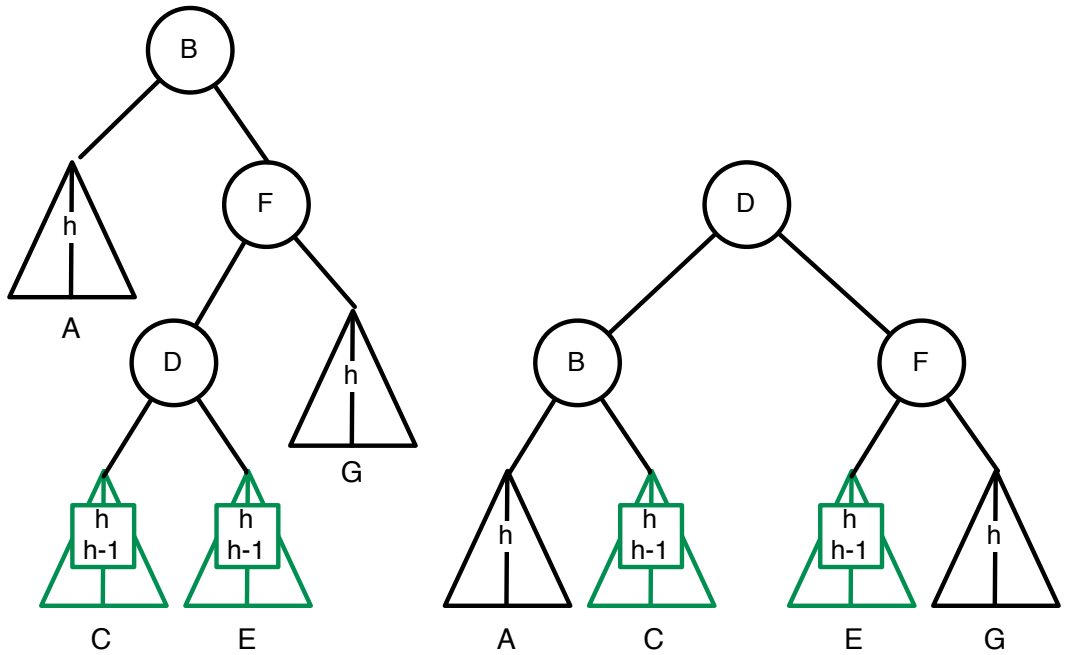
- At pivot, height of right subtree is 2 + height of left subtree
- At right child, height of right subtree is greater than height of left subtree



Change in height from $h+3$ to $h+2$.

Rotation Case 5

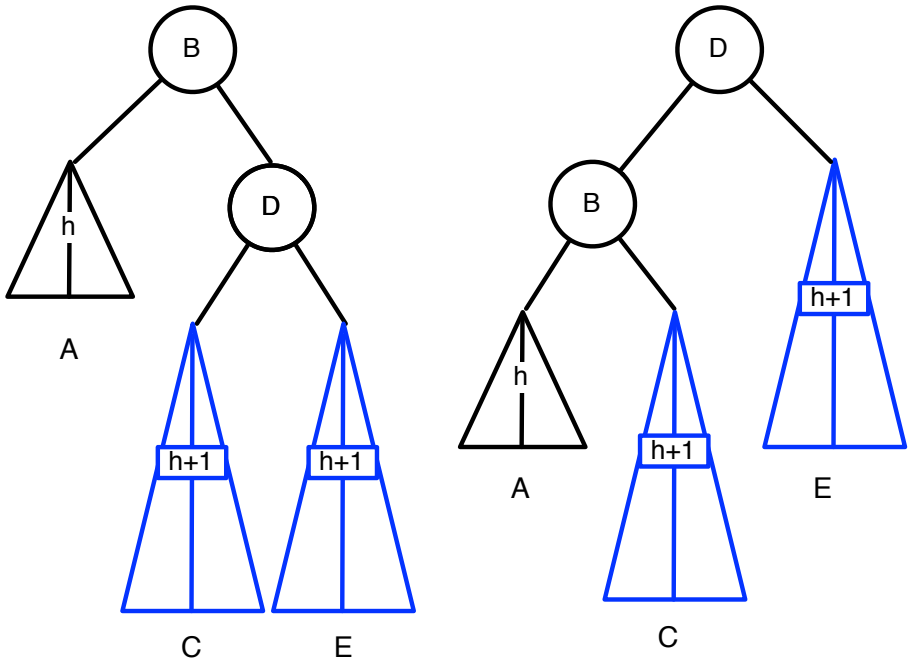
- At pivot, height of right subtree is 2 + height of left subtree
- At right child, height of left subtree is greater than height of right subtree



Change in height from $h+3$ to $h+2$.

Rotation Case 6

- At pivot, height of right subtree is 2 + height of left subtree
- At right child, heights of left and right subtrees are equal



Change in height of subtree from $h+3$ to $h+3$.

Analysis of implementation of *ADD*

Observations:

- Cases 3 and 6 cannot happen, as adding one node cannot result in two subtrees getting too tall.
- In all remaining cases, height of subtree goes from $h+3$ (too tall due to addition) to $h+2$.
- After one rotation, the rest of the tree is fixed.

Cost of *ADD*:

- Search in $O(\log n)$ time.
- Update in $O(1)$ time.
- Visit $O(\log n)$ nodes.
- Execute one rotation in $O(1)$ time.
- Worst case total $\Theta(\log n)$.

Analysis of implementation of *DELETE*

Observations:

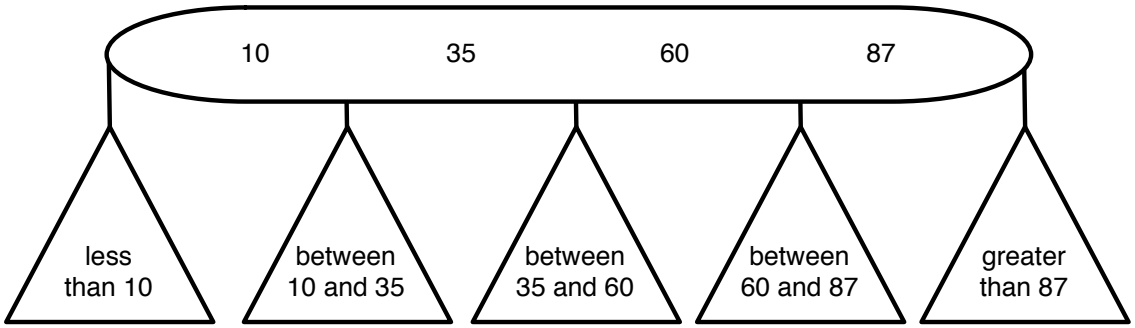
- In Cases 3 and 6, the height before deletion and the height after rotation are the same.
- In the other cases, the height before deletion is one greater than the height after rotation.
- After one rotation, more rotations might be required.

Cost of *DELETE*:

- Search in $O(\log n)$ time (including finding of inorder successor, if needed).
- Update in $O(1)$ time.
- Visit $O(\log n)$ nodes.
- Execute at most one rotation per level.
- Worst case total $\Theta(\log n)$.

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 (number of keys per node) 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 or only key, in the middle subtree (if any) are between the first and second keys, and in the right subtree are greater than the second or only key.

[scale=.4]../images/23tree.pdf

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

Augmenting the ADT Ordered Tree

We will augment the ADT Ordered Tree in order to support:

- Up to two (key, element) pairs in each node
- Up to three children for each node

Modified ADT Ordered Tree operations:

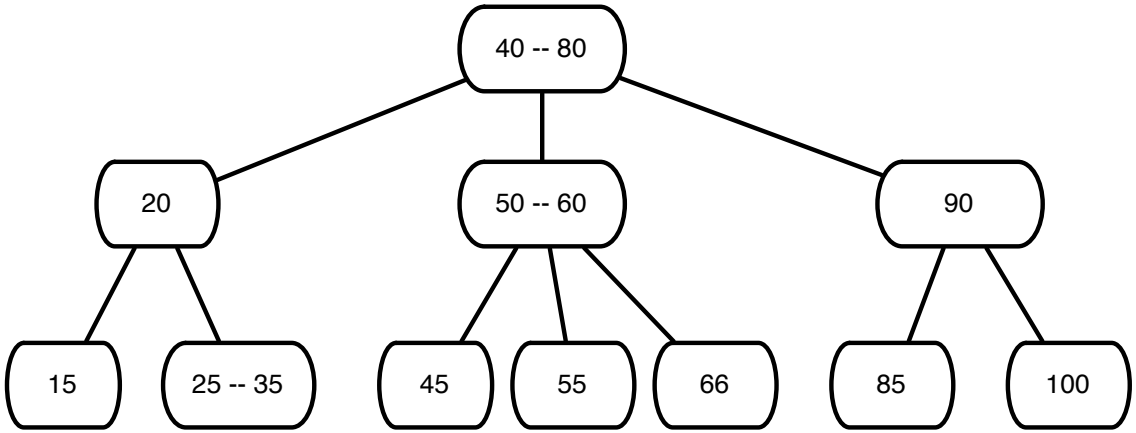
- *ADD_LEAF(O, Par, Sib, Key, Element)*
- *SET_VALUE(O, Node, Key1, Element1, Key2, Element2)*

New operations:

- *KEY_ONE(O, Node)*
- *ELEMENT_ONE(O, Node)*
- *KEY_TWO(O, Node)*
- *ELEMENT_TWO(O, Node)*

If a node has a single key, then the second key and element are empty.

Implementing *LOOK_UP* 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* 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:

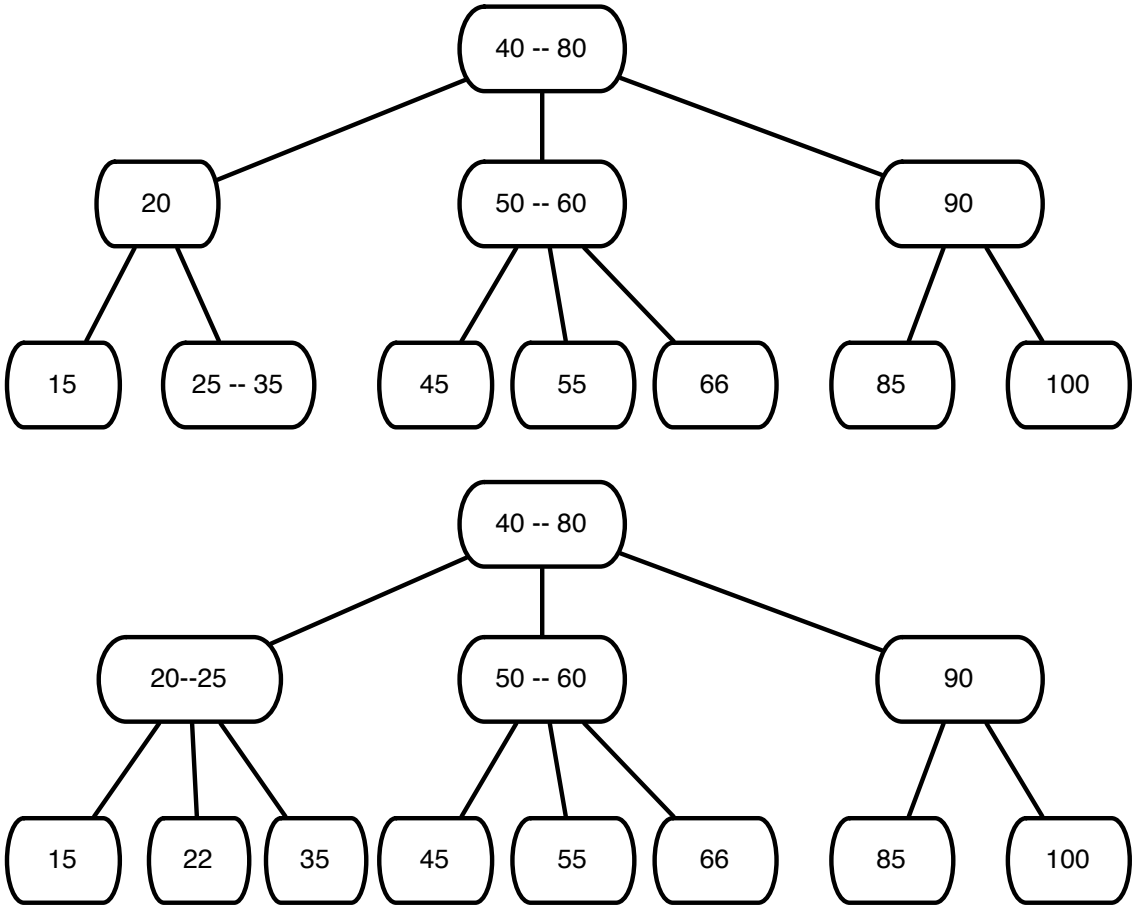
- Search leads to a leaf.
- Add item to leaf.
- If overflow, split leaf.
- If splitting the leaf leads to overflow in the parent, then the parent may need to be split as well.
- Splitting can propagate up to the root.

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.

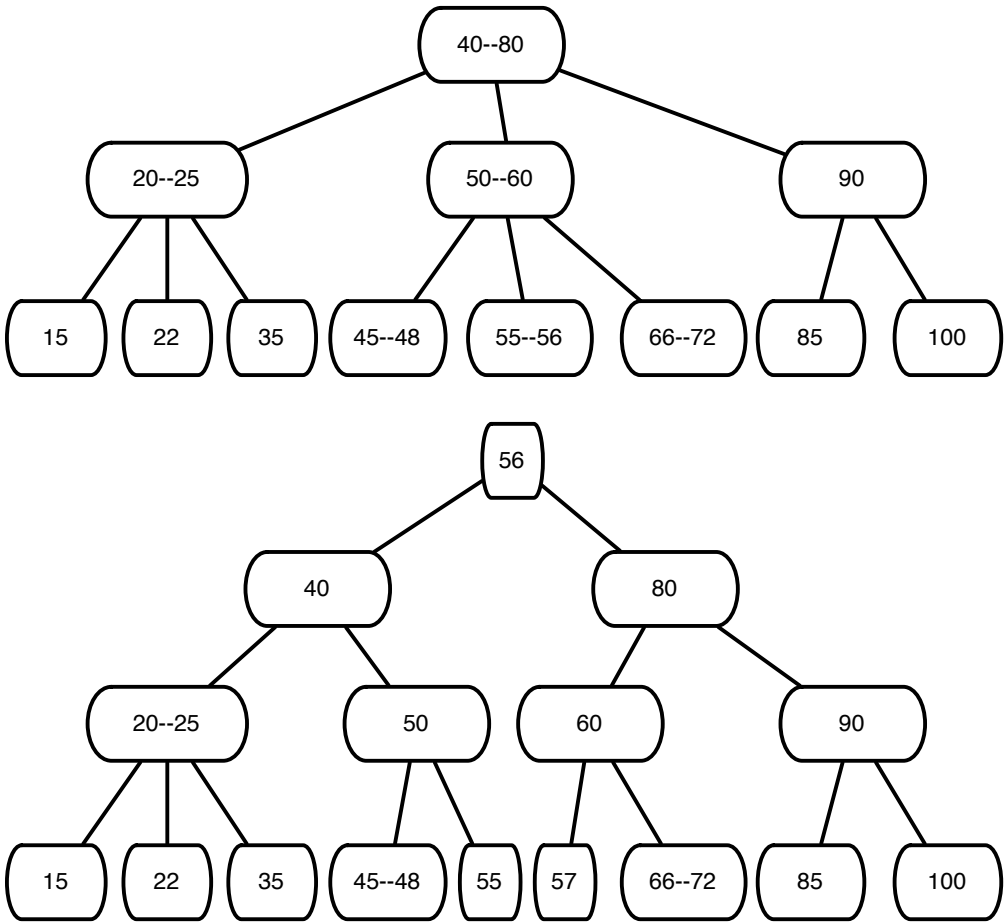
Implementing *ADD*, single split

Add 22.



Implementing *ADD*, cascading split

Add 57.



Implementing *DELETE* 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.
- Inorder successor must be in a leaf (otherwise it has a left child).
- Remove item from leaf.
- If underflow, need to rearrange values or 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.

Analysis of addition and deletion

Cost of *ADD*:

- Search in $O(\log n)$ time.
- Update in $O(1)$ time.
- Execute one split in $O(1)$ time.
- Number of splits is in $O(\log n)$.
- Worst case total $\Theta(\log n)$.

Cost of *DELETE*:

- Search in $O(\log n)$ time (including finding of inorder successor, if needed).
- Update in $O(1)$ time.
- Execute one fuse in $O(1)$ time.
- Number of fuses is in $O(\log n)$.
- Worst case total $\Theta(\log n)$.

Module summary

Topics covered:

- Case study: Student records
- ADT Dictionary
- Array and linked implementations
- Compound data
- Special case: Orderable keys
- Searching in sorted data structures
- Ordered array implementation
- Ordered linked implementation
- Searching
- Creating an ADT Dictionary filled with n items
- Revisiting comparison-based sorting algorithms
- Data structure: Binary Search Tree
- Data structure: AVL tree
- Multiway search tree
- Data structure: (2,3) tree