Search Trees

What are search trees?

- Allow efficient searching of ordered data
- Implement Ordered Dictionary ADT
- Provide flexible mechanism for storing and retrieving data

Binary Search Tree

- Each node stores a key-element pair
- key(leftsubtree) <= key(node) <= key(rightsubtree)
- Trees in Goodrich/Tamassia store elements only at internal nodes
  - I generally store elements at all nodes

Tree Search

```
TreeSearch(k, node) {
if ((k < key(node)) &&
   (leftChild(node) != null))
     return TreeSearch(k, leftchild(node));
else if (k > key(node)) &&
     (rightchild(node) != null))
       return TreeSearch(k, rightchild(node));
else
  return node; }
```


Analysis of TreeSearch

- At each node, perform O(1) work
- Maximum nodes visited is *h*, the height of tree
- Total running time is thus $O(h)$

Inserting into Binary Tree

- Call TreeSearch on root to find appropriate parent node
  - Call again using a child if key already exists
- Parent node will be external
- Insert element as new child of parent

Also takes $O(h)$

Removing from Binary Tree

Call TreeSearch to locate node for removal

If node is external, just remove node

If node has one child, replace node with child

If node has two children

- **Find smallest element greater than node (will have 0 or 1 children)**
- **Replace element with that node**

Again, $O(h)$

AVL Trees

Binary search trees which maintain $O(\log n)$ height

Maintain *height balance property*

- **Heights of children differ by at most <u>1</u>**
- **Local property to maintain, but guarantees global property of overall height**

Analyzing AVL height

n(h) : minimum nodes for AVL tree of height h

Base conditions

$n(0) = 1$; $n(1) = 2$

Recurrance relation

$n(h) = 1 + n(h-1) + n(h-2) > 2*n(h-2)$

$n(h) > 2*n(h-2) > 4*n(h-4) > 8*n(h-6)$, etc.

$n(h) > 2^i*n(h-2i)$

Set i to achieve base condition

$h-2i=1 \rightarrow i=(h-1)/2$

$n(h) > 2^{(h-1)/2}*n(1) = 2^{(h-1)/2}*2 = 2^{(h-1/2)+1}$

Bounding h

- $\log n(h) > (h-1)/2 + 1$
- $h < 2(\log n(h) - 1) + 1 = 2\log n(h) - 1 = O(\log n)$

Inserting with balanced height

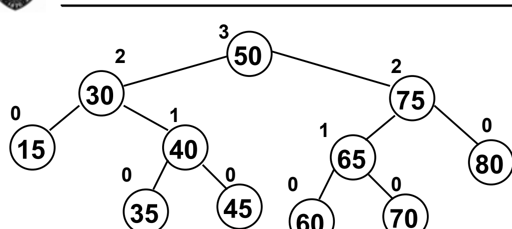
Insert node into binary search tree as usual

- **Increases height of some nodes along path to root**

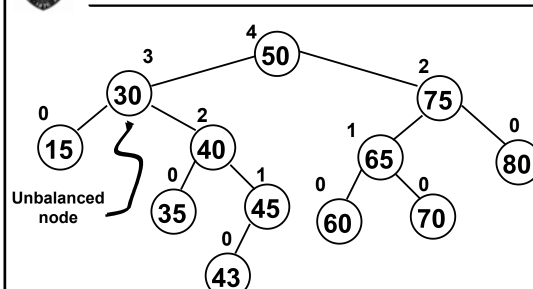
Walk up towards root

- **If unbalanced height is found, restructure unbalanced region with *rotation* operation**

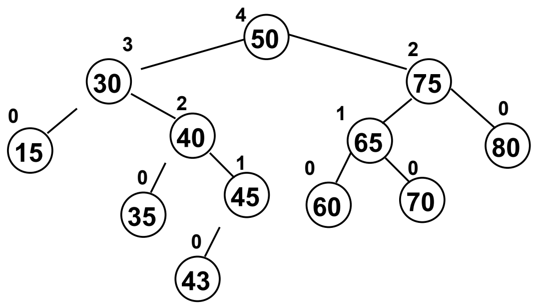
Balanced Tree



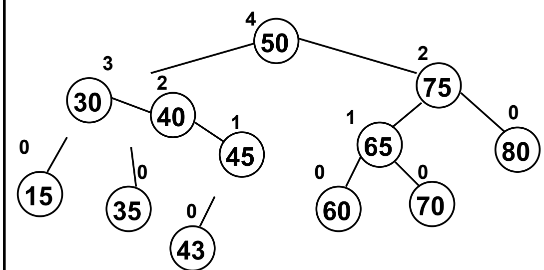
Insert (case 1)



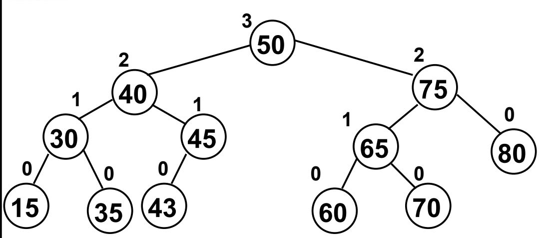
“Rotate Left” . . .



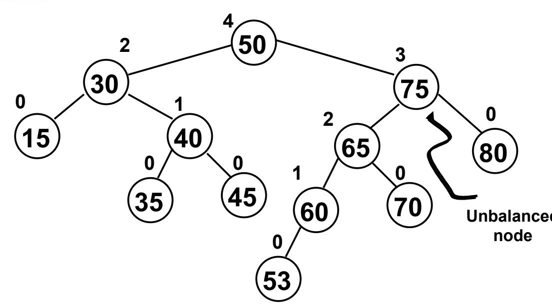
“Rotate Left” . . .



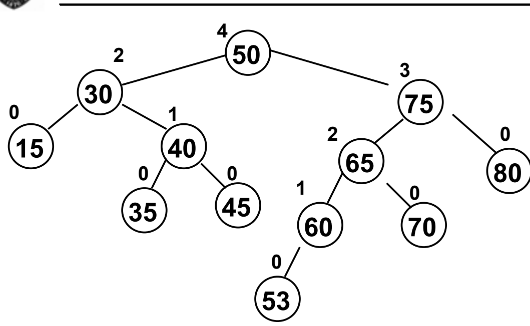
“Rotate Left” - Balanced!



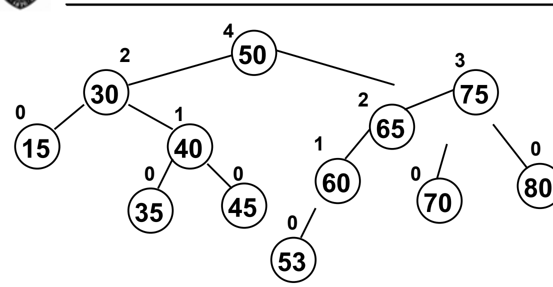
Insert (case 2)



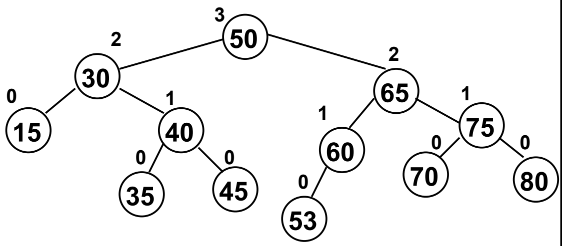
“Rotate Right”. . .



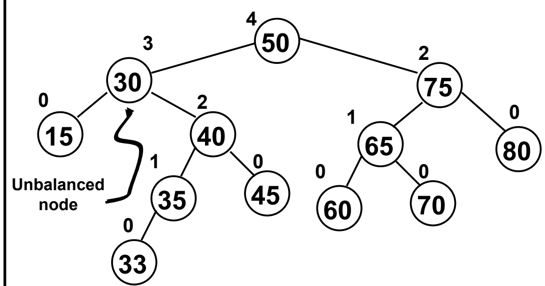
“Rotate Right”. . .



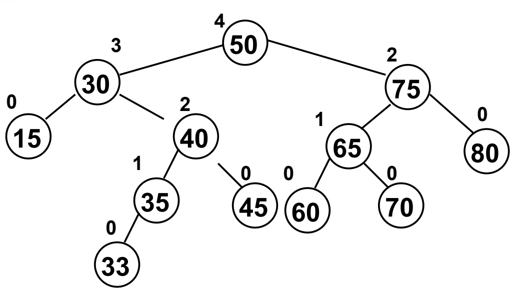
“Rotate Right” - Balanced!



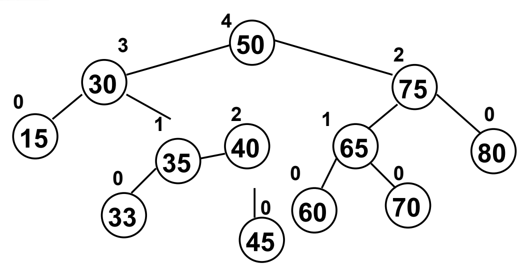
Insert (case 3)



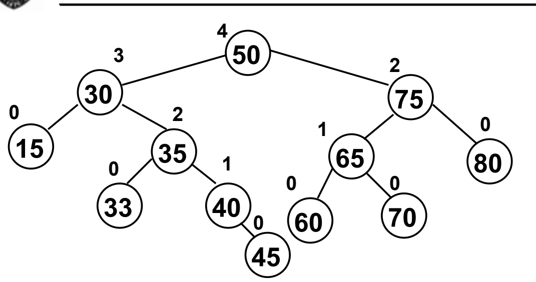
“Double Rotation Right-Left” - Right . . .



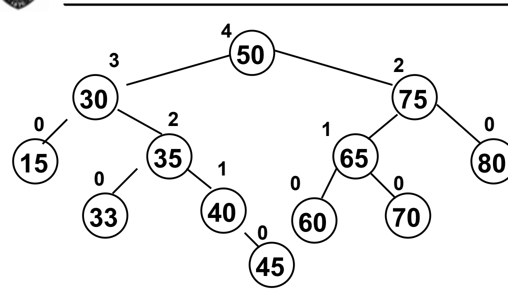
“Double Rotation Right-Left” - Right . . .



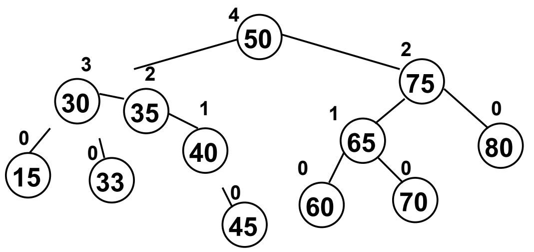
“Double Rotation Right-Left” - Right . . .done!



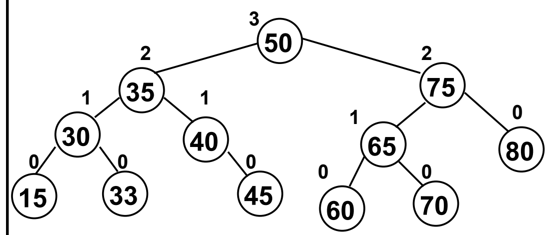
“Double Rotation Right-Left” - Left . . .



“Double Rotation Right-Left” - Left . . .



“Double Rotation Right-Left” - Balanced!



Restructure Procedure

Consider the first unbalanced node encountered (walking upward) and its two descendants along that path

Sort them in increasing order and label as *a*, *b*, and *c*

Place *b* as the parent of *a* and *c* where the unbalanced node was

Hook up the (up to) 4 subtrees as the appropriate children of *a* and *c*

Analyzing Insert

Upward traversal with height recomputation takes $O(h) = O(\log n)$

Restructure takes $O(1)$

The restructure always reduces the height of the unbalanced node

- **So only one restructure is necessary**

Total time: $O(\log n) + O(1) = O(\log n)$

Remove Algorithm

Perform removal as with binary search tree

- **May decrease height of some nodes on path to the root**

Walk upwards to the root

- **If unbalanced height is found, restructure unbalanced region with *rotation* operation**

Remove is also $O(\log n)$

- **But multiple restructure operations may be necessary along the way**

Multi-way Search Trees

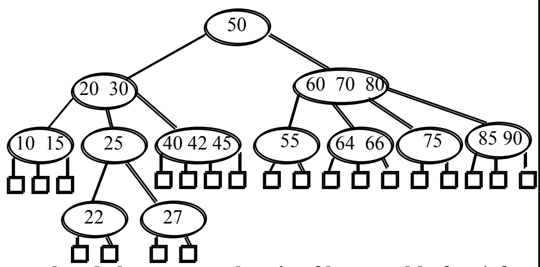
Each node may store multiple key-element pairs

Node with *d* children (*d*-node) stores *d*-1 key-element pairs

Children have keys that fall either before smallest parent key, after largest parent key, or between two parent keys

(for this section, let’s use convention of external nodes storing no element, as in book)

Example Multi-way Search Tree



External node between each pair of keys and before/after

(n-1) + 1 + 1 = n+1 external nodes

Multi-way Tree Searching

Basically same as for binary tree

1. Start at root
2. Find appropriate child path to go down
3. Traverse to child
4. Repeat 1-3 until found or reach external

Multi-way Search Analysis

Number of nodes traversed is up to *h*

Work at each node is function of *d*

- O(log*d*) if structure storing keys provides efficient search, otherwise *O*(*d*)

Total worst case time

- $O(h \log d_{max})$ or $O(h d_{max})$
- If d_{max} is bounded by small constant, just $O(h)$

(2,4) Trees

Efficient multi-way search trees

- *O*(log*n*) height

Maintain two properties:

1. Size property: nodes have at most 4 children
2. Depth property: All external nodes have same depth (i.e. all at the same *level*)

(2,4) Tree Height Analysis

Lower bound on h

$n(h) <= 4^h$: max children per node is 4

external nodes = n+1

$n+1 <= 4^h \Rightarrow h >= \log(n+1) / 2$

Upper bound on h

At least 2 nodes at depth 1, 4 at depth 2, etc.

At least 2^d nodes at depth d

At least 2^h external nodes

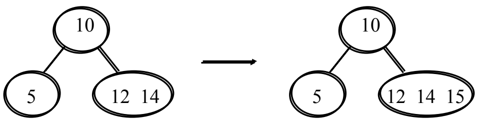
$2^h <= n+1 \Rightarrow h <= \log(n+1)$

$h = \Theta(\log n) \Rightarrow$ search is $O(\log n)$

Inserting into (2,4) Tree

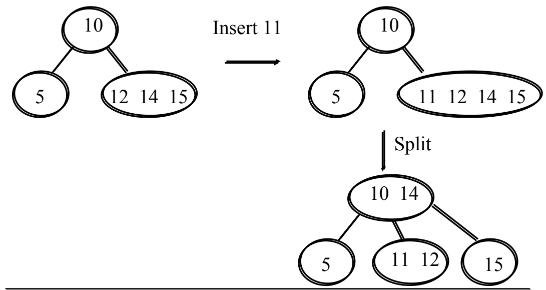
1. Search for position in deepest internal node
2. Insert into position
3. If # elements > 3, do a *split* operation
   - Split node into 2 nodes
   - Push 1 element up to parent
     - —Create new root if no parent
     - —If parent overflows, split parent

Simple Insertion (no overflow)

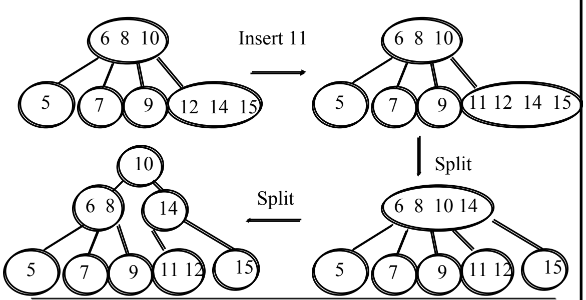


Insert 15

Insertion with Overflow



Insert with Cascading Split



Removing from (2,4) Tree

1. **Search for element**
2. **Remove element**
3. **If element's child is internal**
   - ***Swap* next larger element into hole (so we've removed element above an external)**
4. **If node has no elements**

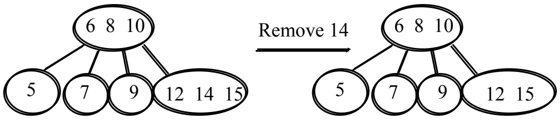
   If an adjacent sibling has > 1 element

   Perform *transfer* (kind of rotation)

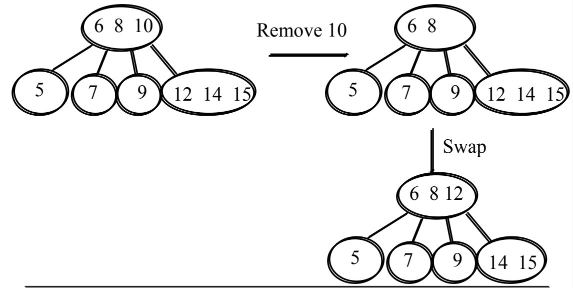
   Else

   Perform *fusion* (can cascade upward)

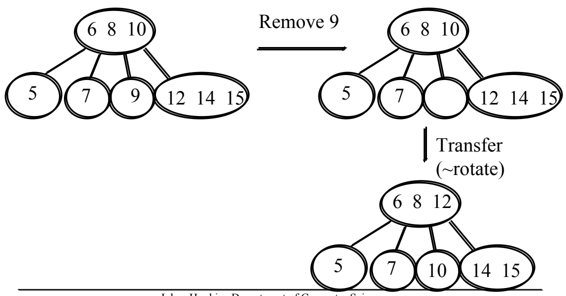
Simple Removal



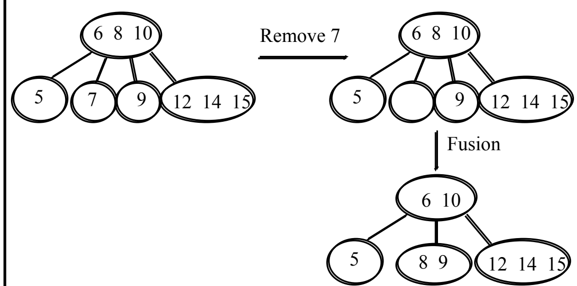
Removal with Swap



Removal with Transfer



Removal with Fusion



Performance

Insertion

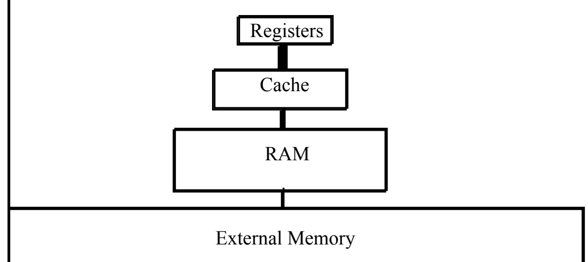
- **Find is *O*(log*n*)**
- **Each split is *O*(1); only *O*(logn) splits necessary**

Remove

- **Each transfer/fusion is O(1); only *O*(logn) necessary**

External Memory Searching

Memory Hierarchy



Types of External Memory

Hard disk

Floppy disk

Compact disc

Tape

Distributed/networked memory

Primary Motivation

External memory access much slower than internal memory access

- **orders of magnitude slower**
- **need to minimize *I/O Complexity***
- **can afford slightly more work on data in memory in exchange for lower I/O complexity**

Application Areas

Searching

Sorting

Data Processing

Data Mining

Data Exploration

Disk Blocks

Data is read one block at a time

- **pack as much into a block as possible**
- **minimize number of block reads necessary**

I/O Efficient Dictionaries

Balanced tree structures

- **Typically $O(\log_2 n)$ transfers for query or update**
- **Want to reduce height by constant factor as much as possible**
- **Can be reduced to $O(\log_B n) = O(\log_2 n / \log_2 B)$**
  - **—B is number of nodes per block**

(a,b) Trees

Generalization of (2,4) trees

Size property: internal node has at least *a* children and at most *b* children

- **$2 <= a <= (b+1)/2$**

Depth property: all external nodes have same depth

Height of (a,b) tree is Ω (logn/logb) and O(logn/loga)

B-Trees

Choose *a* and *b* to be $\Theta(B)$

Height is now $O(\log_B n)$

I/O complexity for search is $O(\log_B n)$