# CSC 427: Data Structures and Algorithm Analysis

# Fall 2006

## TreeSets and TreeMaps

- tree structure, root, leaves
- recursive tree algorithms: counting, searching, traversal
- divide & conquer
- binary search trees, efficiency
- simple TreeSet implementation, iterator
- balanced trees: AVL, red-black

1

# Recall: Tree & Set

## java.util.Set interface: an unordered collection of items, with no duplicates

```
public interface Set<E> extends Collection<E> {
    boolean add(E o);               // adds o to this Set
    boolean remove(Object o);       // removes o from this Set
    boolean contains(Object o);     // returns true if o in this Set
    boolean isEmpty();              // returns true if empty Set
    int size();                     // returns number of elements
    void clear();                   // removes all elements
    Iterator<E> iterator();         // returns iterator
    . . .
}
```

implemented by TreeSet & HashSet

## java.util.Map interface: a collection of key → value mappings

```
public interface Map<K, V> {
    boolean put(K key, V value);   // adds key→value to Map
    V remove(Object key);          // removes key→? entry from Map
    V get(Object key);             // returns true if o in this Set
    boolean containsKey(Object key);     // returns true if key is stored
    boolean containsValue(Object value); // returns true if value is stored
    boolean isEmpty();             // returns true if empty Set
    int size();                    // returns number of elements
    void clear();                  // removes all elements
    Set<K> keySet();               // returns set of all keys
    . . .
}
```

implemented by TreeMap & HashMap

2

## TreeSet & TreeMap

recall that the TreeSet implementation maintains order

- the elements must be Comparable
- an iterator will traverse the elements in increasing order

likewise, the keys of a TreeMap are ordered

- the key elements must be Comparable
- an iterator will traverse the keySet elements in increasing order

the underlying data structure of TreeSet (and a TreeMap's keySet) is a balanced binary search tree

- a binary search tree is a linked structure (as in LinkedLists), but structured hierarchically to enable binary search
- guaranteed O(log N) performance of add, remove, contains

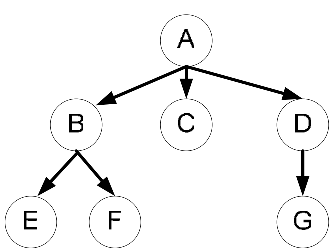
first, a general introduction to trees

3

## Tree

a tree is a nonlinear data structure consisting of nodes (structures containing data) and edges (connections between nodes), such that:

- one node, the *root,* has no *parent* (node connected from above)
- every other node has exactly one parent node
- there is a unique path from the root to each node (i.e., the tree is connected and there are no cycles)



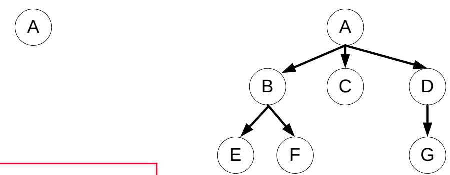

nodes that have no children (nodes connected below them) are known as *leaves*

4

## Recursive definition of a tree

trees are naturally recursive data structures:

- the empty tree (with no nodes) is a tree
- a node with subtrees connected below is a tree




empty tree

tree with 1 node (empty subtrees)

tree with 7 nodes

a tree where each node has at most 2 subtrees (children) is a *binary* tree

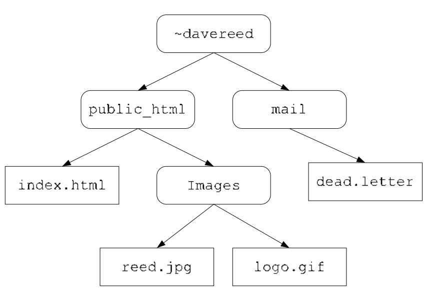
5

## Trees in CS

trees are fundamental data structures in computer science

example: file structure

- an OS will maintain a directory/file hierarchy as a tree structure
- files are stored as leaves; directories are stored as internal (non-leaf) nodes




descending down the hierarchy to a subdirectory
⇕
traversing an edge down to a child node

DISCLAIMER: directories contain links back to their parent directories (e.g., `..`), so not strictly a tree

6

# Recursively listing files

to traverse an arbitrary directory structure, need recursion

to list a file system object (either a directory or file):

1. print the name of the current object
2. if the object is a directory, then
   - recursively list each file system object in the directory

in pseudocode:

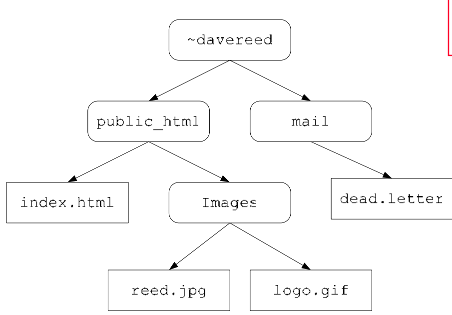
```
public static void ListAll(FileSystemObject current) {
   System.out.println(current.getName());
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       ListAll(obj);
     }
   }
}
```

7

# Recursively listing files

```
public static void ListAll(FileSystemObject current) {
   System.out.println(current.getName());
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       ListAll(obj);
     }
   }
}
```

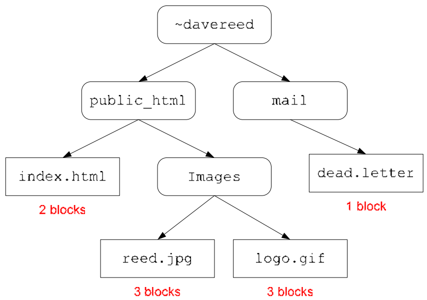
this method performs a *pre-order traversal*: prints the root first, then the subtrees




8

# UNIX du command

in UNIX, the du command list the size of all files and directories




from the `~davereed` directory:

```
unix> du –a
2  ./public_html/index.html
3  ./public_html/Images/reed.jpg
3  ./public_html/Images/logo.gif
7  ./public_html/Images
10 ./public_html
1  ./mail/dead.letter
2  ./mail
13 .
```

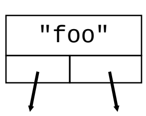
```
public static int du(FileSystemObject current) {
  int size = current.blockSize();
  if (current.isDirectory()) {
    for (FileSystemObject obj : current.getContents()) {
      size += du(obj);
    }
  }
  System.out.println(size + " " + current.getName());
  return size;
}
```

this method performs a *post-order traversal*: prints the subtrees first, then the root

9

# Implementing binary trees

to implement binary trees, we need a node that can store a data value & pointers to two child nodes (RECURSIVE!)




NOTE: exact same structure as with doubly-linked list, only left/right instead of previous/next

```
public class TreeNode<E> {
    private E data;
    private TreeNode<E> left;
    private TreeNode<E> right;

    public TreeNode(E d, TreeNode<E> l, TreeNode<E> r) {
        this.data = d;
        this.left = l;
        this.right = r;
    }

    public E getData() {
        return this.data;
    }

    public TreeNode<E> getLeft() {
        return this.left;
    }

    public TreeNode<E> getRight() {
        return this.right;
    }

    public void setData(E newData) {
        this.data = newData;
    }

    public void setLeft(TreeNode<E> newLeft) {
        this.left = newLeft;
    }

    public void setRight(TreeNode<E> newRight) {
        this.right = newRight;
    }
}
```

10

## Example: counting nodes in a tree

due to their recursive nature, trees are naturally handled recursively

to count the number of nodes in a binary tree:

BASE CASE: if the tree is empty, number of nodes is 0

RECURSIVE: otherwise, number of nodes is
(# nodes in left subtree) + (# nodes in right subtree) + 1 for the root

```
public static <E> int numNodes(TreeNode<E> root) {
    if (root == null) {
        return 0;
    }
    else {
        return numNodes(root.getLeft()) + numNodes(root.getRight()) + 1;
    }
}
```



## Searching a tree

to search for a particular item in a binary tree:

BASE CASE: if the tree is empty, the item is not found
BASE CASE: otherwise, if the item is at the root, then found

RECURSIVE: otherwise, search the left and then right subtrees

```
public static <E> boolean contains(TreeNode<E> root, E value) {
    return (root != null && (root.getData().equals(value) ||
                             contains(root.getLeft(), value) ||
                             contains(root.getRight(), value)));
}
```

## Traversing a tree: preorder

there are numerous patterns that can be used to traverse the entire tree

pre-order traversal:

BASE CASE: if the tree is empty, then nothing to print

RECURSIVE: print the root, then recursively traverse the left and right subtrees

```
public static <E> void preOrder(TreeNode<E> root) {
    if (root != null) {
          System.out.println(root.getData());
          preOrder(root.getLeft());
          preOrder(root.getRight());
    }
}
```

13

## Traversing a tree: inorder & postorder

in-order traversal:

BASE CASE: if the tree is empty, then nothing to print

RECURSIVE: recursively traverse left subtree, then display root, then right subtree

```
public static <E> void inOrder(TreeNode<E> root) {
    if (root != null) {
          inOrder(root.getLeft());
          System.out.println(root.getData());
          inOrder(root.getRight());
    }
}
```

post-order traversal:

BASE CASE: if the tree is empty, then nothing to print

RECURSIVE: recursively traverse left subtree, then right subtree, then display root

```
public static <E> void postOrder(TreeNode<E> root) {
    if (root != null) {
          postOrder(root.getLeft());
          postOrder(root.getRight());
          System.out.println(root.getData());
    }
}
```

14

## Exercises

```
/** @return the number of times value occurs in the tree with specified root */
public static <E> int numOccur(TreeNode<E> root, E value) {



}
```

```
/** @return the sum of all the values stored in the tree with specified root */
public static <E> int sum(TreeNode<E> root) {



}
```

```
/** @return the # of nodes in the longest path from root to leaf in the tree */
public static <E> int height(TreeNode<E> root) {



}
```

15

## Divide & Conquer algorithms

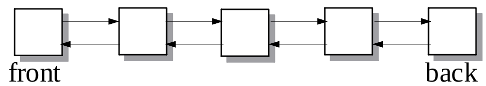
since trees are recursive structures, most tree traversal and manipulation operations can be classified as *divide & conquer algorithms*

- can divide a tree into root + left subtree + right subtree
- most tree operations handle the root as a special case, then recursively process the subtrees
- e.g., to display all the values in a (nonempty) binary tree, divide into
  1. *displaying the root*
  2. *(recursively) displaying all the values in the left subtree*
  3. *(recursively) displaying all the values in the right subtree*
- e.g., to count number of nodes in a (nonempty) binary tree, divide into
  1. *(recursively) counting the nodes in the left subtree*
  2. *(recursively) counting the nodes in the right subtree*
  3. *adding the two counts + 1 for the root*

16

## Searching linked lists

recall: a (linear) linked list only provides sequential access → O(N) searches




it is possible to obtain O(log N) searches using a tree structure

in order to perform binary search efficiently, must be able to

- access the middle element of the list in O(1)
- divide the list into halves in O(1) and recurse

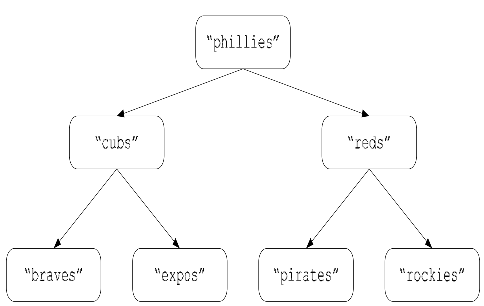
HOW CAN WE GET THIS FUNCTIONALITY FROM A TREE?

17

## Binary search trees

a *binary search tree* is a binary tree in which, for every node:

- the item stored at the node is ≥ all items stored in the left subtree
- the item stored at the node is < all items stored in the right subtree




in a (balanced) binary search tree:

- middle element = root
- 1st half of list = left subtree
- 2nd half of list = right subtree

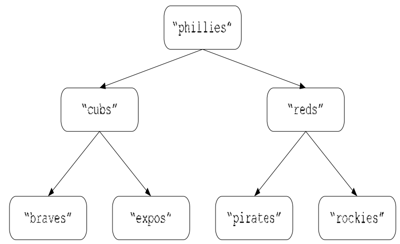
furthermore, these properties hold for each subtree

18

## Binary search in BSTs

to search a binary search tree:

1. if the tree is empty, NOT FOUND
2. if desired item is at root, FOUND
3. if desired item < item at root, then recursively search the left subtree
4. if desired item > item at root, then recursively search the right subtree




can define as a static method in a library class

```
public class BST {

    public static <E extends Comparable<? super E>>
                  TreeNode<E> findNode(TreeNode<E> current, E value) {
        if (current == null || value.compareTo(current.getData()) == 0) {
            return current;
        }
        else if (value.compareTo(current.getData()) < 0) {
            return BST.findNode(current.getLeft(), value);
        }
        else {
            return BST.findNode(current.getRight(), value);
        }
    }
    . . .
}
```

19

## Search efficiency

how efficient is search on a BST?

- in the best case?

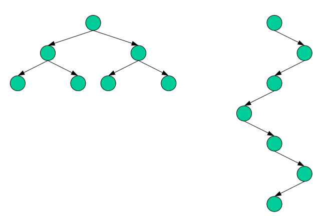
  O(1) if desired item is at the root

- in the worst case?

  O(height of the tree) if item is leaf on the longest path from the root

in order to optimize worst-case behavior, want a (relatively) balanced tree

- otherwise, don't get binary reduction
- e.g., consider two trees, each with 7 nodes



20

# How deep is a balanced tree?

THEOREM: A binary tree with height H can store up to $2^H$-1 nodes.

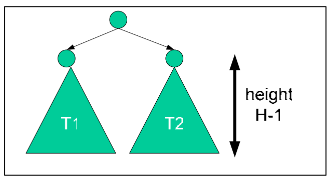
Proof (by induction):

BASE CASES: when H = 0, $2^0$ - 1 = 0 nodes ✓

when H = 1, $2^1$ - 1 = 1 node ✓

HYPOTHESIS: assume a tree with height H-1 can store up to $2^{H-1}$ -1 nodes

INDUCTIVE STEP: a tree with height H has a root and subtrees with height up to H-1




by our hypothesis, T1 and T2 can each store $2^{H-1}$ -1 nodes, so tree with height H can store up to

$1 + (2^{H-1} -1) + (2^{H-1} -1) =$

$2^{H-1} + 2^{H-1} -1 =$

$2^H$ -1 nodes ✓

equivalently: N nodes can be stored in a binary tree of height $\lceil \log_2(N+1) \rceil$

21

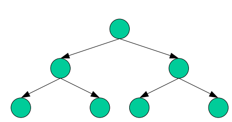
# Search efficiency (cont.)

so, in a balanced binary search tree, searching is O(log N)

N nodes → height of ⌈log2(N+1)⌉ → in worst case, have to traverse ⌈log2(N+1)⌉ nodes

what about the average-case efficiency of searching a binary search tree?

- assume that a search for each item in the tree is equally likely
- take the cost of searching for each item and average those costs



costs of search

1

2 + 2

3 + 3 + 3 + 3

→ 17/7 → 2.42

define the *weight* of a tree to be the sum of all node depths (root = 1, …)

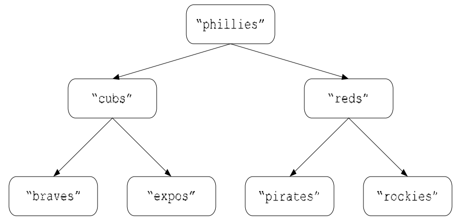
*average cost of searching a tree = weight of tree / number of nodes in tree*

22

# Inserting an item

<u>inserting into a BST</u>

1. traverse edges as in a search
2. when you reach a leaf, add the new node below it




note: the `add` method returns the root of the updated tree

- must maintain links as recurse

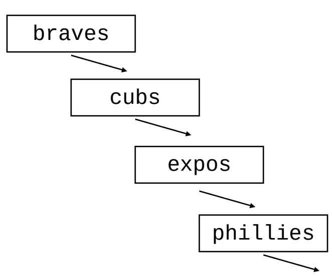
```
public static <E extends Comparable<? super E>>
              TreeNode<E> add(TreeNode<E> current, E value) {
    if (current == null) {
        return new TreeNode(value, null, null);
    }

    if (value.compareTo(current.getData()) <= 0) {
        current.setLeft(BST.add(current.getLeft(), value));
    }
    else {
        current.setRight(BST.add(current.getRight(), value));
    }
    return current;
}
```

23

# Maintaining balance

PROBLEM: random insertions do not guarantee balance

- e.g., suppose you started with an empty tree & added words in alphabetical order
  braves, cubs, expos, phillies, pirates, red, rockies, …




with repeated insertions, can degenerate so that height is O(N)

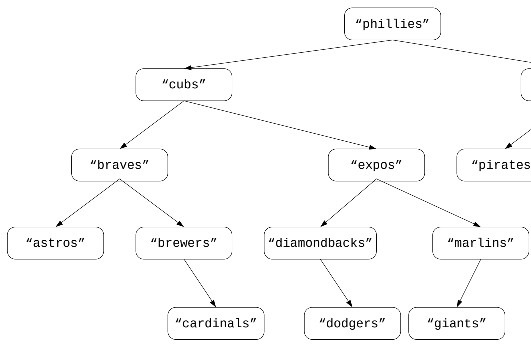
- specialized algorithms exist to maintain balance & ensure O(log N) height <u>(LATER)</u>
- or take your chances: on average, N random insertions yield O(log N) height

24

## Removing an item

we could define an algorithm that finds the desired node and removes it

- tricky, since removing from the middle of a tree means rerouting pointers
- have to maintain BST ordering property




simpler solution

1. find node (as in search)
2. if a leaf, simply remove it
3. if no left subtree, reroute parent pointer to right subtree
4. otherwise, replace current value with largest value in left subtree

25

## Recursive implementation

if item to be removed is at the root

- if no left subtree, return right subtree
- otherwise, remove largest value from left subtree, copy into root, & return

otherwise, remove the node from the appropriate subtree

```
public static <E extends Comparable<? super E>>
              TreeNode<E> remove(TreeNode<E> current, E value) {
    if (current == null) {
        return null;
    }

    if (value.equals(current.getData())) {
        if (current.getLeft() == null) {
            current = current.getRight();
        }
        else {
            current.setData(BST.lastNode(current.getLeft()).getData());
            current.setLeft(BST.remove(current.getLeft(), current.getData()));
        }
    }
    else if (value.compareTo(current.getData()) < 0) {
        current.setLeft(BST.remove(current.getLeft(), value));
    }
    else {
        current.setRight(BST.remove(current.getRight(), value));
    }
    return current;
}
```

26

# firstNode & lastNode

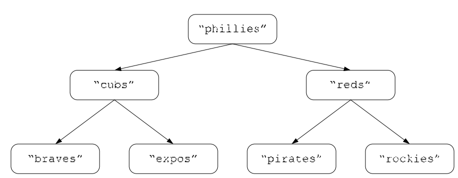
`remove` required finding the largest value in a subtree

- define `firstNode` to find the leftmost node (containing smallest value in the tree)
- define `lastNode` to find the rightmost node (containing largest value in the tree)

```
public static <E extends Comparable<? super E>>
              TreeNode<E> firstNode(TreeNode<E> current) {
    if (current == null) {
        return null;
    }

    while (current.getLeft() != null) {
        current = current.getLeft();
    }
    return current;
}

public static <E extends Comparable<? super E>>
              TreeNode<E> lastNode(TreeNode<E> current) {
    if (current == null) {
        return null;
    }

    while (current.getRight() != null) {
        current = current.getRight();
    }
    return current;
}
```

27

# toString method




to help in testing/debugging, can define a `toString` method

`treeToRight.toString()` →

`"[braves,cubs,expos,phillies,pirates,reds,rockies]"`

```
    public static <E extends Comparable<? super E>>
                  String toString(TreeNode<E> current)
{
        if (current == null) {
            return "[]";
        }

        String recStr = BST.stringify(current);
        return "[" +
recStr.substring(0,recStr.length()-1) + "]";
    }

    private static <E extends Comparable<? super E>>
                   String stringify(TreeNode<E>
current) {
        if (current ==  null) {
            return "";
        }
```

28

## SimpleTreeSet implementation

```
public class SimpleTreeSet<E extends Comparable<? super E>> implements Iterable<E>{
    private TreeNode<E> root;
    private int nodeCount = 0;

    public SimpleTreeSet() {
        this.root = null;
    }

    public int size() {
        return this.nodeCount;
    }

    public void clear() {
        this.root = null;
        this.nodeCount = 0;
    }

    public boolean contains(E value) {
        return (BST.findNode(this.root, value) != null);
    }

    public boolean add(E value) {
        if (this.contains(value)) {
            return false;
        }

        this.root = BST.add(this.root, value);
        this.nodeCount++;
        return true;
    }
    . . .
```

we can now implement a simplified TreeSet class with an underlying binary search tree (and utilizing the BST static methods)

29

## SimpleTreeSet implementation (cont.)

```
    public boolean remove(E value) {
        if (!this.contains(value)) {
            return false;
        }

        root = BST.remove(root, value);
        this.nodeCount--;
        return true;
    }

    public E first() {
        if (this.root == null) {
            throw new NoSuchElementException();
        }

        return BST.firstNode(this.root).getData();
    }

    public E last() {
        if (this.root == null) {
            throw new NoSuchElementException();
        }

        return BST.lastNode(this.root).getData();
    }

    public String toString() {
        return BST.toString(this.root);
    }

    . . .
```

what about an iterator?

where should it start?

how do you get the next item?

30

## SimpleTreeSet implementation (cont.)

```java
    private class TreeIterator implements Iterator<E> {
        private TreeNode<E> nextNode;

        public TreeIterator() {
            this.nextNode = BST.firstNode(SimpleTreeSet.this.root);
        }

        public boolean hasNext() {
            return this.nextNode != null;
        }

        public E next() {
           if (!this.hasNext()) {
               throw new NoSuchElementException();
           }

           E returnValue = this.nextNode.getData();
           if (this.nextNode.getRight() != null) {
               this.nextNode = BST.firstNode(this.nextNode.getRight());
           }
           else {
               TreeNode<E> parent = null;
               TreeNode<E> stepper = SimpleTreeSet.this.root;
               while (stepper != this.nextNode) {
                   if (this.nextNode.getData().compareTo(stepper.getData()) < 0) {
                       parent = stepper;
                       stepper = stepper.getLeft();
                   }
                   else {
                       stepper = stepper.getRight();
                   }
               }
               this.nextNode = parent;
           }
           return returnValue;
        }

        public void remove() {
            // TO BE IMPLEMENTED
        }
    }

    public Iterator<E> iterator() {
        return new TreeIterator();
    }
}
```

similar to LinkedList, keep a reference to next node

- initialize to leftmost node

to find the next node

- if have right child, get leftmost node in right subtree
- otherwise, find the nearest parent such that current node is not in that parent's right subtree

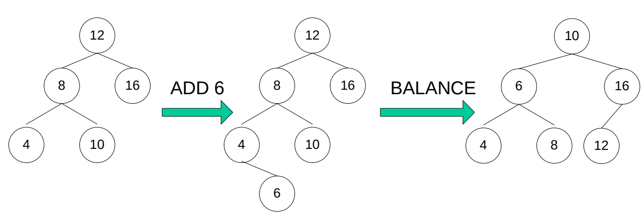
31

## Balancing trees

on average, N random insertions into a BST yields O(log N) height

- however, degenerative cases exist (e.g., if data is close to ordered)

we can ensure logarithmic depth by maintaining balance




maintaining full balance can be costly

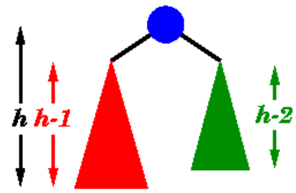
- however, full balance is not needed to ensure O(log N) operations
- specialized structures/algorithms exist: AVL trees, 2-3 trees, red-black trees, …

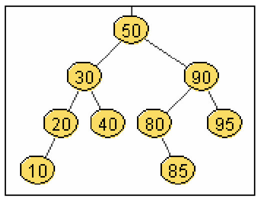
32

## AVL trees

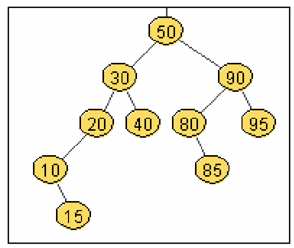
an AVL tree is a binary search tree where

- for every node, the heights of the left and right subtrees differ by at most 1
- first self-balancing binary search tree variant
- named after Adelson-Velskii & Landis (1962)




AVL tree



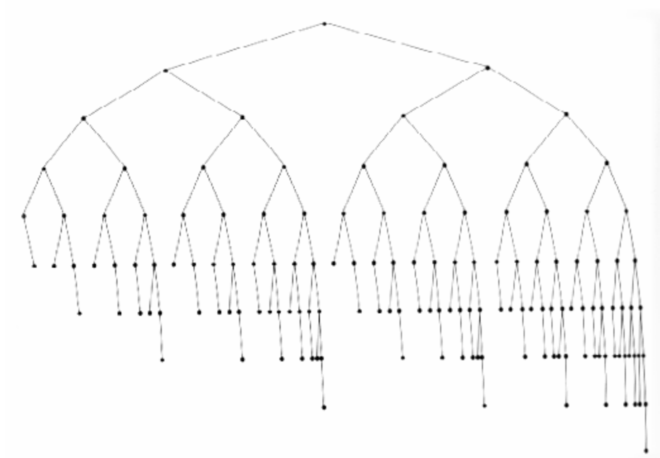

not an AVL tree – WHY?

33

## AVL trees and balance

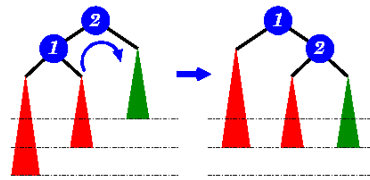
the AVL property is weaker than full balance, but sufficient to ensure logarithmic height

- height of AVL tree with N nodes < 2 log(N+2) → searching is O(log N)



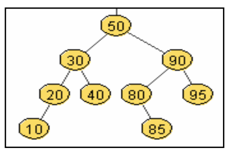
34

## Inserting/removing from AVL tree

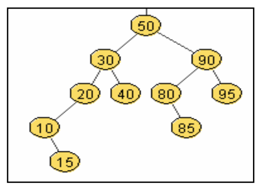



when you insert or remove from an AVL tree, may need to rebalance

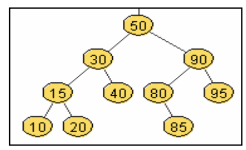
- add/remove value as with binary search trees
- may need to rotate subtrees to rebalance
- see www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html




consider AVL tree




inserting ruins balance




move up levels & rotate

worst case, inserting/removing requires traversing the path back to the root and rotating at each level

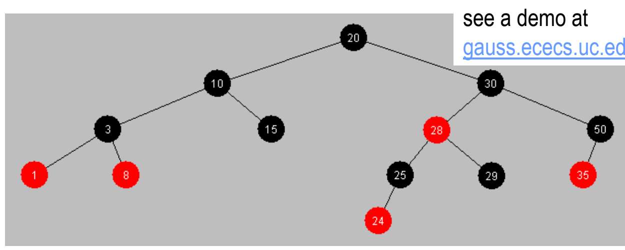
- each rotation is a constant amount of work → inserting/removing is O(log N)

35

## Red-black trees & TreeSets & TreeMaps

`java.util.TreeSet` uses *red-black trees* to maintain balance

- a red-black tree is a binary search tree in which each node is assigned a color (either red or black) such that
  1. *the root is black*
  2. *a red node never has a red child*
  3. *every path from root to leaf has the same number of black nodes*
- add & remove preserve these properties (complex, but still O(log N))
- red-black properties ensure that tree height < 2 log(N+1) → O(log N) search




see a demo at gauss.ececs.uc.edu/RedBlack/redblack.html

similarly, `TreeMap` uses a red-black tree to store the key-value pairs

36