Splay Trees and B-Trees

CSE 373 - Data Structures

April 19, 2002

Readings and References

- Reading
  - › Section 4.5-4.7, *Data Structures and Algorithm Analysis in C*, Weiss
- Other References

Self adjustment for better living

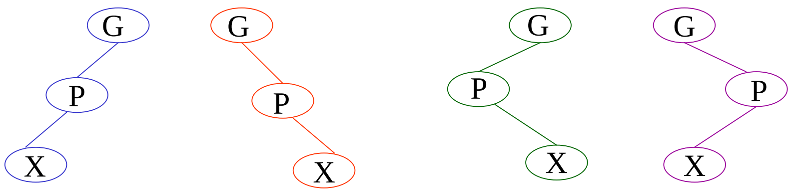
- Ordinary binary search trees have no balance conditions
  - › what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - › tree is always balanced after an insert or delete
- Self adjusting trees get reorganized over time as nodes are accessed

Splay Trees

- Splay trees are tree structures that:
  - Are not perfectly balanced all the time
  - Use Find operations to balance the tree
    - future operations may run faster
- Based on the heuristic:
  - If X is accessed once, it is likely to be accessed again.
- The procedure:
  - After node X is accessed, perform “splaying” operations to bring it up to the root of the tree.
  - Do this in a way that leaves the tree more balanced as a whole

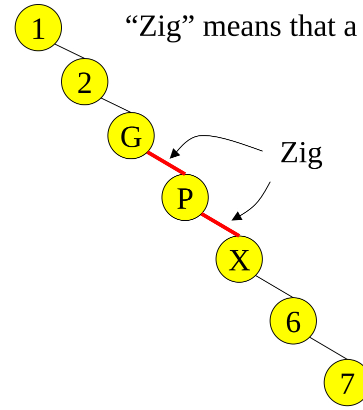
Splay Tree Terminology

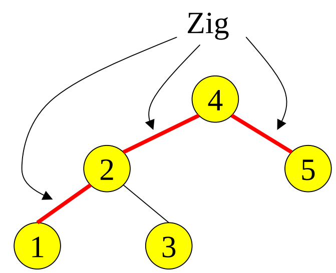
- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.



Zig

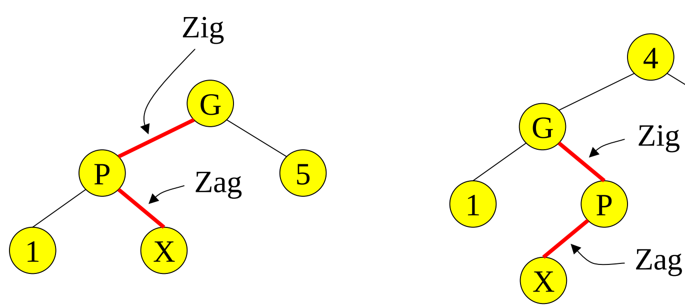
“Zig” means that a node is the outside child of its parent



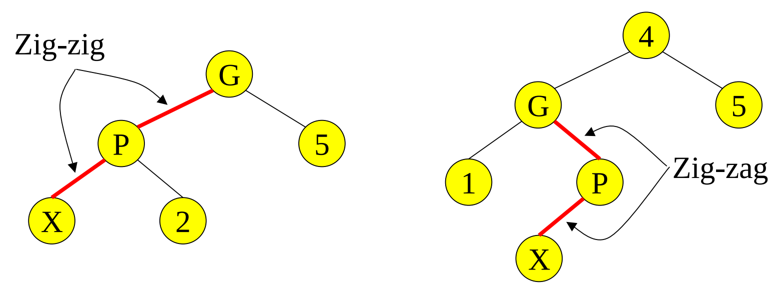


Zag

“Zag” means that a node is the inside child of its parent, relative to the "zig" above it

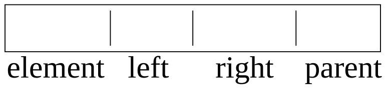


Zig-Zig and Zig-Zag



Splay Tree Operations

1. Nodes must contain a parent pointer.

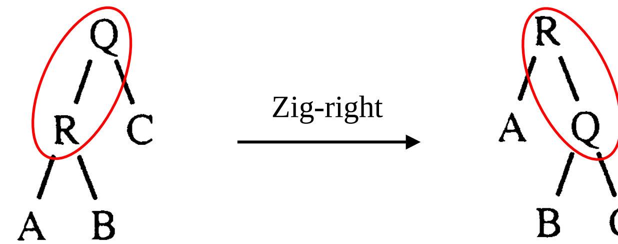


2. When X is accessed, apply one of six rotation routines.

- Single Rotations (X has a P (the root) but no G)
  - zig_left, zig_right
- Double Rotations (X has both a P and a G)
  - zig_zig_left, zig_zig_right
  - zig_zag_left, zig_zag_right

Zig at depth 1

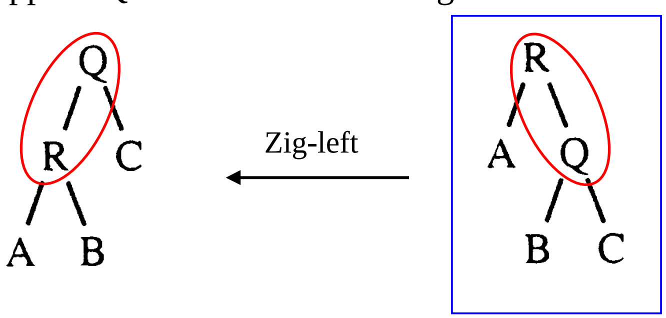
- “Zig” is just a single rotation, as in an AVL tree
- Let R be the node that was accessed (e.g. using Find)



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

Zig at depth 1

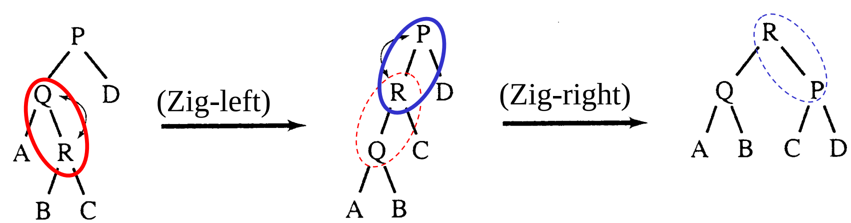
- Suppose Q is now accessed using Find



- Zig-left moves Q back to the top

Zig-Zag operation

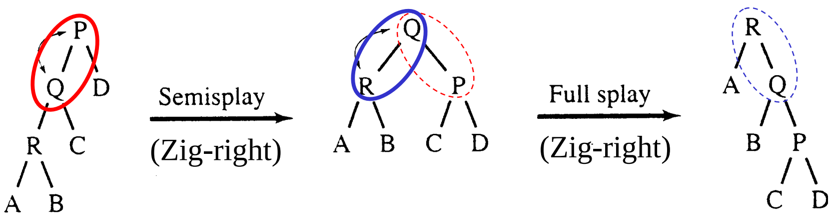
- “Zig-Zag” consists of two rotations of the opposite direction (assume R is the node that was accessed)



- “Zig-Zag” splaying causes R to move to the top

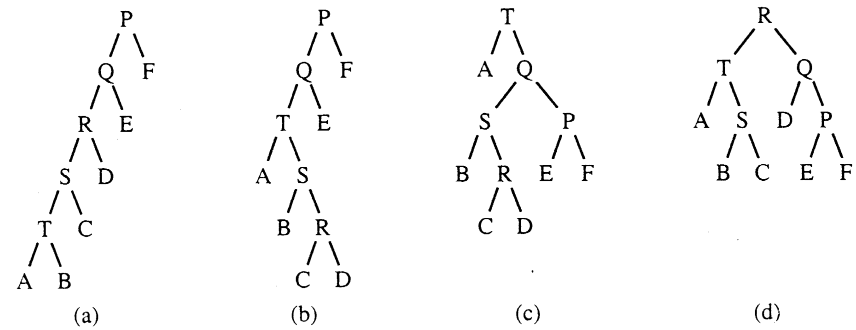
Zig-Zig operation

- “Zig-Zig” consists of two single rotations of the same direction (R is the node that was accessed)



- Again, due to “zig-zig” splaying, R has bubbled to the top

Decreasing depth - "autobalance"



Restructuring a tree with splaying after accessing *T* (a–c) and then *R* (c–d).

Analysis of Splay Trees

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

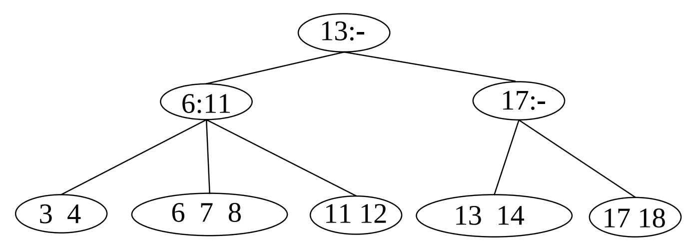
Result of Analysis: Any sequence of M operations on a splay tree of size N takes O(M log N) time.

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

This guarantees that even if the depths of some nodes get very large, you cannot get a long sequence of O(N) searches because each search operation causes a rebalance.

Beyond Binary Search Trees: Multi-Way Trees

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



- 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.

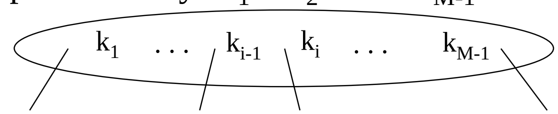
All data records are stored at the leaves.
Leaves store between $\lceil M/2 \rceil$ and M data records.

B-Tree Details

Each internal node of a B-tree has:

› Between $\lceil M/2 \rceil$ and M children.

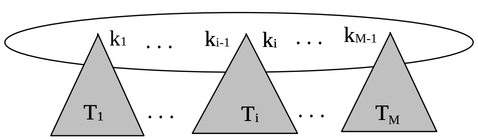
› up to M-1 keys $k_1 < k_2 < ... < k_{M-1}$



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 ith child of the node:

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

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

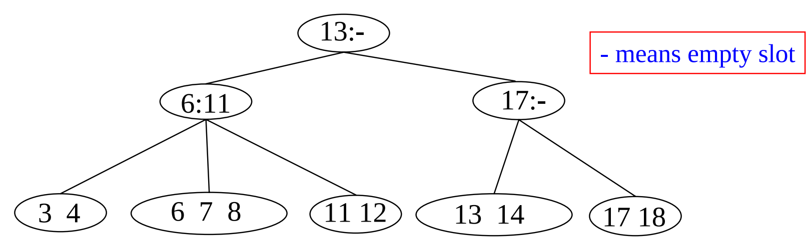
k_{i-1} is the smallest key in T_i

All keys in first subtree $T_1 < k_1$

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

Example: Searching in B-trees

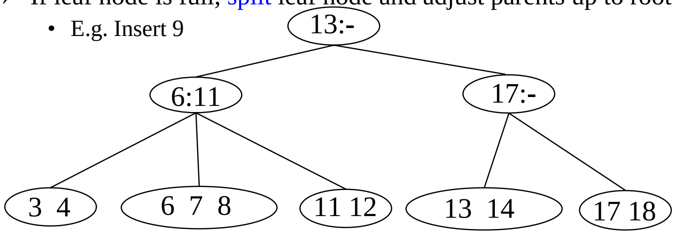
- B-tree of order 3: also known as 2-3 tree (2 to 3 children)



- Examples: Search for 9, 14, 12
- Note: If leaf nodes are connected as a Linked List, B-tree is called a B+ tree – Allows sorted list to be accessed easily

Inserting and Deleting 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
  - › If leaf node is full, split leaf node and adjust parents up to root node
    - E.g. Insert 9



- 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

Run Time Analysis of B-Tree Operations

- For a B-Tree of order M
  - › Each internal node has up to M-1 keys to search
  - › Each internal node has between $\lceil M/2 \rceil$ and M children
  - › Depth of B-Tree storing N items is $O(\log_{\lceil M/2 \rceil} N)$
- 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)

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: Repeated Find operations produce 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