Splay Trees and B-Trees

CSE 373

Data Structures

Lecture 9

Readings

- Reading
  - › Sections 4.5-4.7

Self adjusting Trees

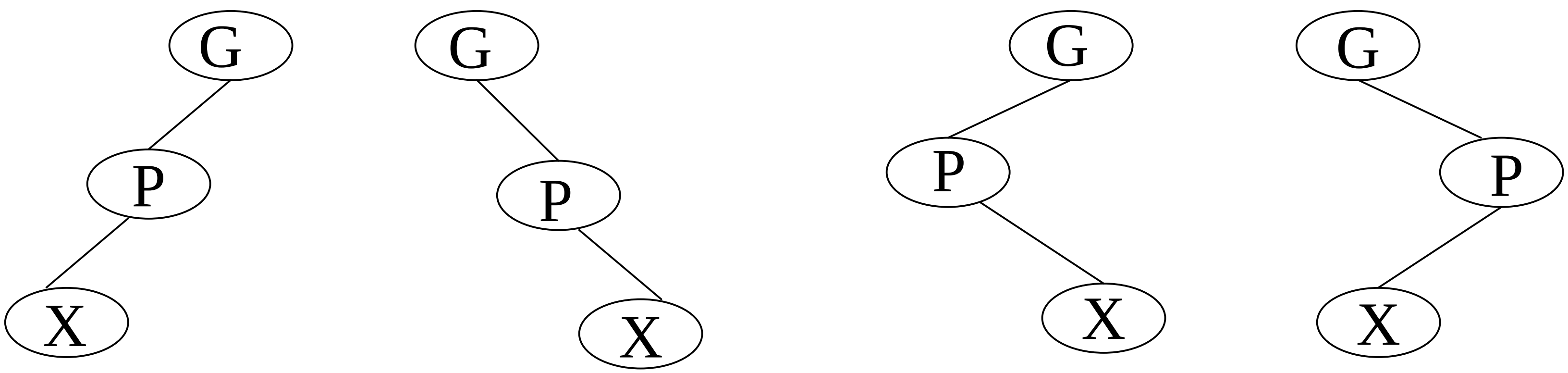
- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed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

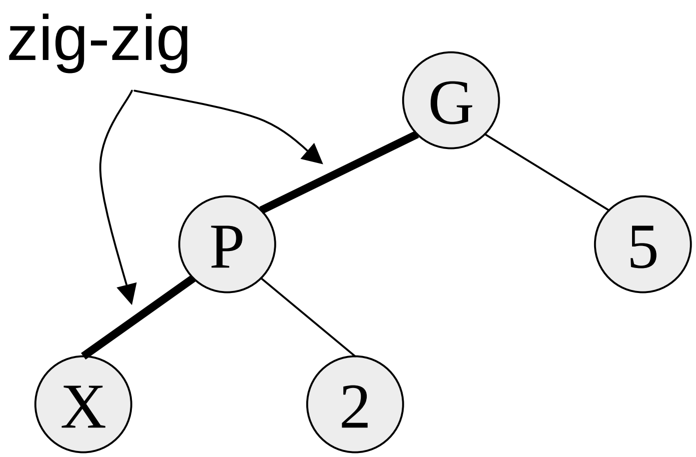
- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.

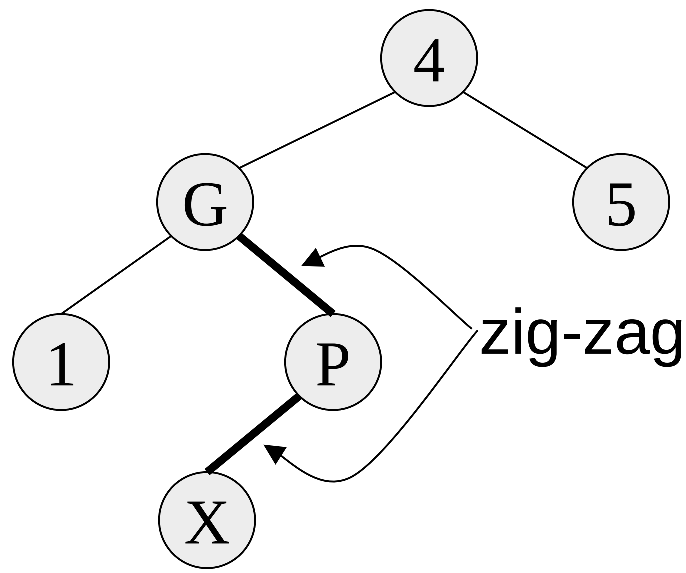


Zig-Zig and Zig-Zag

Parent and grandparent in same direction.

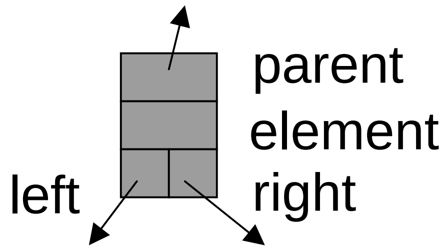
Parent and grandparent in different directions.





Splay Tree Operations

1. Helpful if nodes contain a parent pointer.

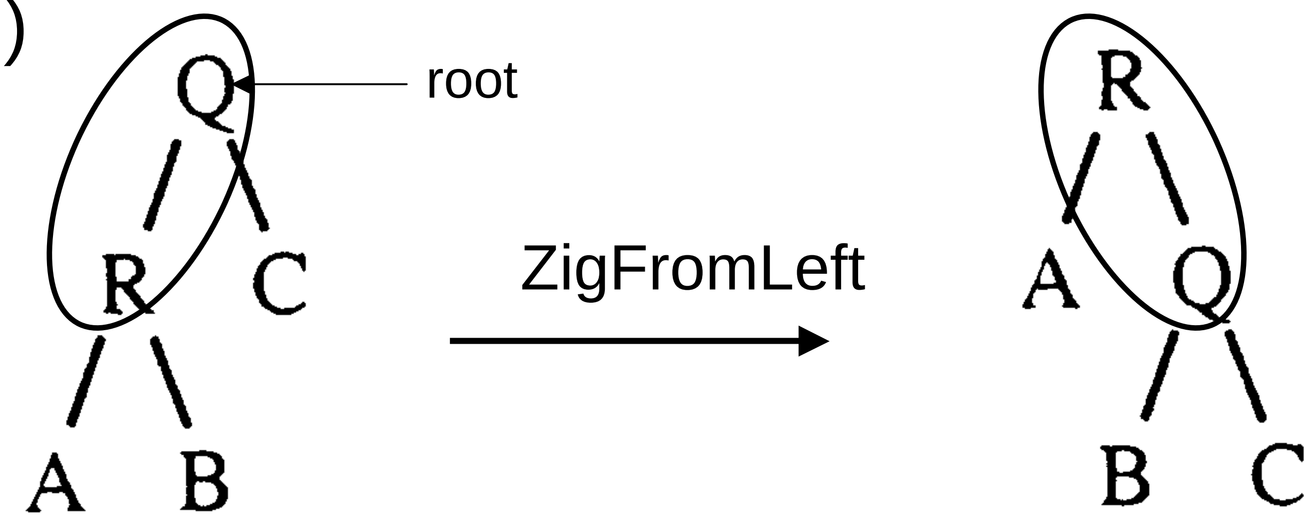


2. When X is accessed, apply one of six rotation routines.

- 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 (root)

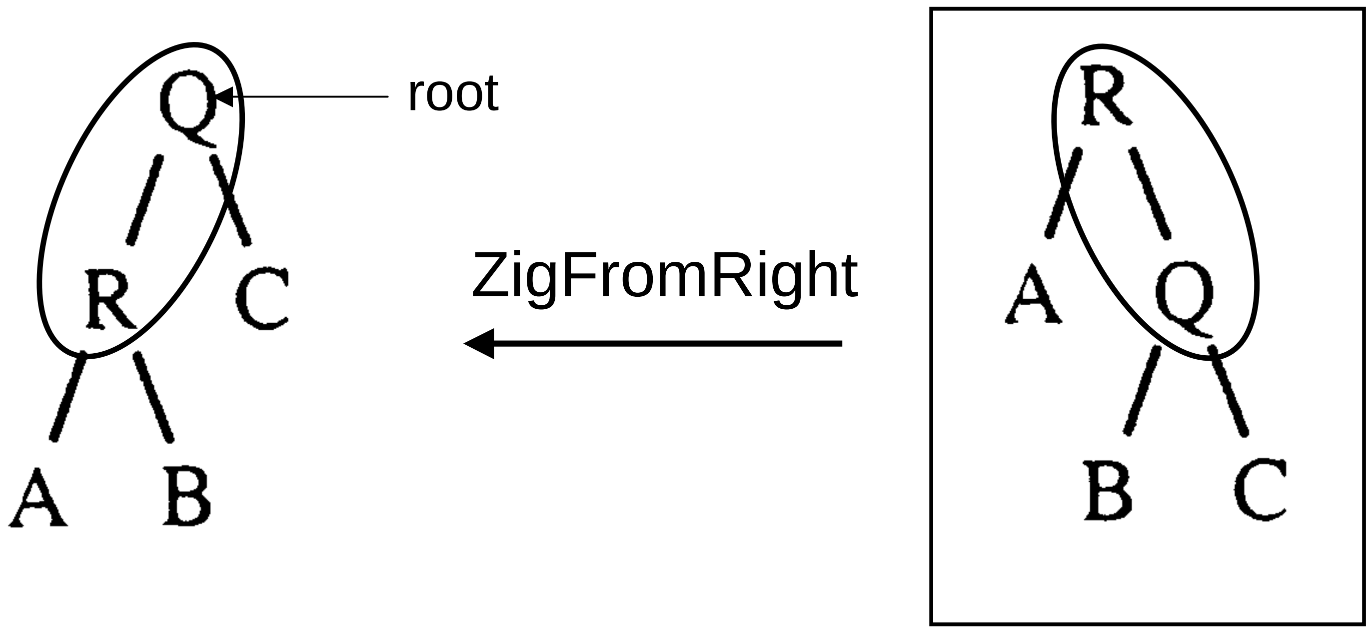
- "Zig" is just a single rotation, as in an AVL tree
- Let R be the node that was accessed (e.g. using Find)



- ZigFromLeft moves R to the top $\rightarrow$faster access next time

Zig at depth 1

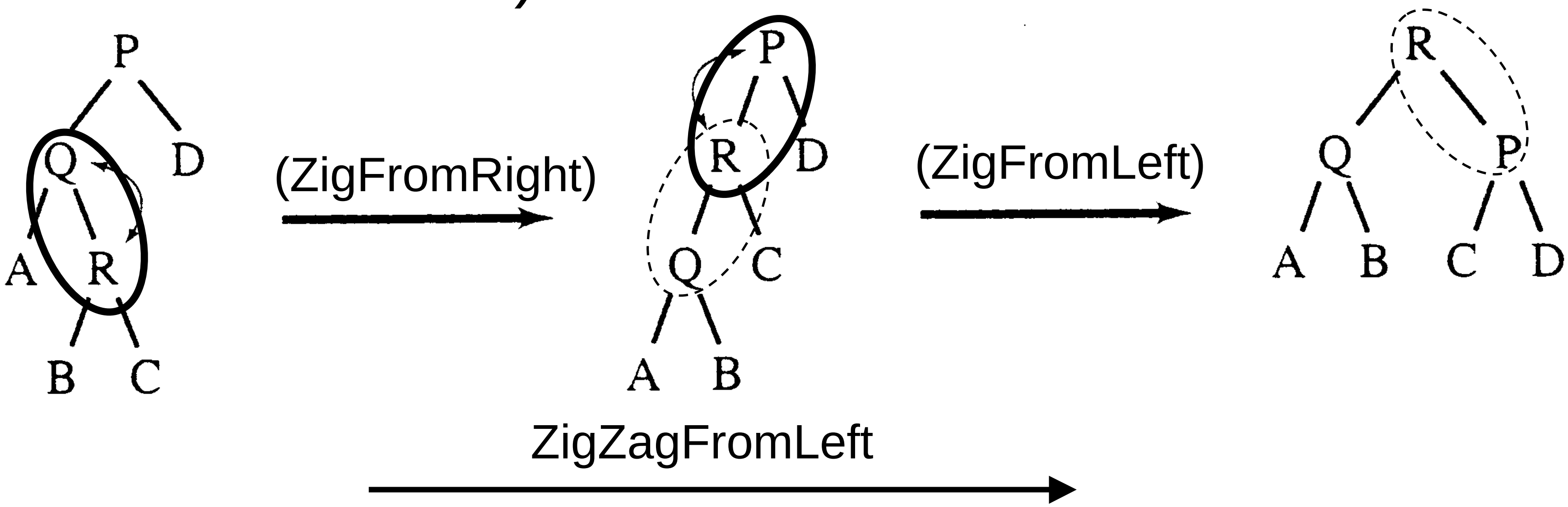
- Suppose Q is now accessed using Find



- ZigFromRight moves Q back to the top

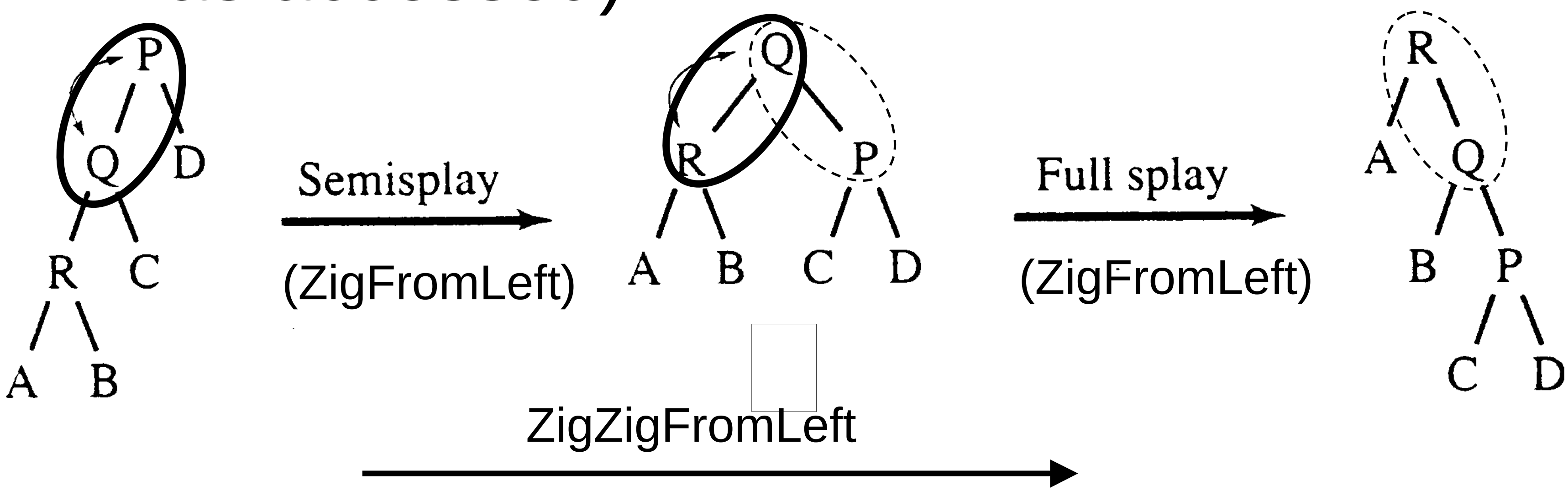
Zig-Zag operation

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

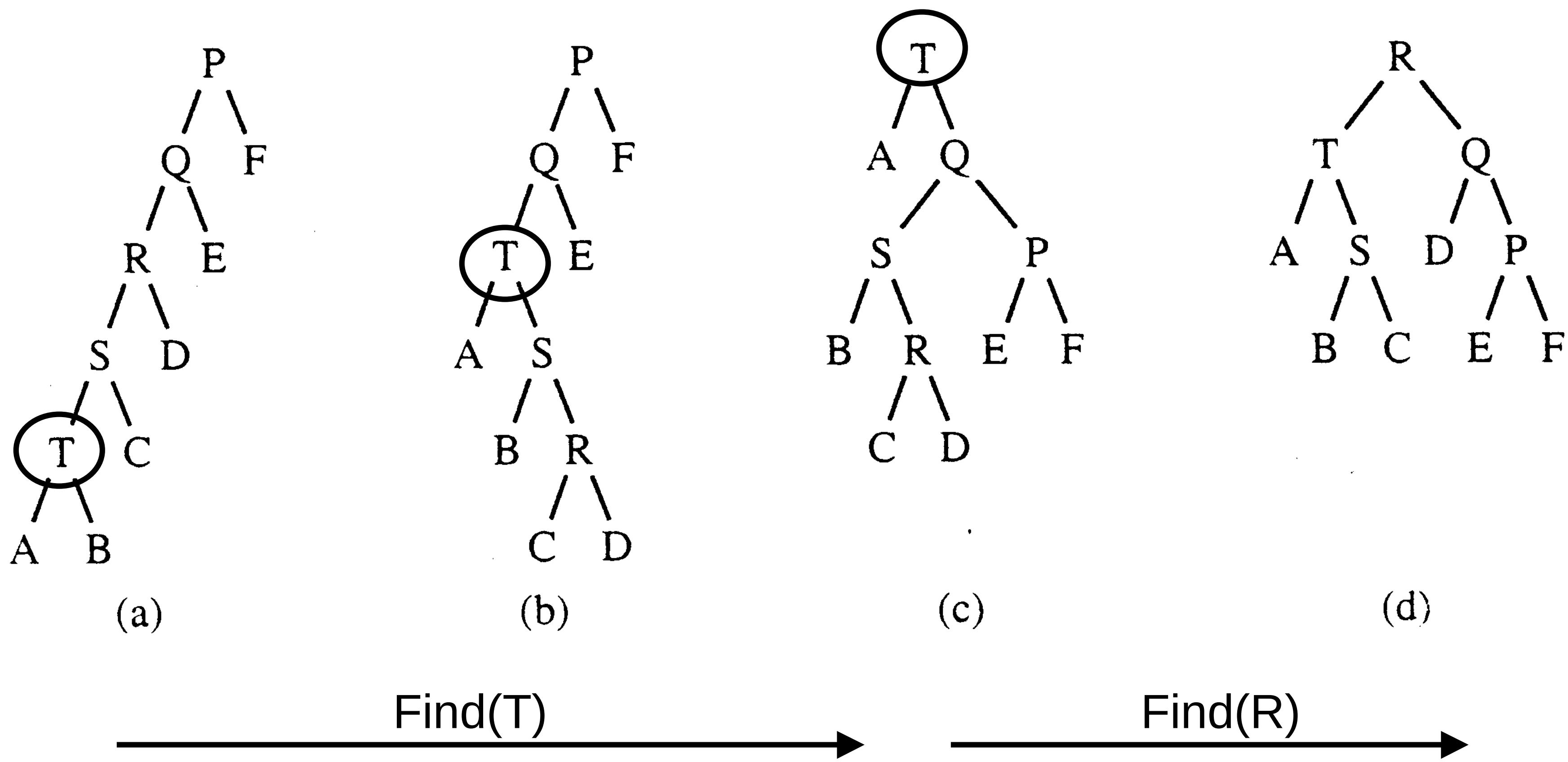


Zig-Zig operation

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



Decreasing depth - "autobalance"

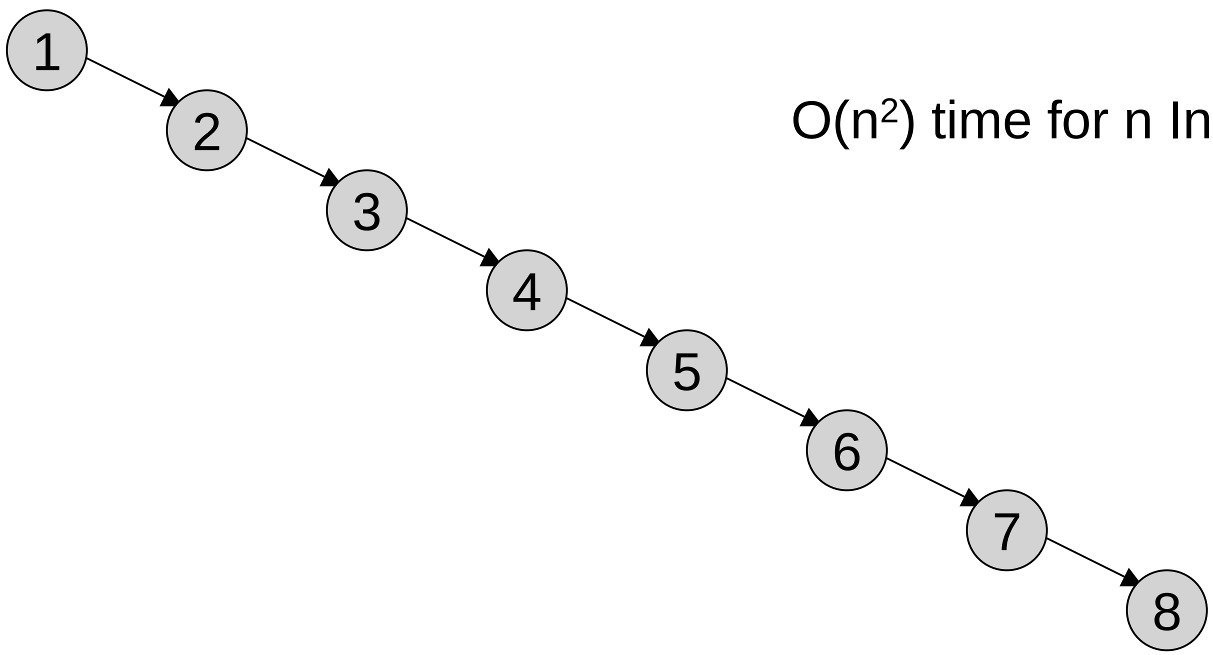


Splay Tree Insert and Delete

- 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



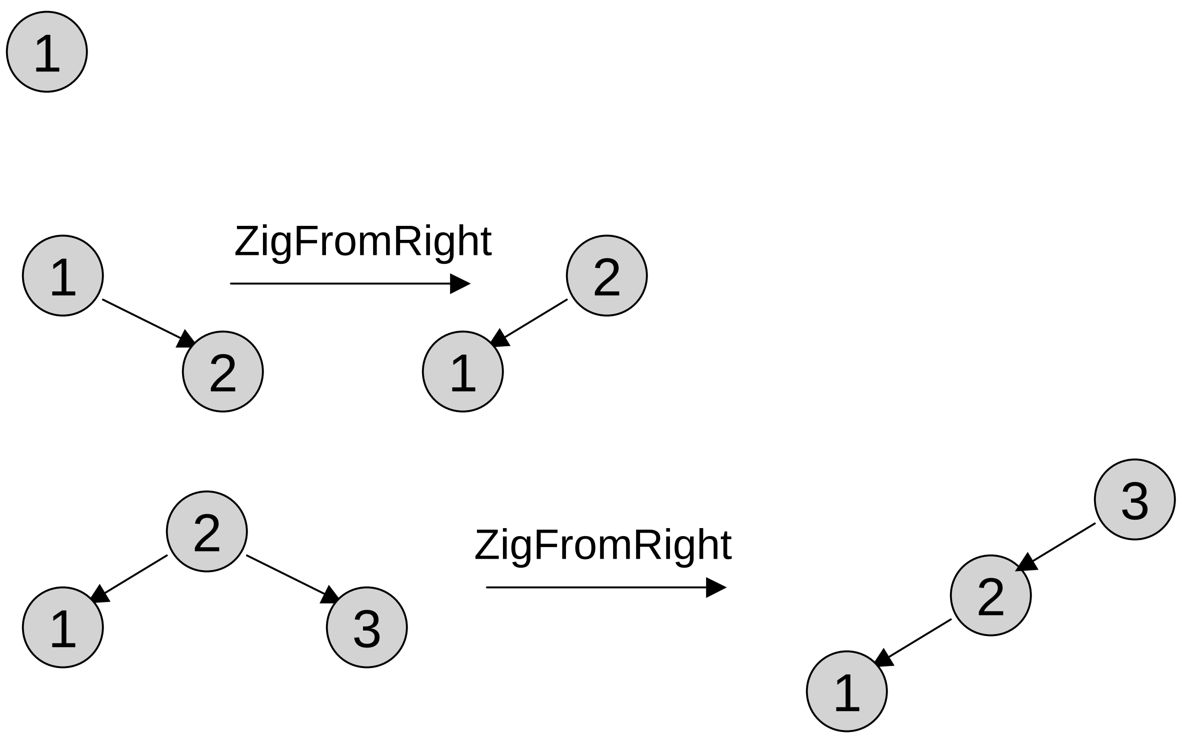
$O(n^2)$ time for n Insert

With Self-Adjustment

1

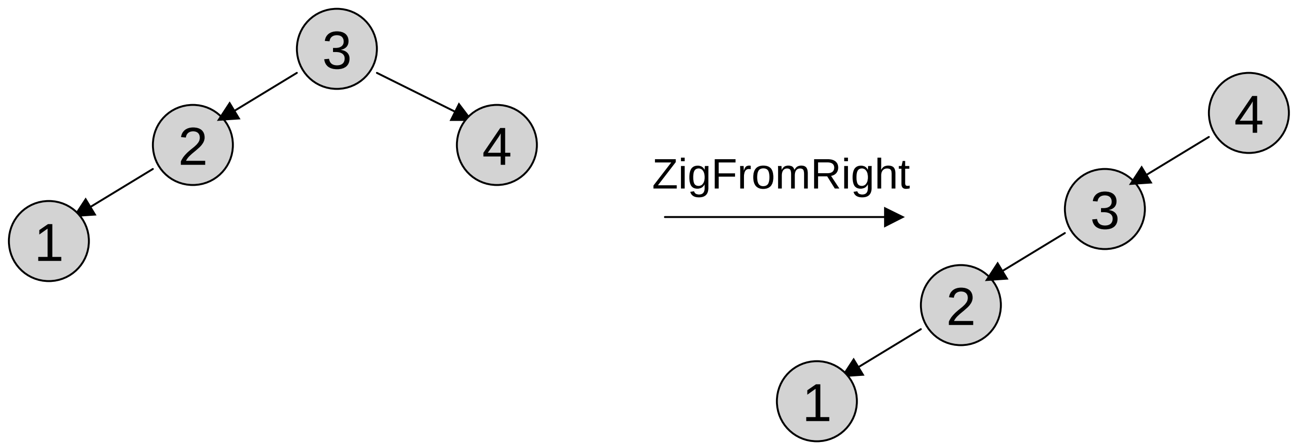
2

3



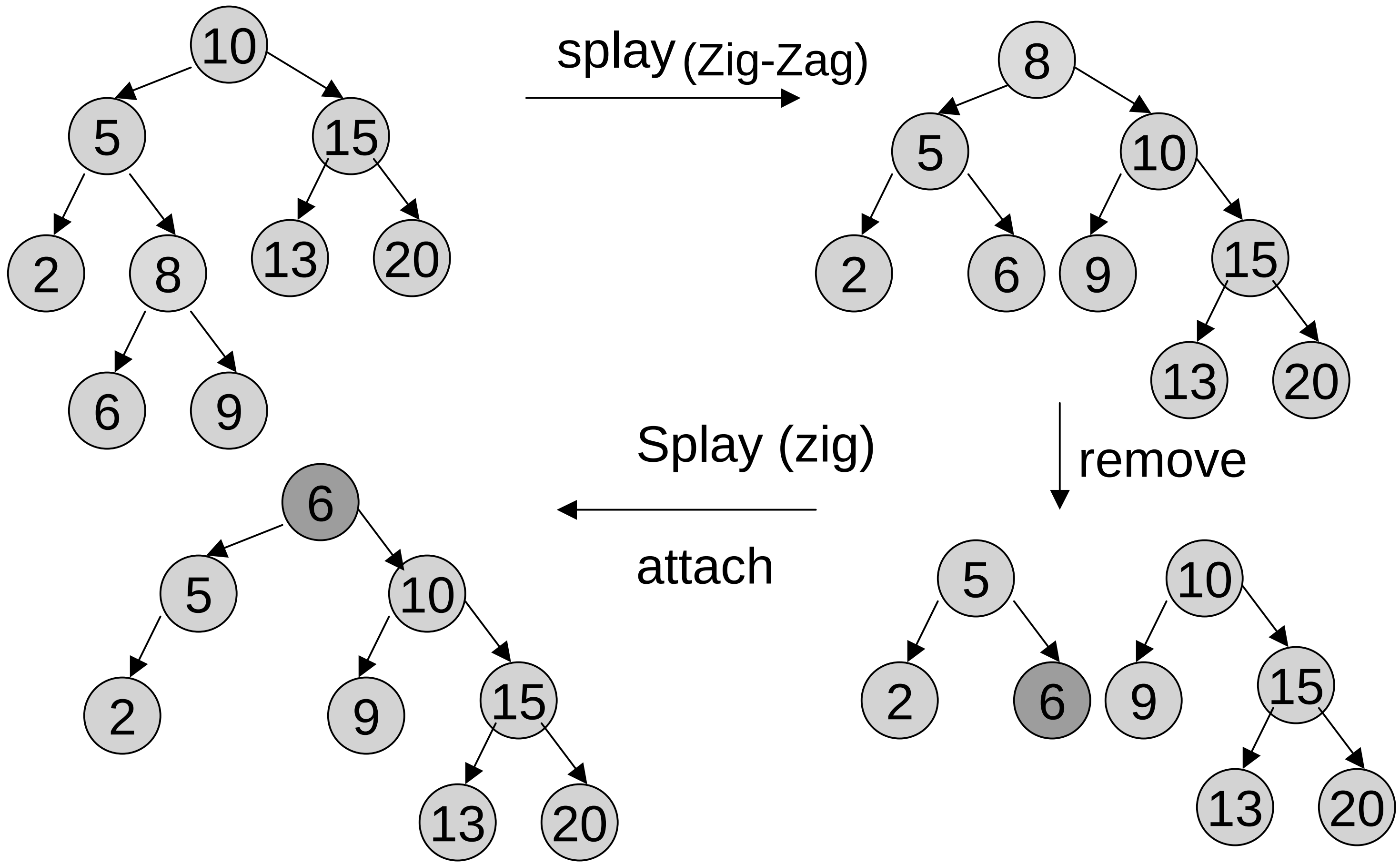
With Self-Adjustment

4



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

Example Deletion

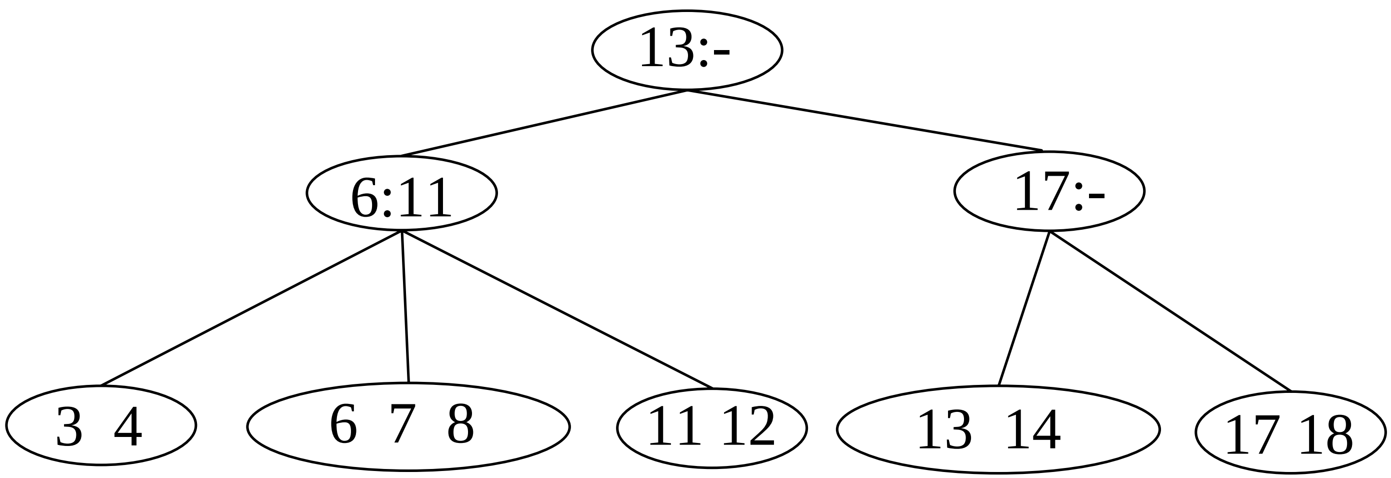


Analysis of Splay Trees

- 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.

Beyond Binary Search Trees: Multi-Way Trees

- Example: 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:

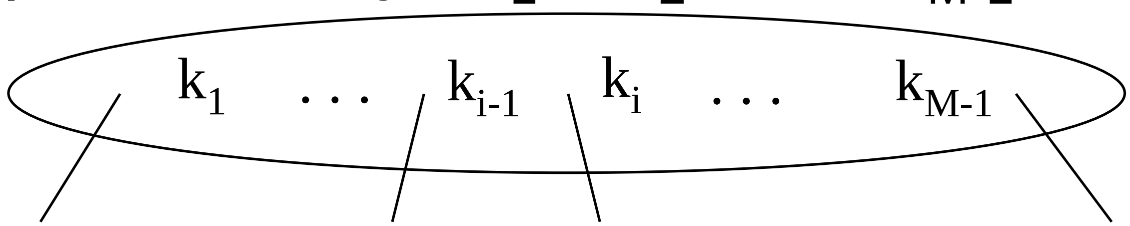
1. The root is either a leaf or has between 2 and M children.
2. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
3. All leaves are at the same depth.

All data records are stored at the leaves. Internal nodes have “keys” guiding to the leaves. Leaves store between $\lceil M/2 \rceil$ and M data records.

B-Tree Details

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

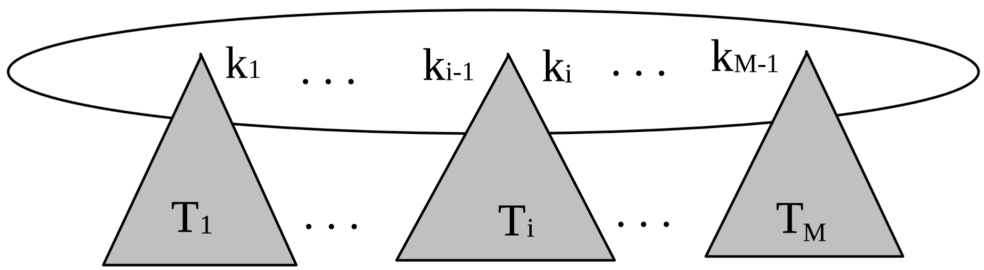
- › Between $\lceil M/2 \rceil$ and M children.
- › up to M-1 keys $k_1 < k_2 < ... < k_{M-1}$



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:

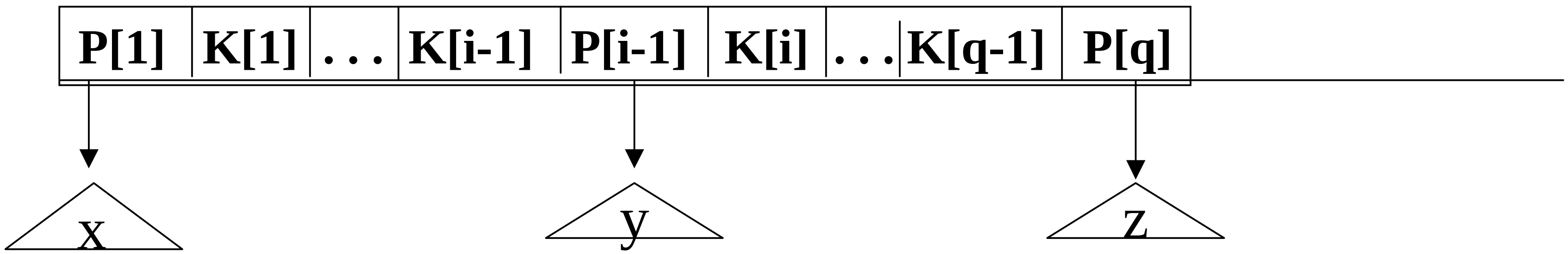
all keys in T_i must be between keys k_{i-1} and k_i

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 $T_M \geq k_{M-1}$

DS.B.13

B-Tree Nonleaf Node



x < K[1]   K[i-1]≤y<K[i]   K[q-1] ≤ z

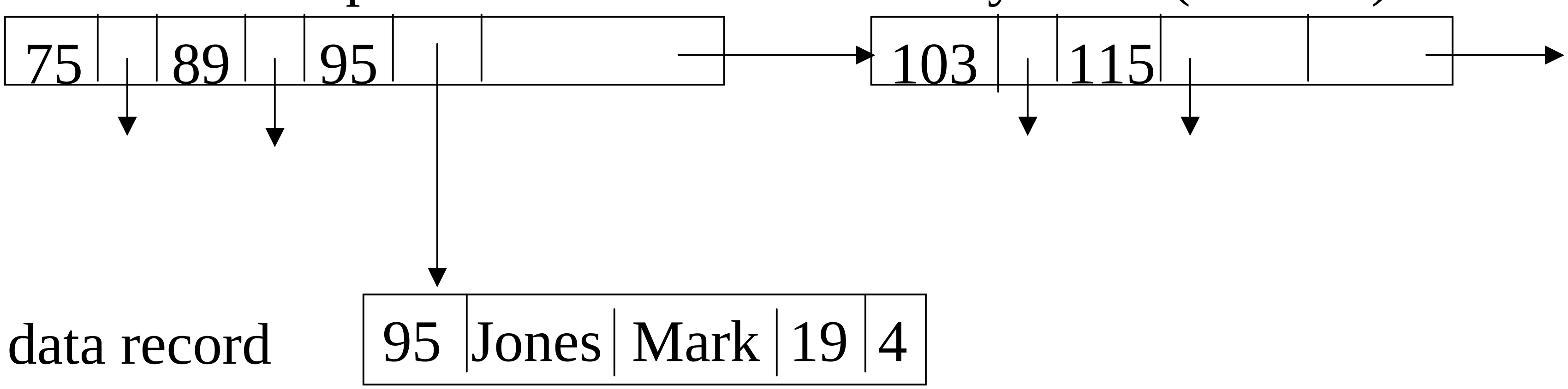
- The Ks are keys

- The Ps are pointers to subtrees.

Leaf Node Structure

K[1]	R[1]	. . .	K[q-1]	R[q-1]	Next

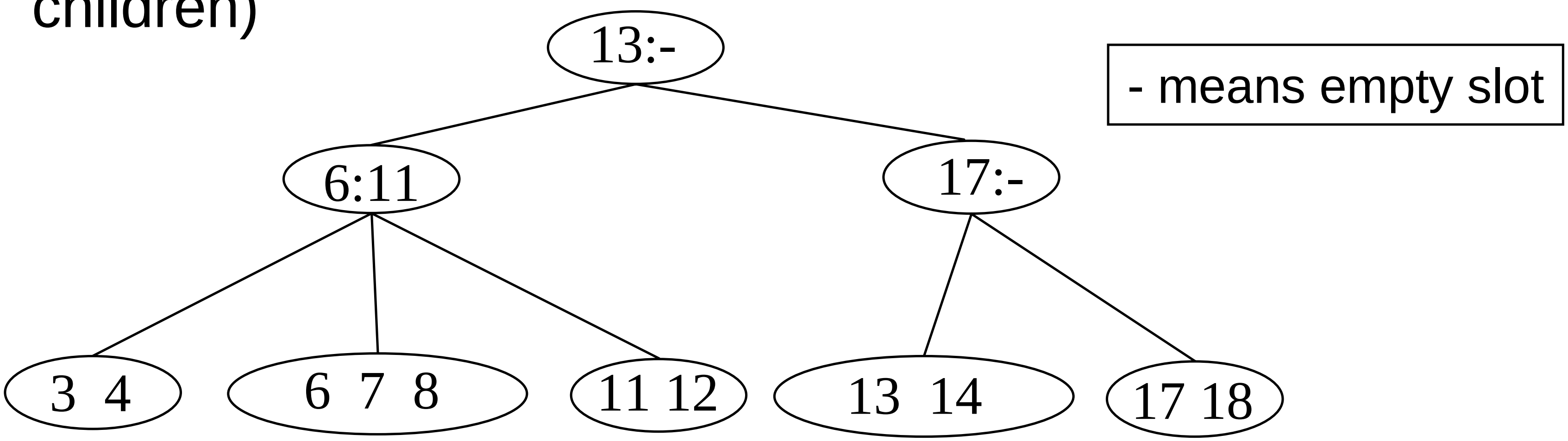
- The Ks are keys (assume unique).
- The Rs are pointers to records with those keys.
- The Next link points to the next leaf in key order (B+-tree).



data record

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

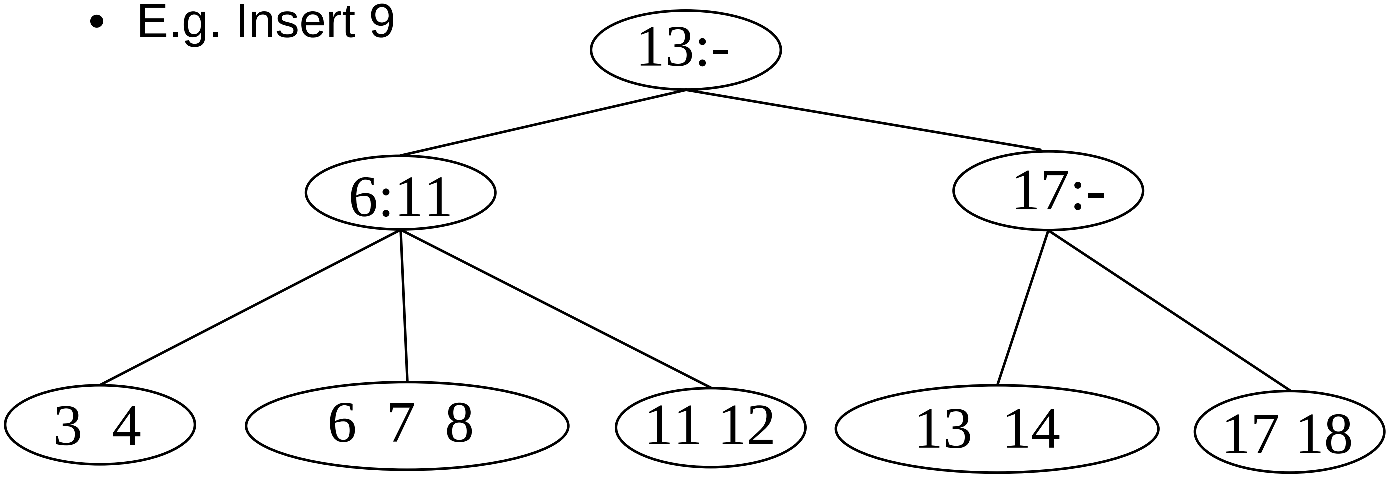
Searching a B-Tree T for a Key Value K

```
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;
}
```

How would you search for a key in a node?

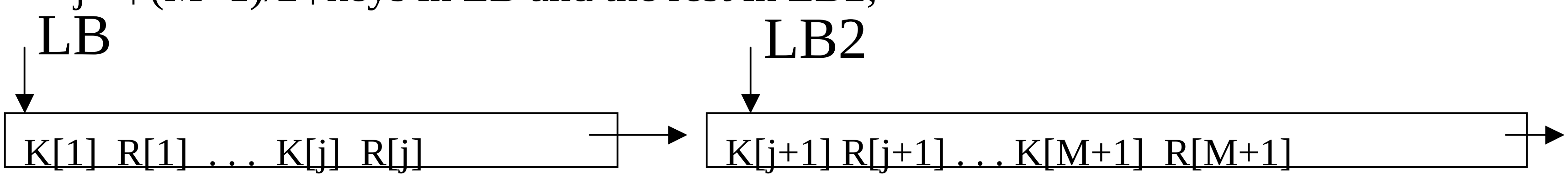
Inserting into B-Trees

- 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



Inserting a New Key in a B-Tree of Order 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 = ⌈(M+1)/2⌉ keys in LB and the rest in LB2;
```




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

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

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

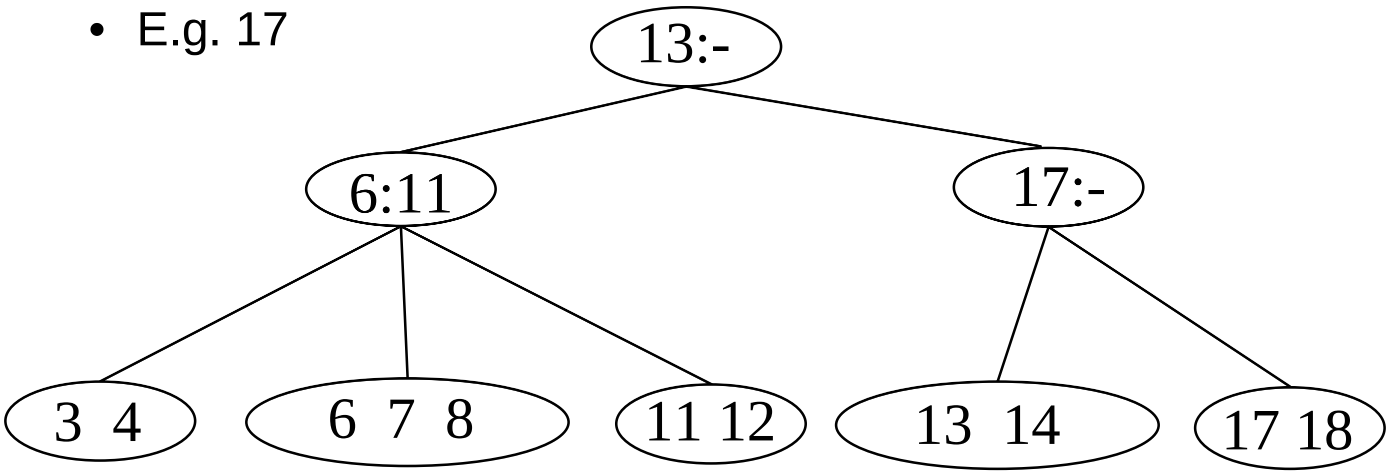
If the node is already full (M pointers, M-1 keys), find the place for the new pair and split the adjusted (Key,Ptr) sequence into two internal nodes with

$j = \lfloor (M+1)/2 \rfloor$ 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.

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)

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.