# Trees

Fall 2020

School of Software Engineering

South China University of Technology

# Contents

- Definitions of tree
- Binary tree
- AVL tree
- Splay tree
- B-tree

# Splay Trees

# Readings

- Reading
  - Sections 4.5-4.7

# Self adjusting Trees

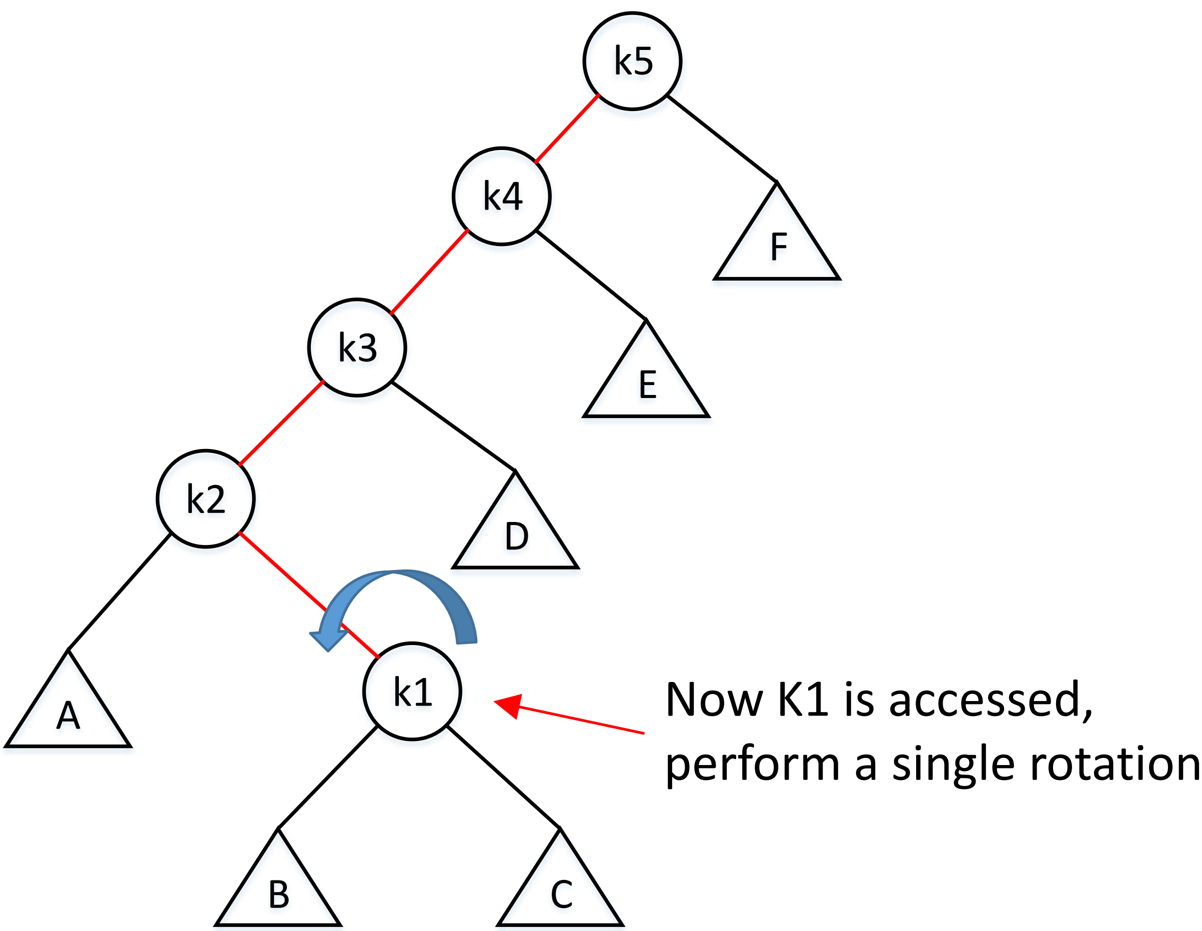
- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed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

# Splay Trees

- Splay trees are tree structures that:
  - Are not perfectly balanced all the time
  - Data most recently accessed is near the root. (principle of locality; 80-20 “rule”)
- The procedure:
  - After node X is accessed, perform “splaying” operations to bring X to the root of the tree.
  - Do this in a way that leaves the tree more balanced as a whole

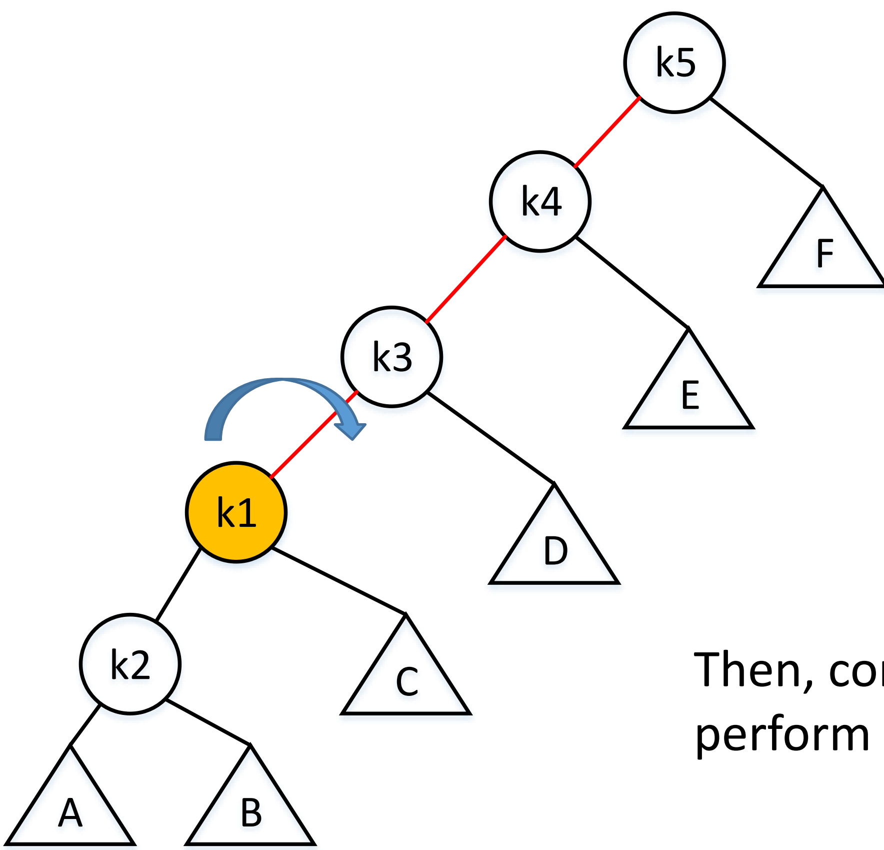
# Splay Trees

- How to perform splaying?
  - A Simple Idea
  - to perform single rotations, bottom up



# Splay Trees

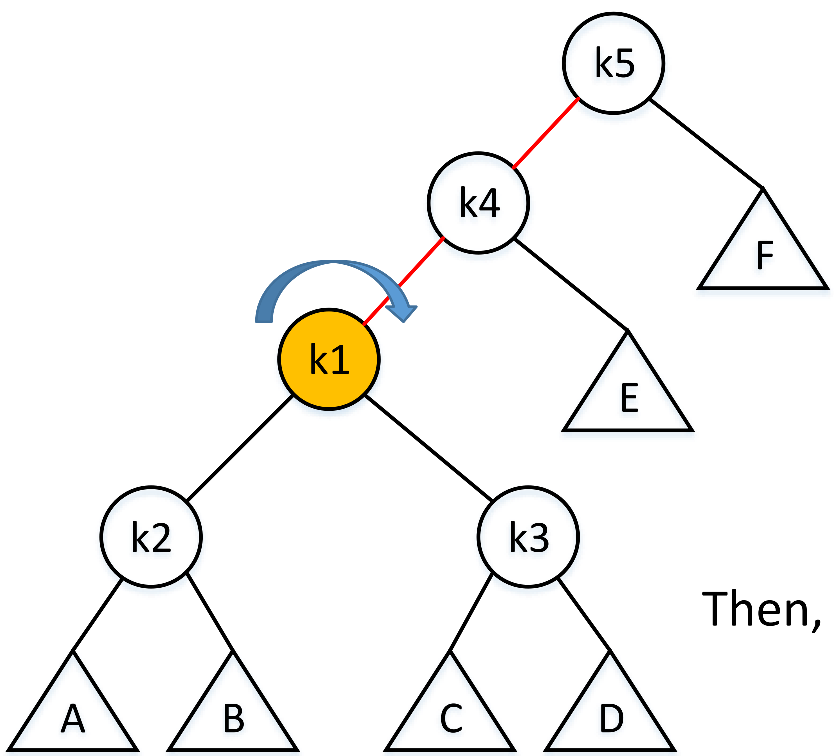
- How to perform splaying?
  - A Simple Idea
  - to perform single rotations, bottom up




Then, continue to perform a single rotation

# Splay Trees

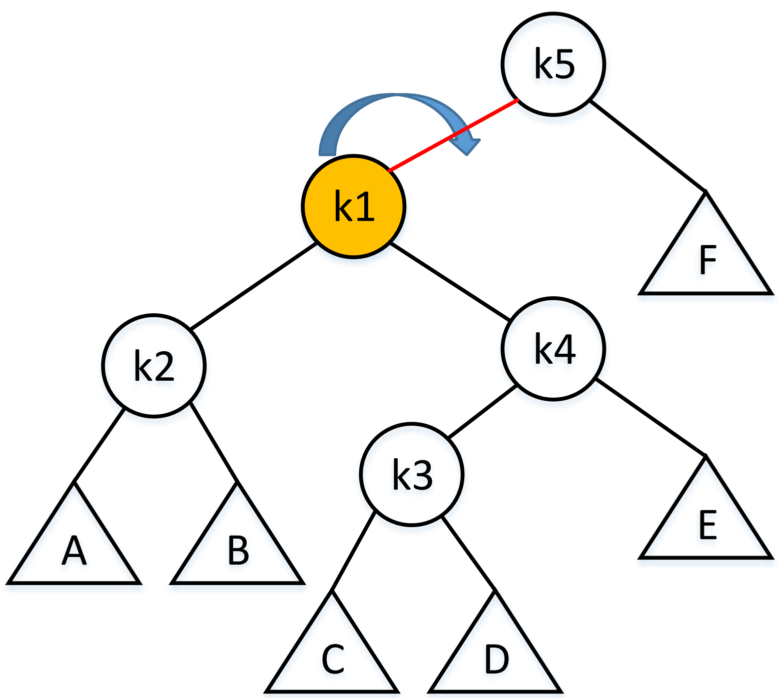
- How to perform splaying?
  - A Simple Idea
  - to perform single rotations, bottom up




Then, continue…

# Splay Trees

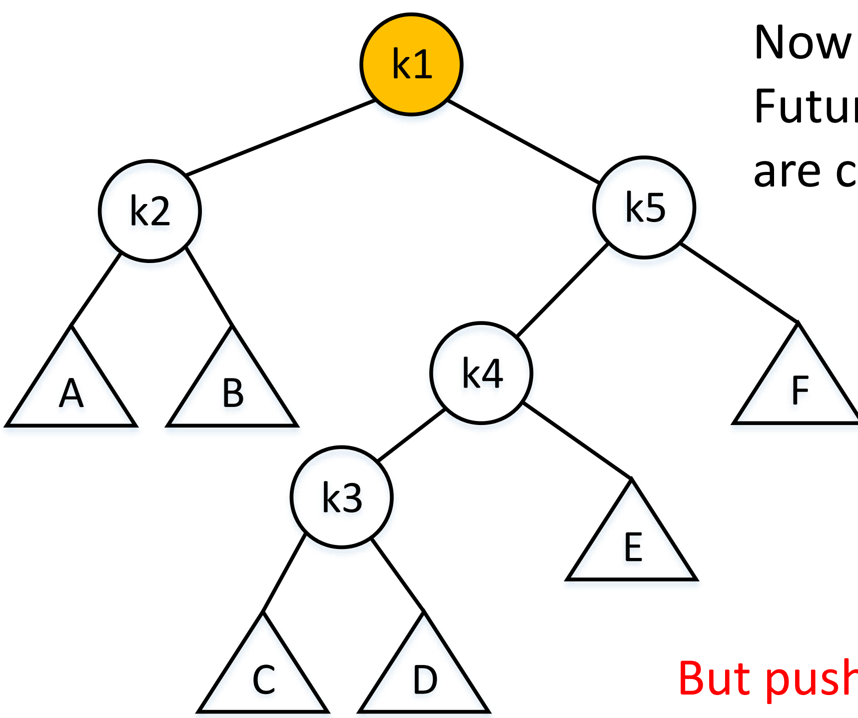
- How to perform splaying?
  - A Simple Idea
  - to perform single rotations, bottom up




continue …

# Splay Trees

- How to perform splaying?
  - A Simple Idea
  - to perform single rotations, bottom up



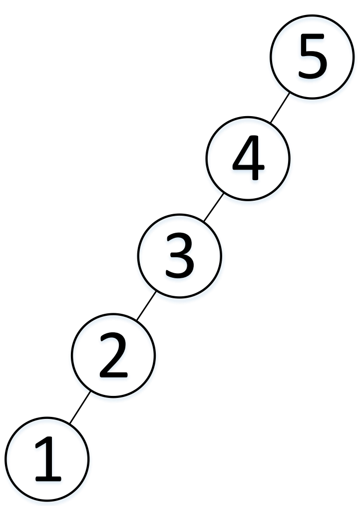

Now k1 reach the root. Future accesses on k1 are cheaper.

But pushing other node deep, ex. k3.

# Splay Trees

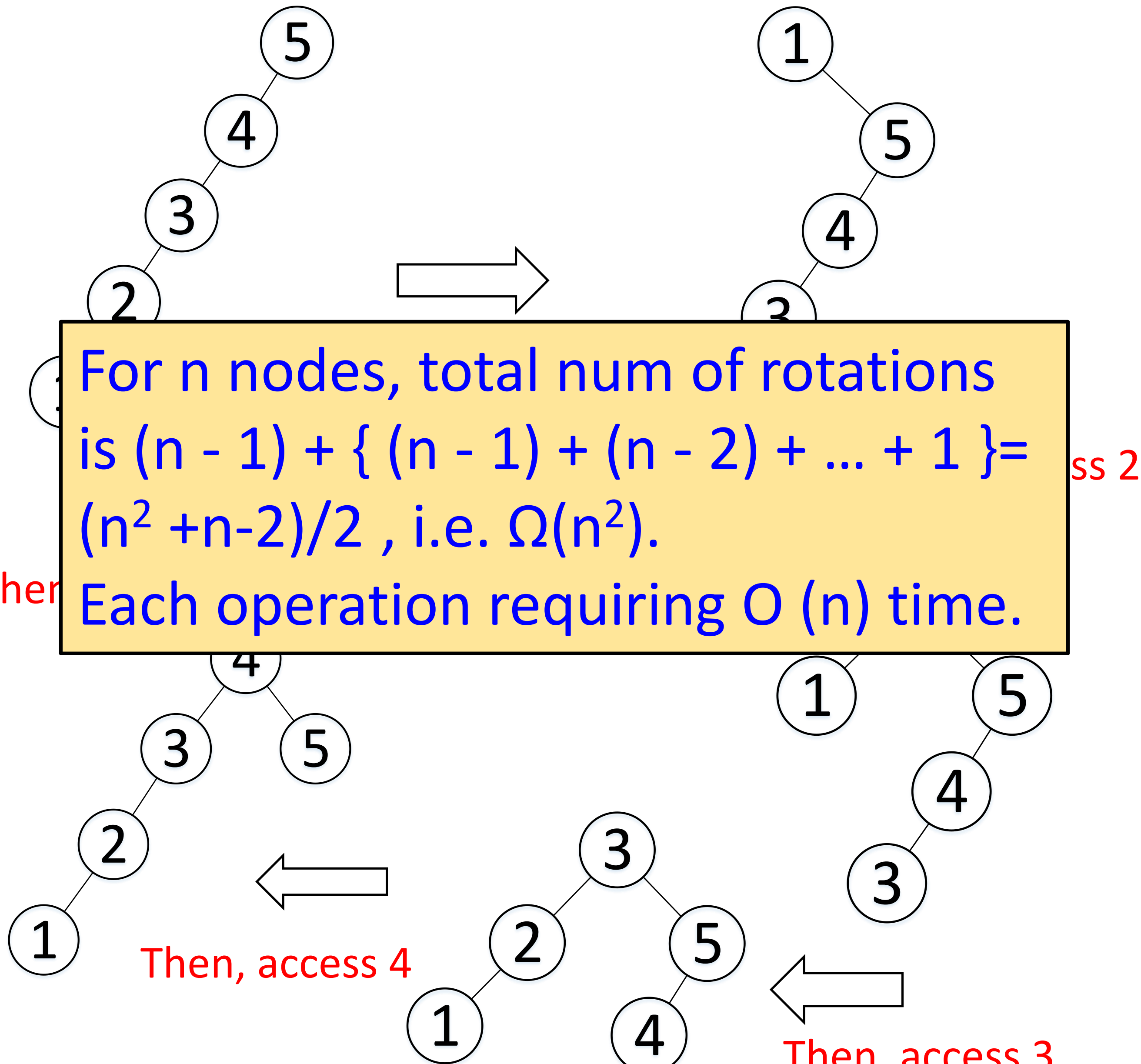
- How to perform splaying?
  - A Simple Idea
  - to perform single rotations, bottom up

  - This strategy is not good enough
    - A sequence of M operations requires $\Omega(N \cdot M)$




Consider the tree formed by inserting keys 1, 2, 3, . . . ,N into an initially empty tree

# Splay Trees




For n nodes, total num of rotations is $(n - 1) + \{ (n - 1) + (n - 2) + \ldots + 1 \} = (n^2 + n - 2)/2$, i.e. $\Omega(n^2)$.

Each operation requiring O (n) time.

# Splay Trees

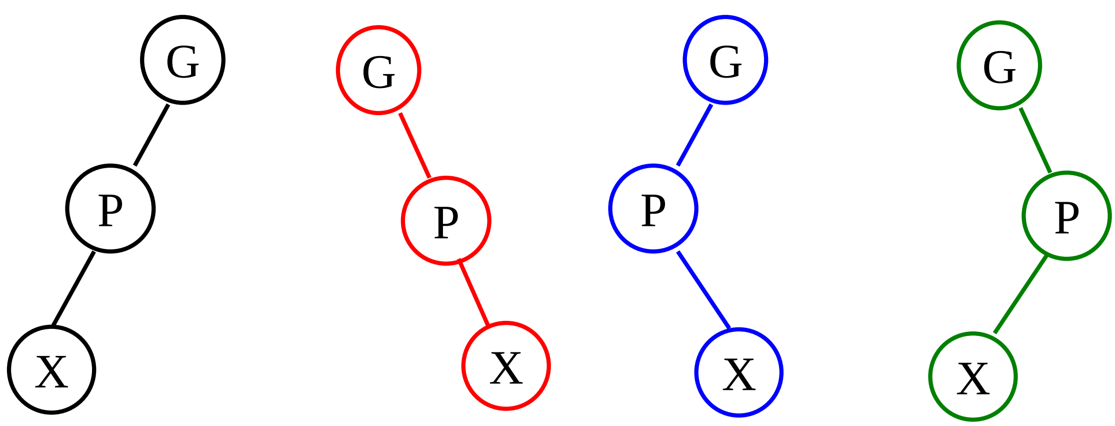
- Goal:
  - Principle of  locality
  - An O(logN) amortized cost per operation

- How to perform splaying?
  - Another strategy to perform double rotations (splay)

# Splay Trees

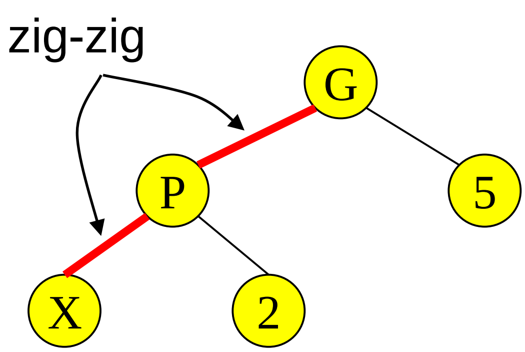
- Terminology

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

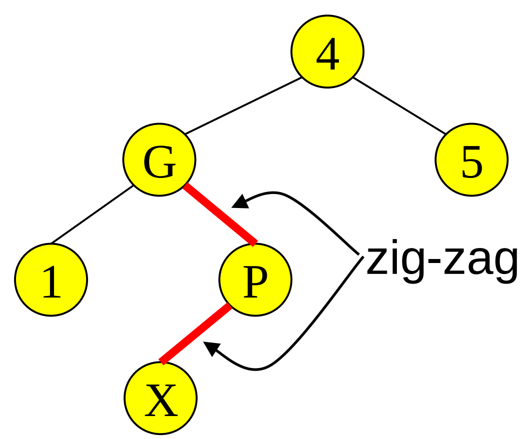


# Zig-Zig and Zig-Zag

Parent and grandparent in same direction.



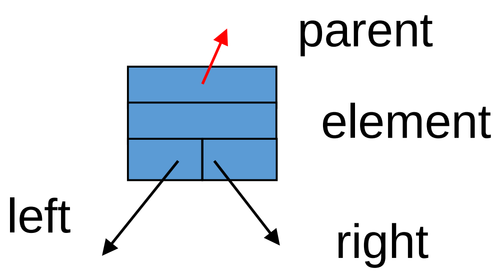

Parent and grandparent in different directions.



# Splay Trees

Splay Tree Operations:

1. Helpful if nodes contain a parent pointer.

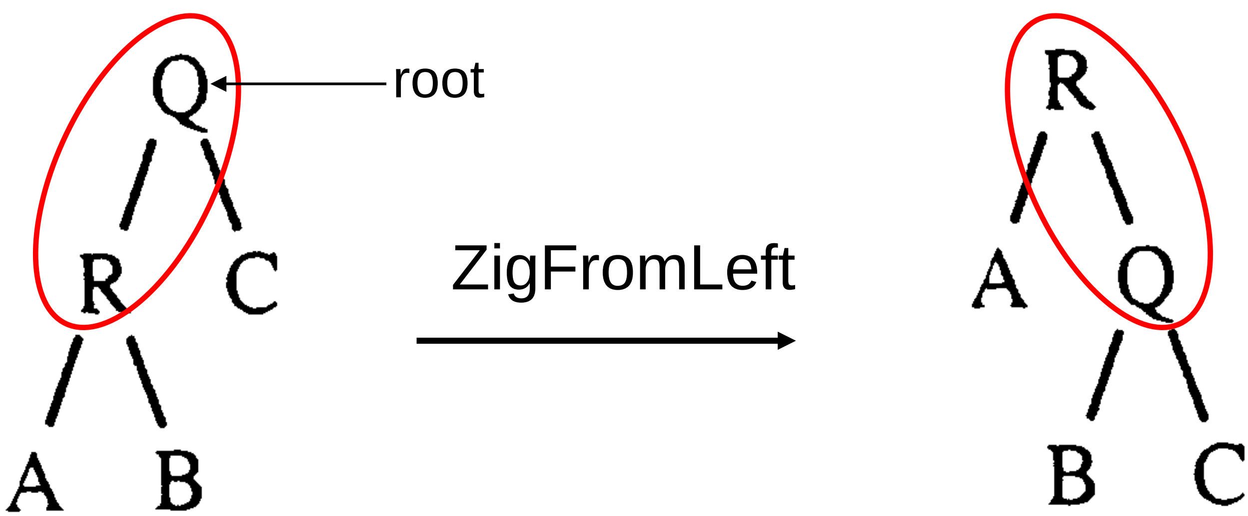



2. When X is accessed, apply one of six rotation routines.

- Single Rotations (X has a P (the root) but no G)
  ZigFromLeft, ZigFromRight
- Double Rotations (X has both a P and a G)
  ZigZigFromLeft, ZigZigFromRight
  ZigZagFromLeft, ZigZagFromRight

# Zig at depth 1 (single rotation)

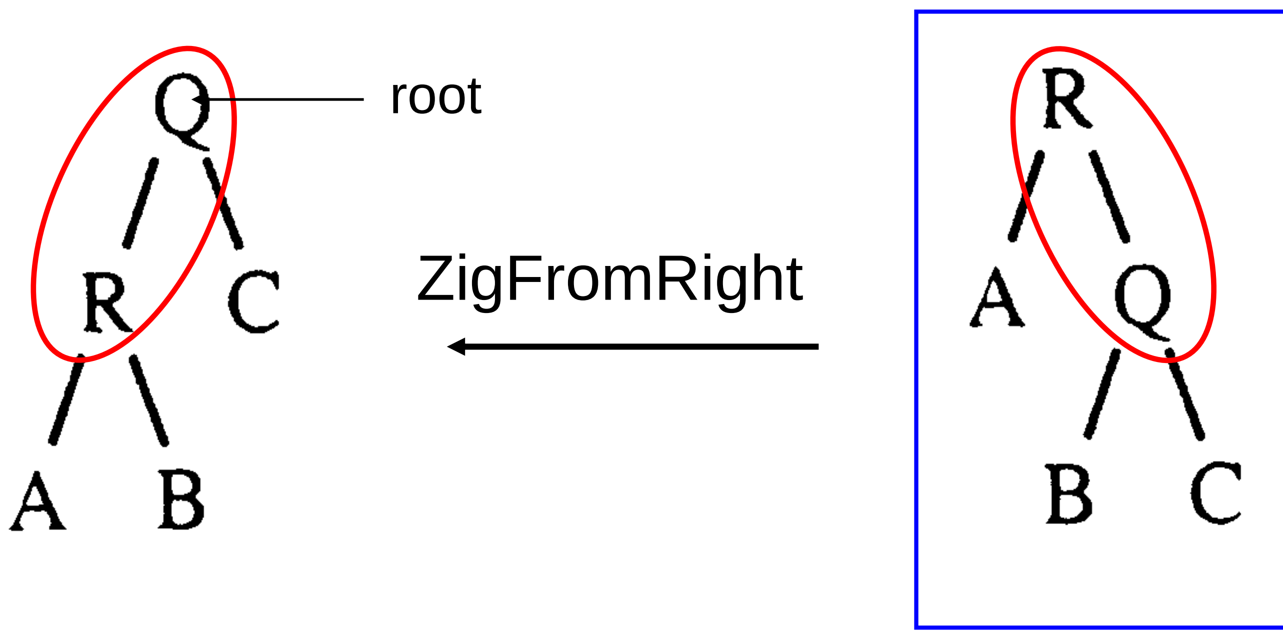
- “Zig” is just a single rotation, as in an AVL tree
- Let R be the node that was accessed (e.g. using Find)




- ZigFromLeft moves R to the top →faster access next time

# Zig at depth 1 (single rotation)

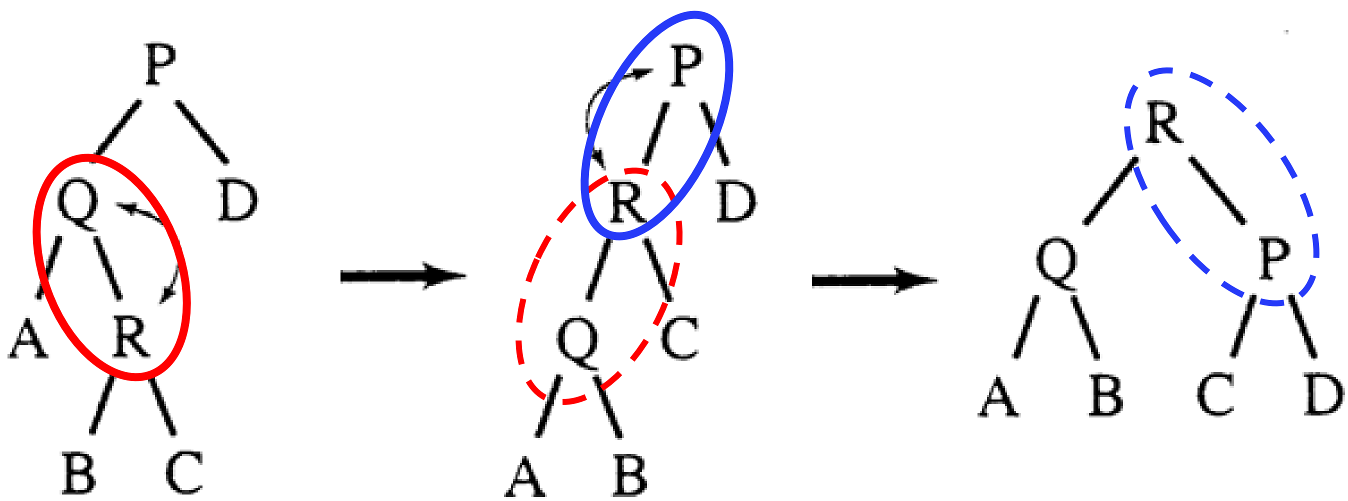
- Suppose Q is now accessed using Find




- ZigFromRight 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)

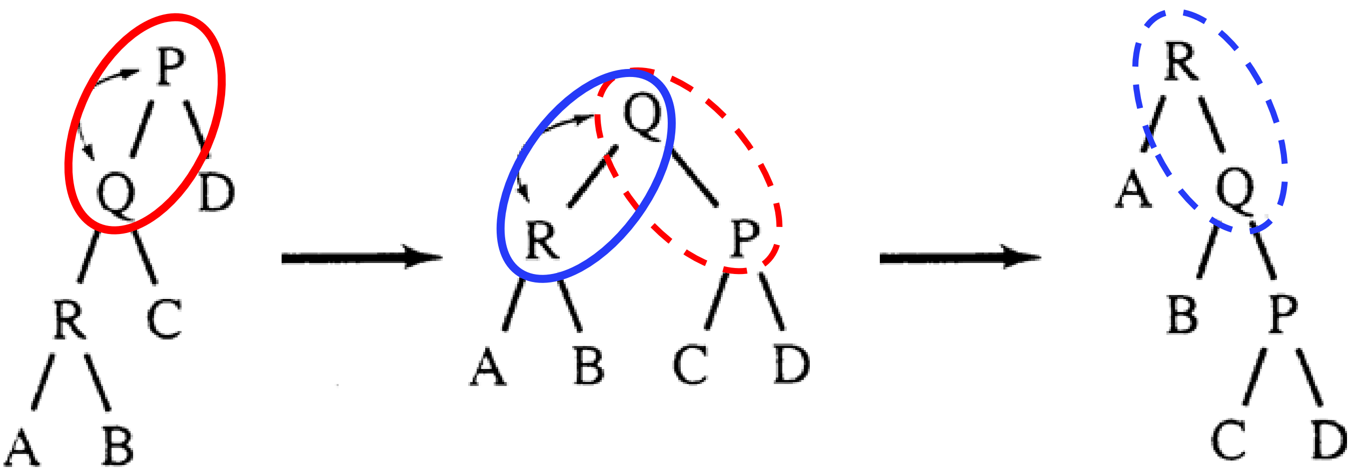



(ZigFromRight) (ZigFromLeft)

ZigZagFromLeft

# Zig-Zig operation

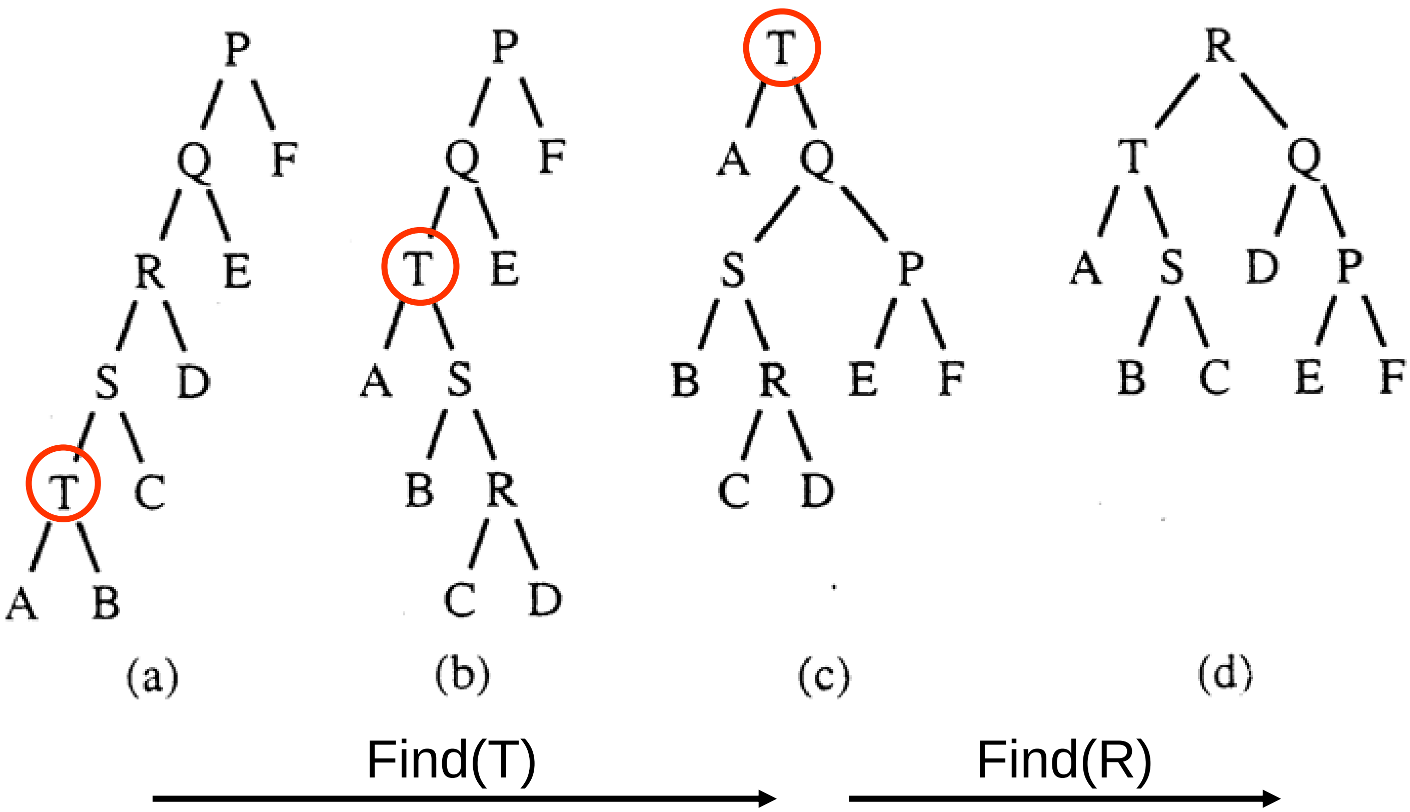
- "Zig-Zig" consists of two single rotations of the same direction (R is the node that was accessed)




(ZigFromLeft) (ZigFromLeft)

ZigZigFromLeft

# Decreasing depth - "autobalance"




- splaying not only moves the accessed node to the root,
- but also has the effect of roughly halving the depth of most nodes on the access path

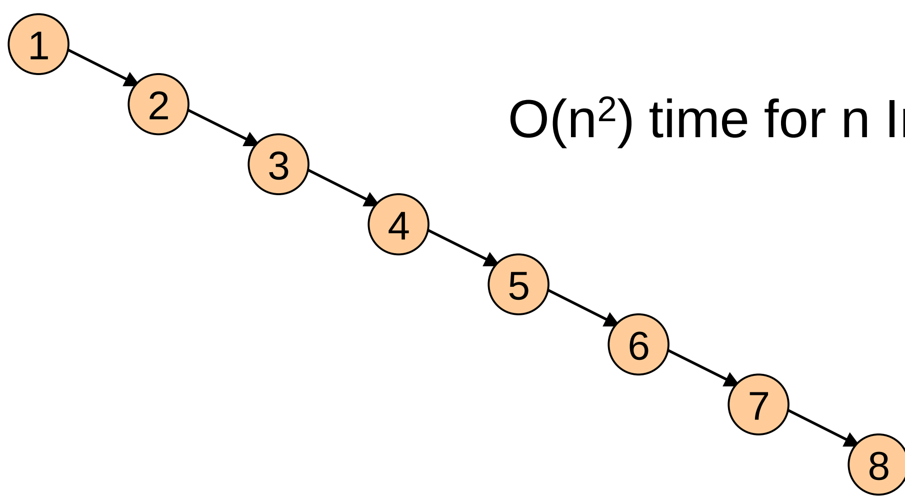
# Splay Tree Insert and Delete

- Insert x
  - Insert x as normal then splay x to root.

- Delete x
  - Splay x to root and remove it. (note: the node does not have to be a leaf or single child node like in BST delete.)  Two trees remain, right subtree and left subtree.
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.

# Example Insert

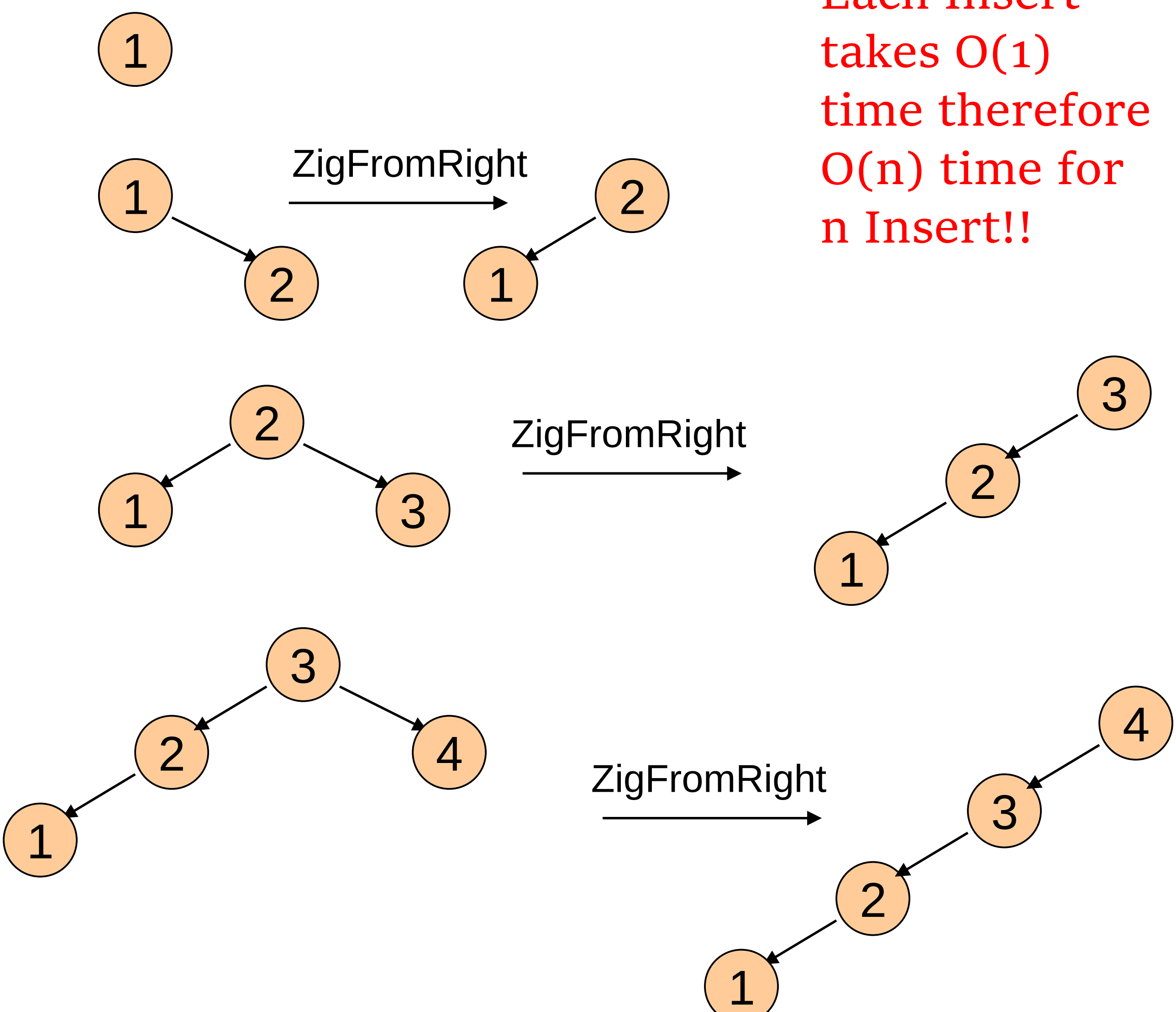
- Inserting in order 1,2,3,…,8
- Without self-adjustment




O($n^2$) time for n Insert

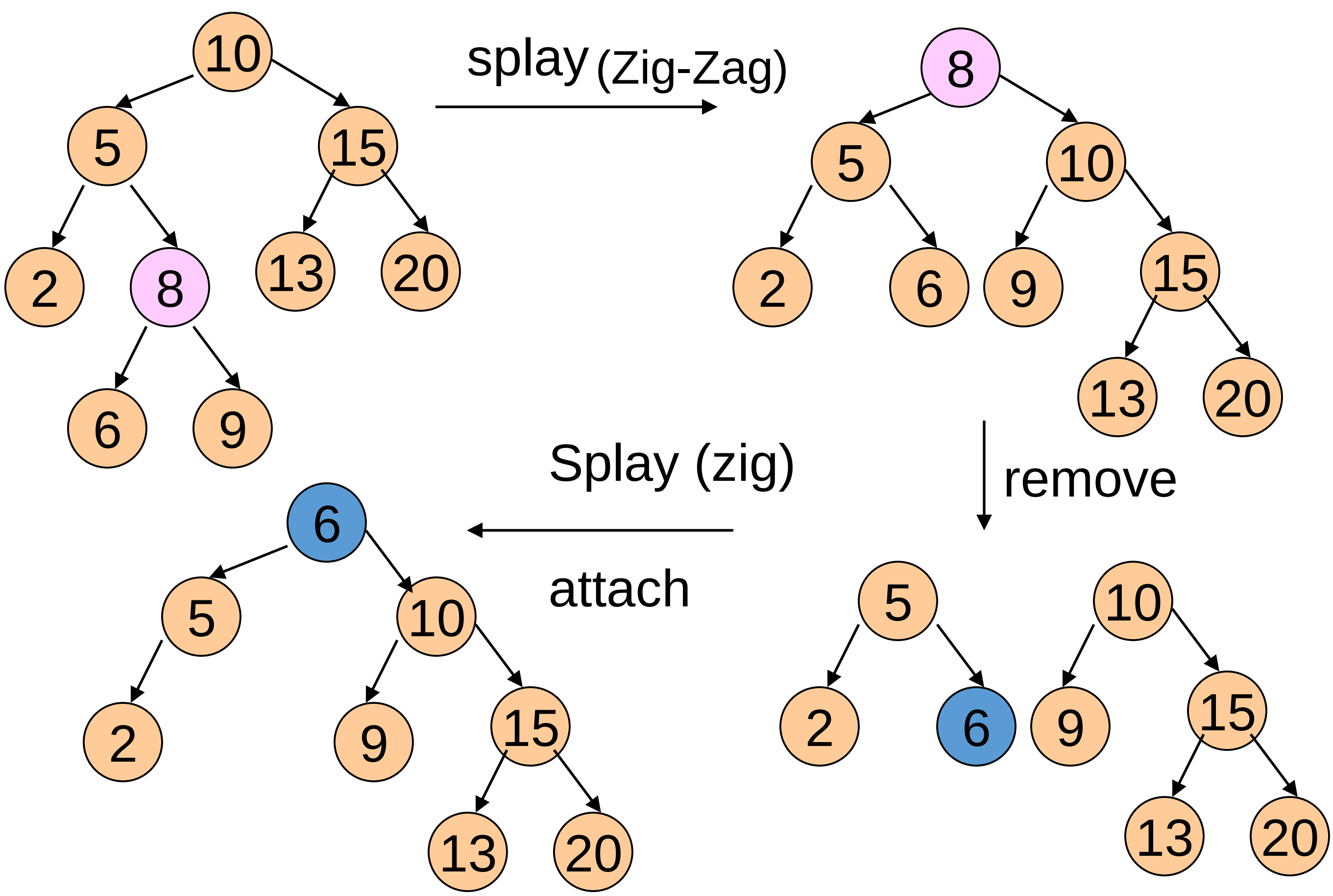
# Example Insert

- Inserting in order 1,2,3,…,8
- With self-adjustment




Each Insert takes O(1) time therefore O(n) time for n Insert!!

# Example Deletion



# Analysis of Splay Trees

- Splay trees tend to be balanced
  - M operations takes time O(M log N) for M $\geq$ N operations on N items. (proof is difficult)
  - Amortized O(log n) time.
- Splay trees have good “locality” properties
  - Recently accessed items are near the root of the tree.
  - Items near an accessed one are pulled toward the root.

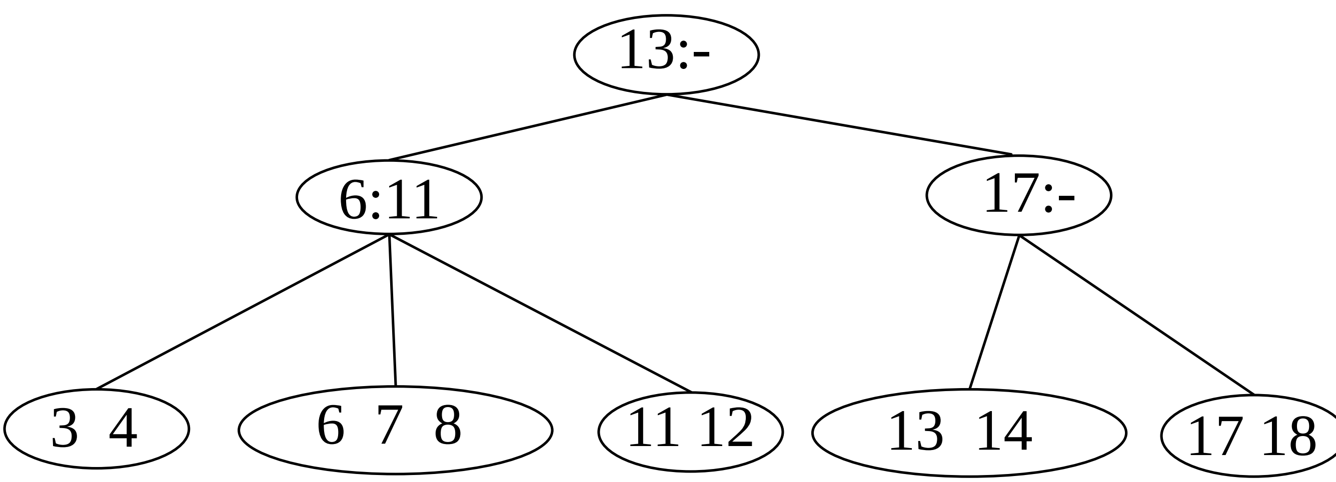
# Homework

- Homework 3-3
    - Textbook exercises 4.27, 4.28

# B-Trees

# Beyond Binary Search Trees: Multi-Way Trees

- Example: B-tree of order 3 has 2 or 3 children per node
- e.g. 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.

# B-Trees

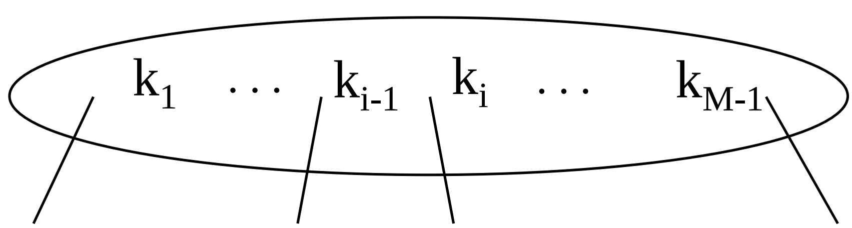
- A B-Tree of order M has the following properties:
  - The root is either a leaf or has between 2 and M children.
  - All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
  - All leaves are at the same depth.

- All data records are stored at the leaves.
- Internal nodes have “keys” guiding to the leaves.
- Leaves store between $\lceil L/2 \rceil$ and L data records, where L can be equal to M (default) or can be different.

# B-Tree Details

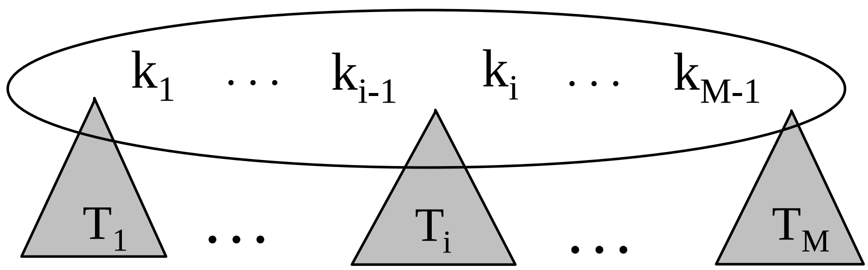
Each (non-leaf) internal node of a B-tree has:

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




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

# 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 keys $k_{i-1}$ and $k_i$

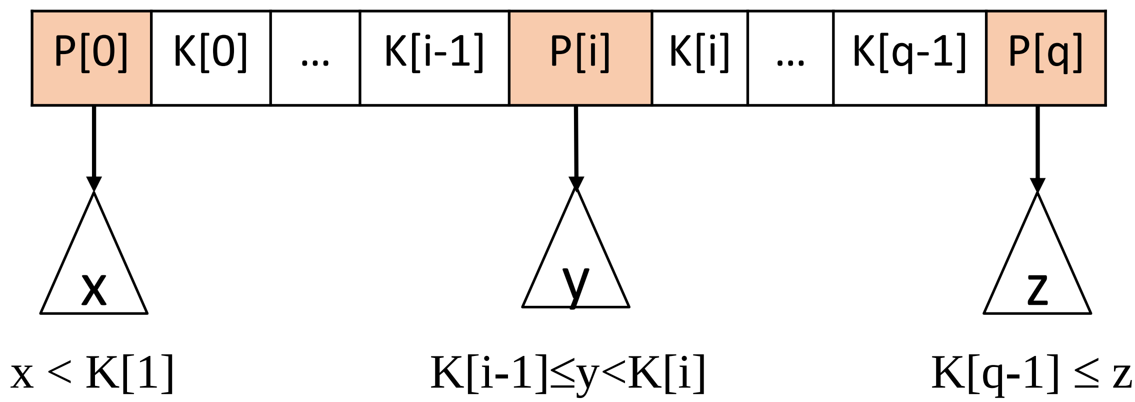
i.e. $k_{i-1} \leq T_i < k_i$ , $k_{i-1}$ is the smallest key in $T_i$

- All keys in first subtree $T_1 < k_1$

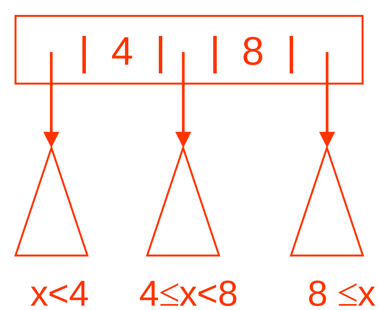
- All keys in last subtree $T_M \geq k_{M-1}$

# Properties of B-Trees

## B-Tree Nonleaf Node




- The Ks are keys
- The Ps are pointers to subtrees

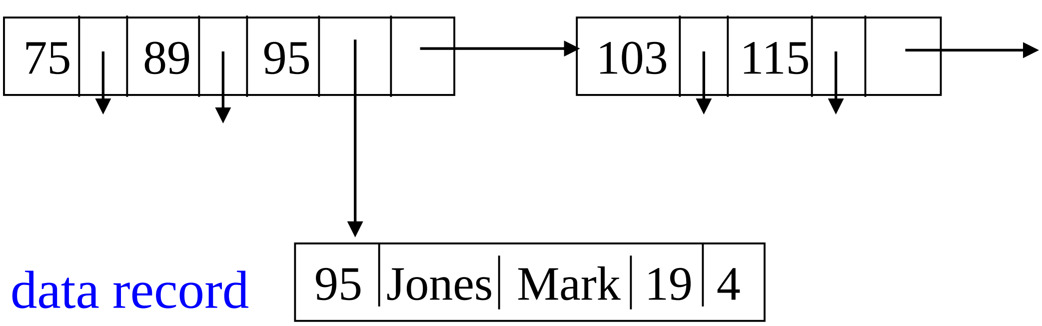


# Properties of B-Trees

## B-Tree leaf Node (B+tree)

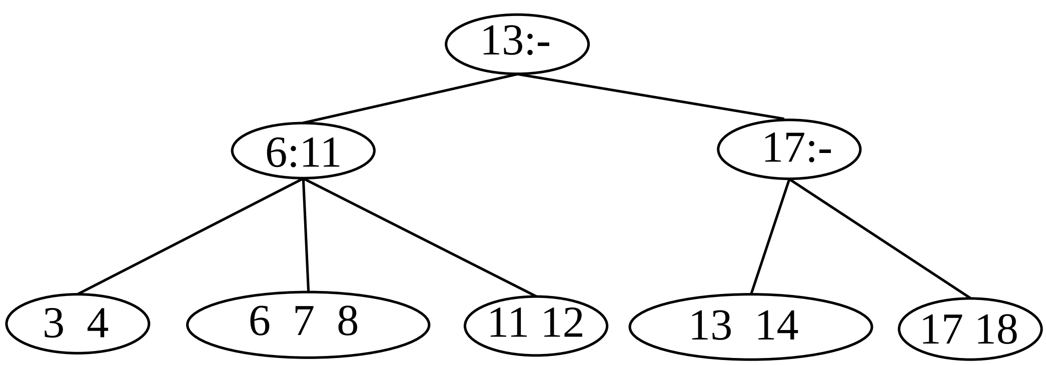
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| k[0] | R[0] | … | k[i-1] | R[i-1] | … | K[q] | R[q] | Next |

- The Ks are keys (assume unique).
- The Rs are pointers to records with those keys.
- The Next link points to the next leaf in key order (B+-tree).



# 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

# Searching in B-trees

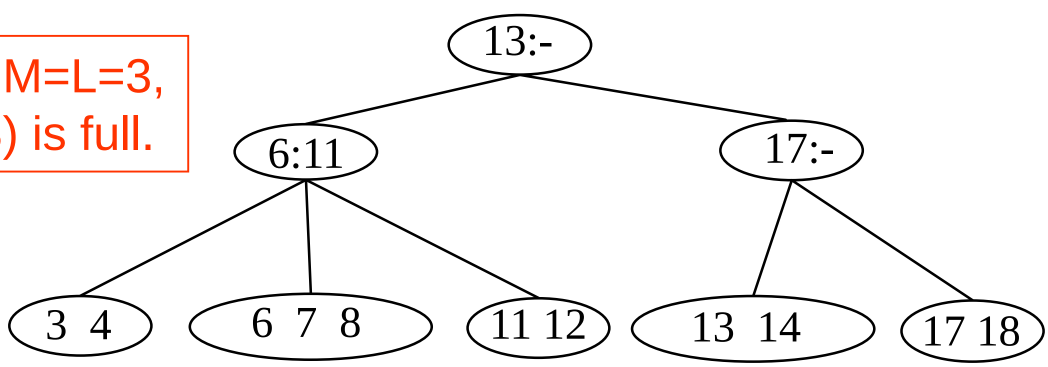
- Searching a B-Tree T for a Key Value K

```
Find(ElementType K, Btree T){
  B = T;
  while (B is not a leaf){
   find the Pi in node B that points to the proper
subtree that K will be in;

   B = Pi;
}

/* Now we're at a leaf */

if key K is the jth key in  leaf  B,
   use the jth record pointer to find the
   associated record;
else /* K is not in leaf B */ report failure;
}
```

# Inserting into 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

Assume M=L=3, so (6 7 8) is full.



# Inserting into B-Trees

// Inserting a New Key in a B-Tree of Order M (and L=M)

Insert(ElementType K, Btree B)
{
  find the leaf node LB of B in which K belongs;
  if notfull(LB) insert K into LB;
  else {
    split LB into two nodes LB and LB2 with
    $j = \lfloor (M+1)/2 \rfloor$ keys in LB and the rest in LB2;

LB → | K[1] R[1] . . . K[j] R[j] | → LB2 → | K[j+1] R[j+1] . . . K[M+1] R[M+1] | →

      if ( IsNull(Parent(LB)) )
          CreateNewRoot(LB, K[j+1], LB2);
      else
          InsertInternal(Parent(LB), K[j+1], LB2);
      }
}

# Inserting into B-Trees

// Inserting a (Key,Ptr) Pair into an Internal Node

If the node is not full, insert them in the proper place and return.

If the node is already full (M pointers, M-1 keys), find the place for the new pair and split the adjusted (Key,Ptr) sequence into two internal nodes with

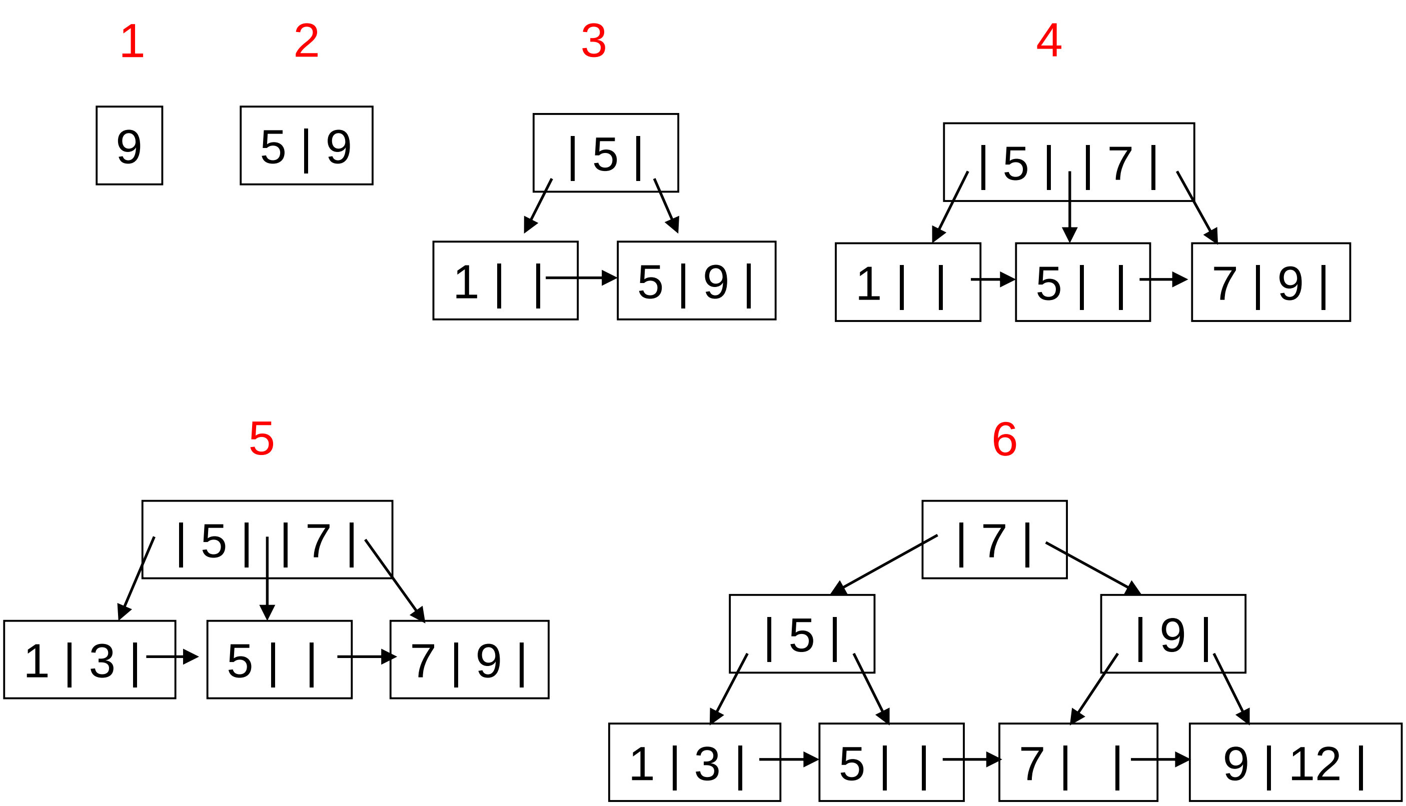
$j = \lfloor (M+1)/2 \rfloor$ pointers and j-1 keys in the first,

the next key is inserted in the node's parent, and the rest in the second of the new pair.

# Example of Insertions

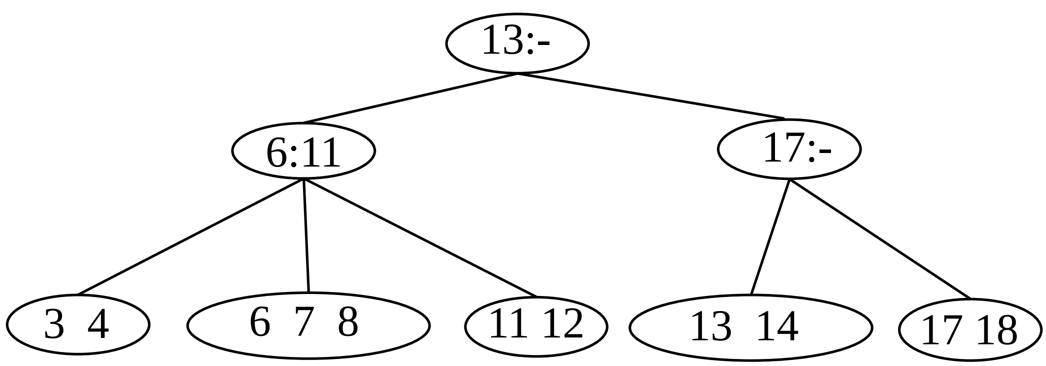
Insertions into a B+tree with M=3, L=2

Insertion Sequence: 9, 5, 1, 7, 3,12



# Deleting From B-Trees

- Delete X : Do a find and remove from leaf
  - Leaf underflows – borrow from a neighbor
    - E.g. 11
  - Leaf underflows and can't borrow – merge nodes, delete parent
    - E.g. 17



# 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. But M is small compared to N.
  - Total time to find an item is O(depth*log M) = O(log N)

# Run Time Analysis of B-Tree Operations

- How Do We Select the Order M?
  - In internal memory, small orders, like 3 or 4 are fine.
  - On disk, we have to worry about the number of disk accesses to search the index and get to the proper leaf.

Rule: Choose the largest M so that an internal node can fit into one physical block of the disk.

- This leads to typical M's between 32 and 256
- And keeps the trees as shallow as possible.

# Summary of Search Trees

- Problem with Binary 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.
  - Excellent for indexes in database systems.

# Homework

- Homework 3-4
  - Show the updated B+-Tree with order 4 that results from inserting the records U and R in order.
  - Assume that the leaf nodes are capable of storing up to 3 records