CSE 326 Lecture 9: Splay Trees and B-Trees

- What's on our plate today?
  - Splaying: Examples and Run Time Analysis
  - B-Trees
    - Insert/Delete Examples and Run Time Analysis
  - Introduction to Heaps and Priority Queues
- Covered in Chapters 4 and 6 in the text

Splay Trees Recap

Splay trees are binary search trees that:

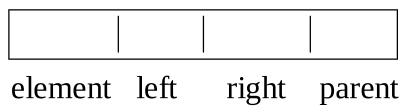
1. Are not perfectly balanced all the time
2. Allow each access to a node to balance the tree so that future operations may run faster

Main Ideas:

- After node X is accessed, perform "splaying" operations to bring X up to the root using rotations.
- Side Effect: Tends to leave the tree more balanced.
- Net Result: Can prove that average (amortized) run time = O(log N) per access over a sequence of ADT operations

Splay Tree Node and Splay Z/ZZ Operations

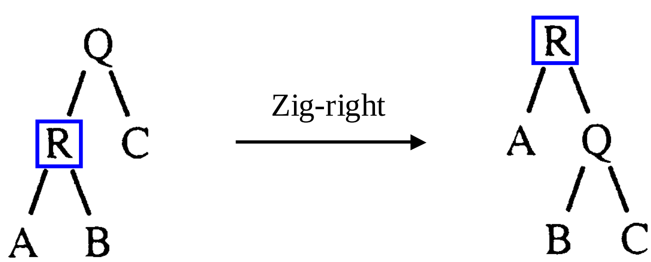
1. Nodes must contain a parent pointer.



2. When X is accessed, apply one of six rotation operations:
- Single Rotations (X has a Parent but no Grandparent)
  - zig-left, zig-right
- Double Rotations (X has both a Parent and a Grandparent)
  - zig-zig-left, zig-zig-right
  - zig-zag-left, zig-zag-right

Zig-Right

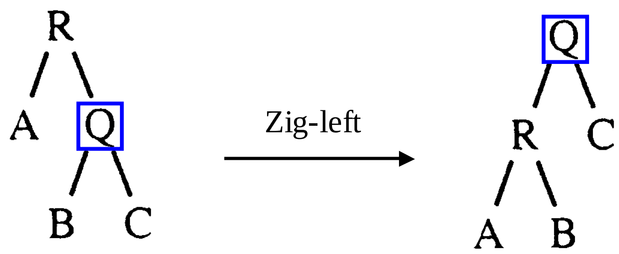
- "Zig-Right" is just a single right rotation, as in an AVL tree
- Suppose R was the node that was accessed (e.g. using Find)



- Zig-right moves R to the top can access R faster next time

Zig-Left

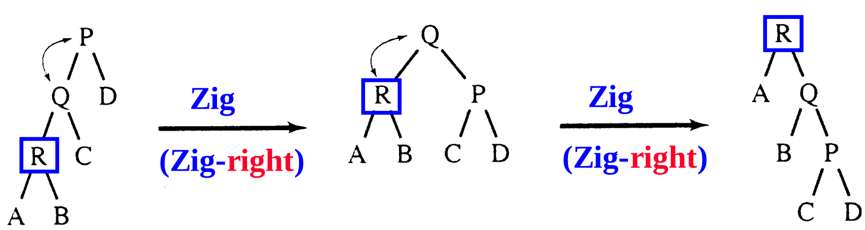
- Suppose Q is accessed (e.g. using Find)



- Zig-left is a single left rotation: moves Q to the top

Zig-Zig

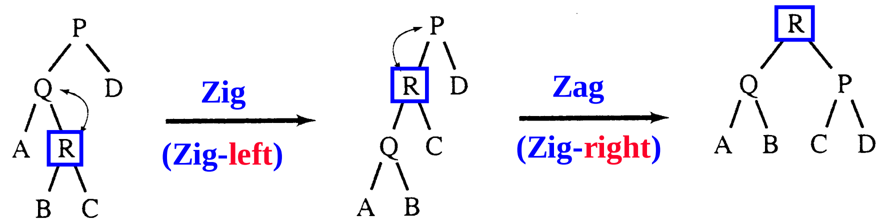
- "Zig-Zig" consists of two single rotations of the same type (assume R is the node that was accessed):



- Note: Parent-Grandparent rotated first
- Zig-Zig Left is just Zig-Left followed by Zig-Left

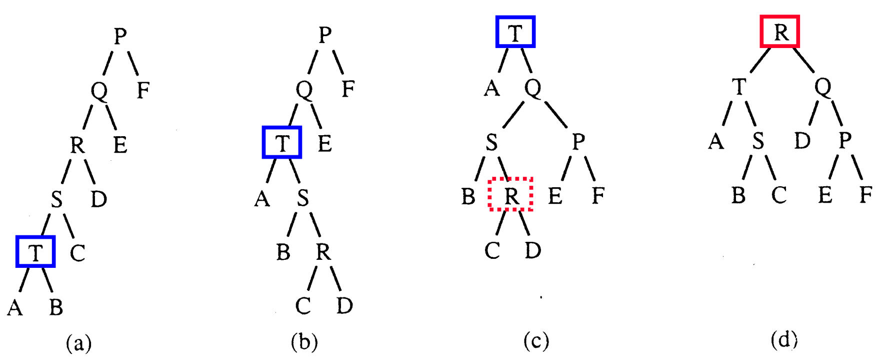
Zig-Zag

- "Zig-Zag" consists of <u>two rotations</u> of the <u>opposite type</u> (assume R is the node that was accessed):



- <u>Note: R and Parent rotated first, R and Grandparent next</u>
- The <u>other Zig-Zag</u> is just Zig-Right followed by Zig-Left

Splay Trees: Example



Restructuring a tree with splaying after <u>accessing</u> $\boxed{T}$ (a–c) and then $\boxed{R}$ (c–d).

Splaying during Other Operations

- Splaying can be done not just after Find, but also after other ADT operations such as Insert/Delete.
- Insert X: After inserting X at a leaf node (as in a regular BST), splay X up to the root
- Delete X: Do a Find on X and get X up to the root. Delete X at the root and move the largest item in its left subtree to the root using splaying.
- Note on Find X: If X was not found, splay the leaf node that the Find ended up with to the root.

Analysis of Splay Trees: Amortization

Examples suggest that splaying causes tree to get balanced. The actual analysis is rather advanced and is in Chapter 11.

Result of Analysis: Any sequence of M operations on a splay tree of size N takes O(M log N) time.

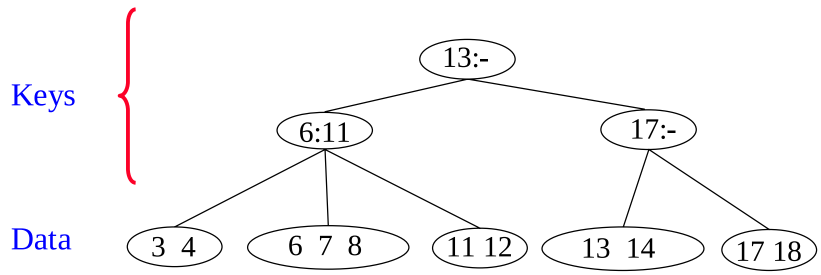
So, the amortized running time for one operation is O(log N).

This guarantees that even if depths of some nodes get very large, you cannot get a long sequence of O(N) operations.

Without splaying, total time could be O(MN).

Beyond Binary Search Trees: Multi-Way Trees

- "B-tree" of order 3: Tree has 2 or 3 children per node



- Example: Search for 8

B-Trees

B-Trees are multi-way search trees commonly used in database systems or other applications where data is stored externally on disks and keeping the tree shallow is important.

A B-Tree of order M has the following properties:

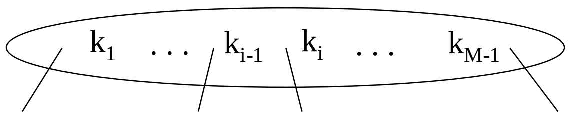
1. The root is either a leaf or has between 2 and M children.
2. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
3. All leaves are at the same depth.

All data records are stored at the leaves.
Leaves store between $\lceil L/2 \rceil$ and L data records.
L depends on disk block size and data record size (e.g. L = M).

B-Tree Details

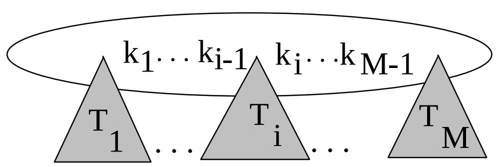
Each internal node of a B-tree has:

- Between $\lceil M/2 \rceil$ and M children.
- up to M-1 keys $k_1 < k_2 < ... < k_{M-1}$



Keys are ordered so that:
$k_1 < k_2 < ... < k_{M-1}$

Properties of B-Trees



Children of each internal node are "between" the items in that node.
Suppose subtree T_i is the *i*th child of the node:

All keys in T_i must be between k_{i-1} and k_i

i.e. $k_{i-1} \leq T_i < k_i$

k_{i-1} is the smallest key in T_i

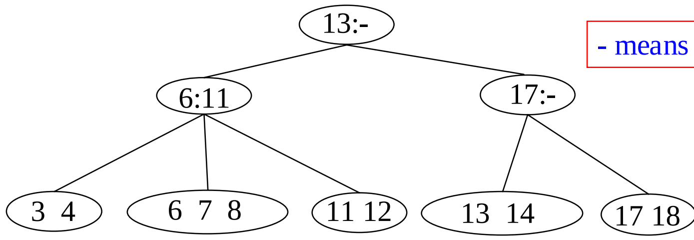
All keys in first subtree $T_1 < k_1$

All keys in last subtree $T_M \geq k_{M-1}$

B-trees Example

- B-tree of order 3: also known as 2-3 tree (2 to 3 children)

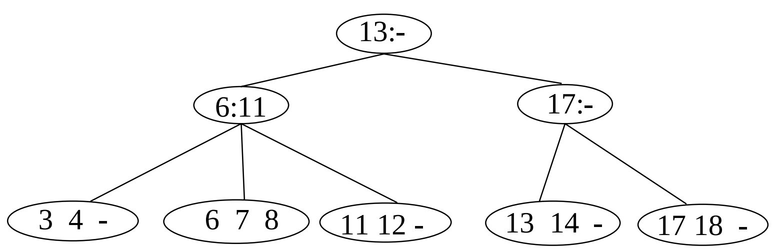
- means empty slot



- Apply B-tree definition for order M = 3 and L = M = 3
  - Each node must have at least $\lceil M/2 \rceil = 2$ and at most M = 3 children
  - Leaves store between $\lceil M/2 \rceil = 2$ and M = 3 data records

Inserting Items in B-Trees

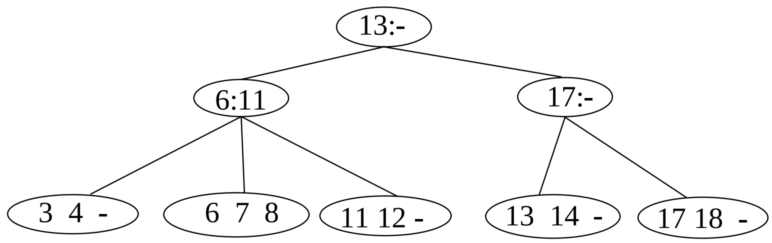
- Insert X: Do a Find on X and find appropriate leaf node
  - If leaf node is not full, fill in empty slot with X. E.g. Insert 5 in the tree below
  - If leaf node is full, split leaf node and adjust parents up to root node. E.g. Insert 9 in the tree below



Deleting Items in B-Trees

- Delete X: Do a Find on X and delete value from leaf node
  - May have to combine leaf nodes and adjust parents up to root node if number of data items falls below $\lceil M/2 \rceil = 2$

    E.g. Delete 17 in the tree below



Run Time Analysis of B-Tree Operations

- For a B-Tree of order M
  1. Each internal node has up to M-1 keys to search
  2. Each internal node has between $\lceil M/2 \rceil$ and M children

     i.e. Depth of B-Tree storing N data items is $O(\log_{\lceil M/2 \rceil} N)$

     (Why? Hint: Draw a B-tree with minimum children at each node. Count its leaves as a function of depth)

- Find: Run time is:

  O(log M) to binary search which branch to take at each node

  Total time to find an item is O(depth*log M) = O(log N)

What about Insert/Delete?

- For a B-Tree of order M
  Depth of B-Tree storing N items is $O(\log_{\lceil M/2 \rceil} N)$
- Insert and Delete: Run time is:
  - O(M) to handle splitting or combining keys in nodes
  - Total time is O(depth*M) = $O((\log N/\log\lceil M/2\rceil)*M)$
    = $O((M/\log M)*\log N)$
- Tree in internal memory   M = 3 or 4
- Tree on Disk   M = 32 to 256. Interior and leaf nodes fit on 1 disk block.
  - Depth = 2 or 3   allows very fast access to data in large databases.

Summary of Search Trees

- Problem with Search Trees: Must keep tree balanced to allow fast access to stored items
- AVL trees: Insert/Delete operations keep tree balanced
- Splay trees: Sequence of operations produces balanced trees
- Multi-way search trees (e.g. B-Trees): More than two children per node allows shallow trees; all leaves are at the same depth keeping tree balanced at all times

Next Class:

Heaps on Heaps

To Do:

Read Chapter 6

Homework # 2 (due this Friday)