CSC 321: Data Structures

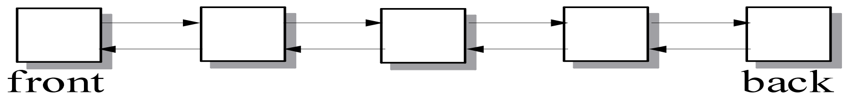
Fall 2013

Binary Search Trees

- BST property
- override binary tree methods: add, contains
- search efficiency
- balanced trees: AVL, red-black
- heaps, priority queues, heap sort

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

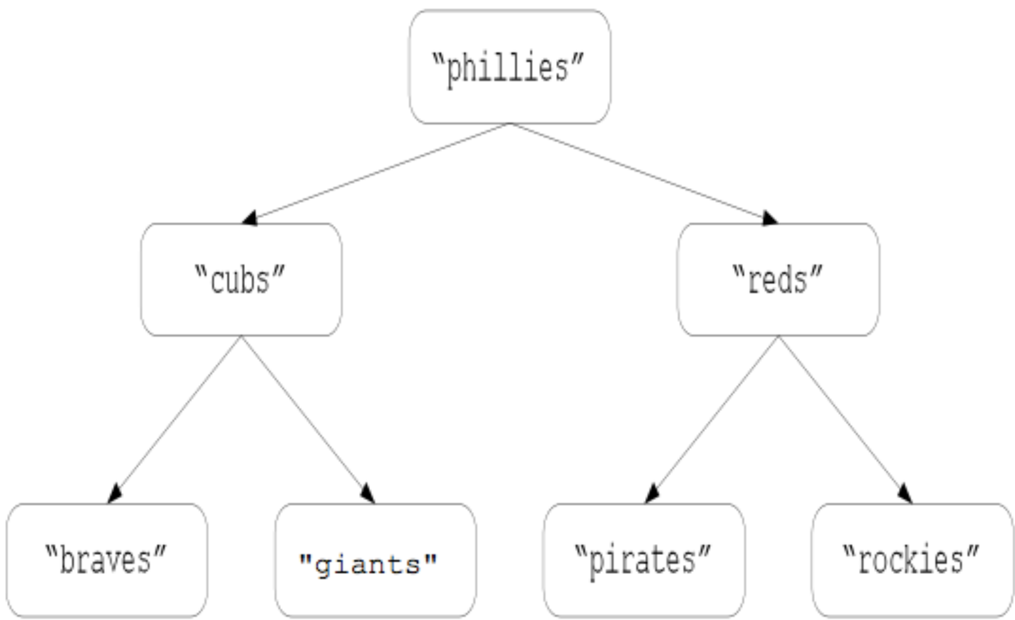
- access the middle element of the list in O(1)
- divide the list into halves in O(1) and recurse

HOW CAN WE GET THIS FUNCTIONALITY FROM A TREE?

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



in a (balanced) binary search tree:

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

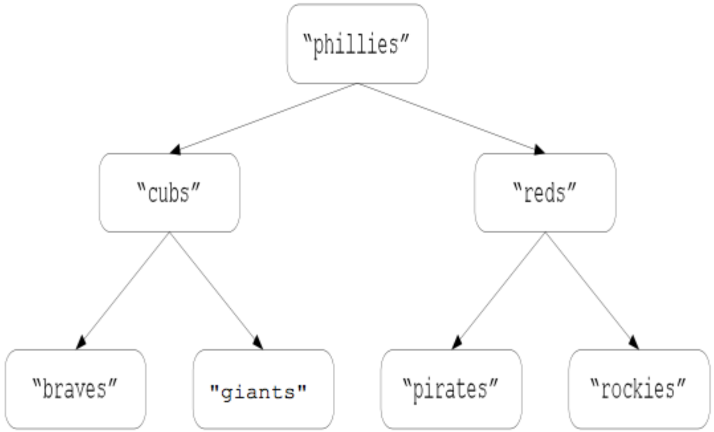
furthermore, these properties hold for each subtree

BinarySearchTree class

can use inheritance to derive BinarySearchTree from BinaryTree

```
public class BinarySearchTree<E extends Comparable<? super E>>
extends BinaryTree<E> {

    public BinarySearchTree() {
        super();
    }

    public void add(E value) {
        // OVERRIDE TO MAINTAIN BINARY SEARCH TREE PROPERTY
    }

    public void contains(E value) {
        // OVERRIDE TO TAKE ADVANTAGE OF BINARY SEARCH TREE PROPERTY
    }

    public void remove(E value) {
        // DOES THIS NEED TO BE OVERRIDDEN?
    }
}
```

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

```
public boolean contains(E value) {
    return this.contains(this.root, value);
}

private boolean contains(TreeNode<E> current, E value) {
    if (current == null) {
        return false;
    }
    else if (value.equals(current.getData())) {
        return true;
    }
    else if (value.compareTo(current.getData()) < 0) {
        return this.contains(current.getLeft(), value);
    }
    else {
        return this.contains(current.getRight(), value);
    }
}
```

Search efficiency

how efficient is search on a BST?

- in the best case?

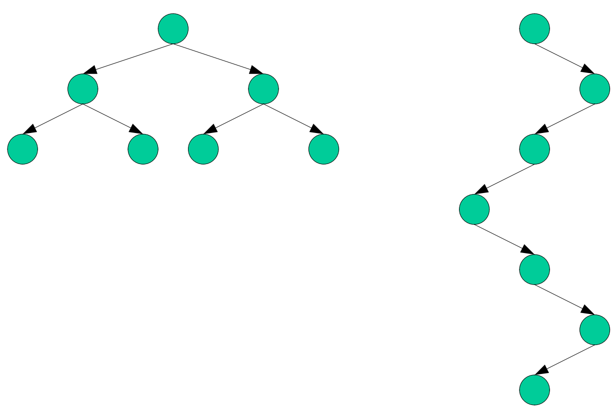
  O(1) if desired item is at the root

- 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



Search efficiency (cont.)

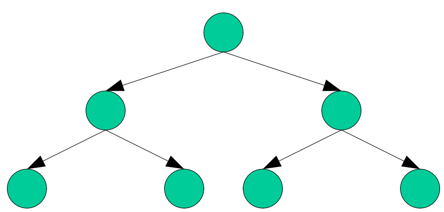
we showed that N nodes can be stored in a binary tree of height $\lceil \log_2(N+1) \rceil$

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

N nodes → height of $\lceil \log2(N+1) \rceil$ → in worst case, have to traverse $\lceil \log2(N+1) \rceil$ 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

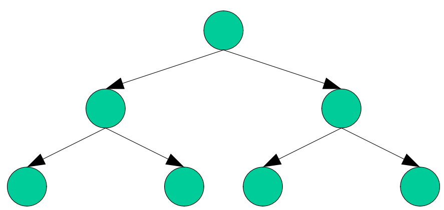
3 + 3 + 3 + 3

→ 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 BST = weight of tree / number of nodes in tree

Search efficiency (cont.)



costs of search

1

2 + 2

3 + 3 + 3 + 3

➔ 17/7 ➔ 2.42

~log N

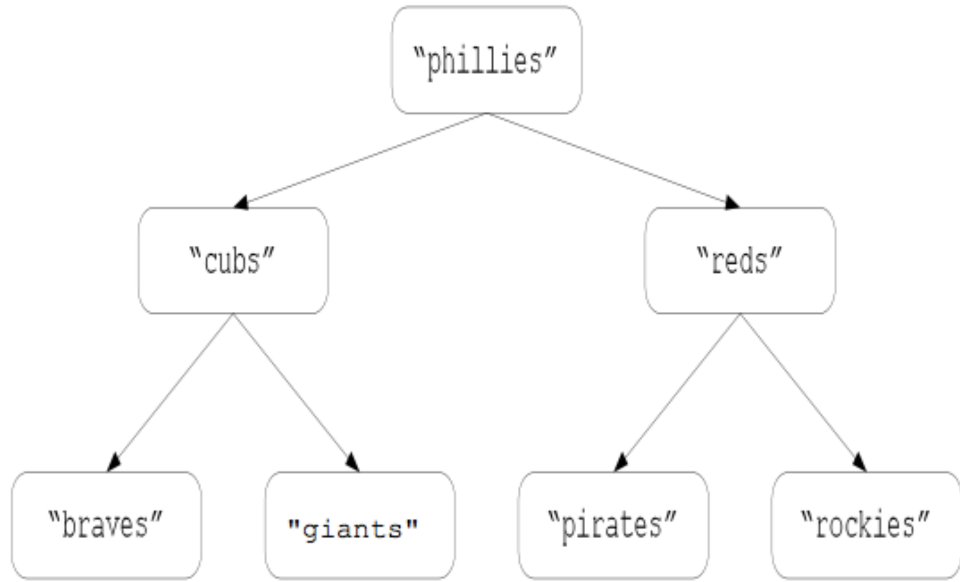
costs of search

1
+ 2
+ 3
+ 4
+ 5
+ 6
+ 7

➔ 28/7 ➔ 4.00

~N/2

Inserting an item

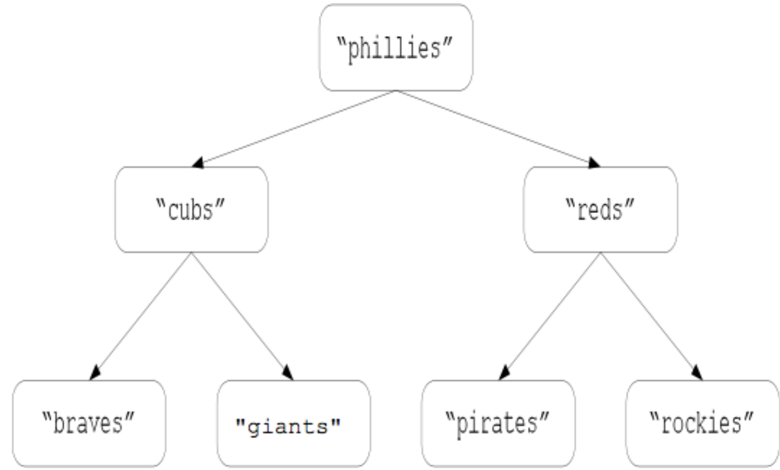


inserting into a BST

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

```
public void add(E value) {
    this.root = this.add(this.root, value);
}

private TreeNode<E> add(TreeNode<E> current, E value) {
    if (current == null) {
        return new TreeNode<E>(value, null, null);
    }

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

Removing an item



recall BinaryTree remove

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 a leaf value from the left subtree (and remove the leaf node)

CLAIM: as long as you select the rightmost (i.e., maximum) value in the left subtree, this remove algorithm maintains the BST property

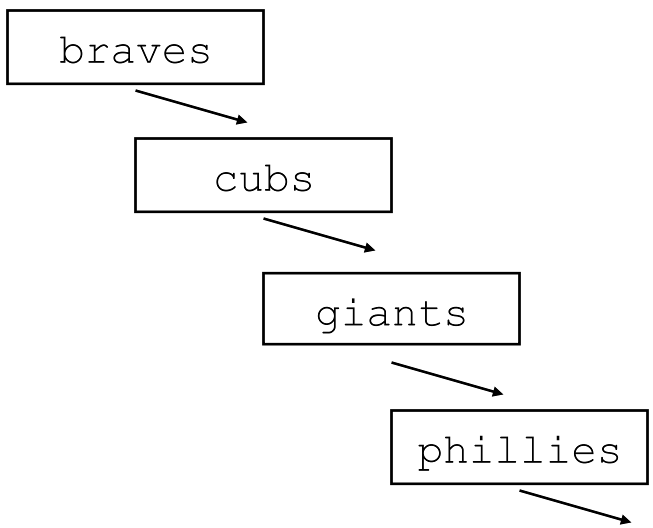
WHY?

so, no need to override remove

Maintaining balance

PROBLEM: random insertions (and removals) do not guarantee balance

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



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

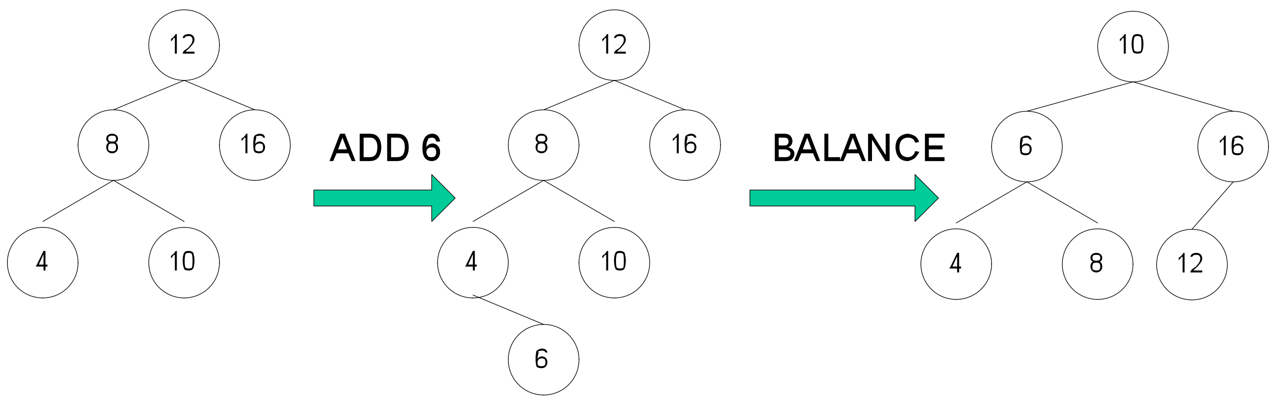
- specialized algorithms exist to maintain balance & ensure O(log N) height
- or take your chances

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



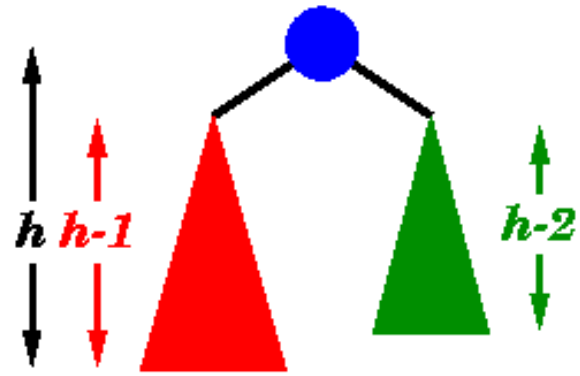
maintaining full balance can be costly

- however, full balance is not needed to ensure O(log N) operations

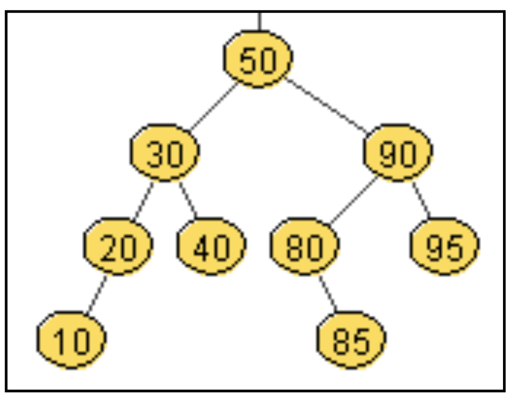
AVL trees

an AVL tree is a binary search tree where

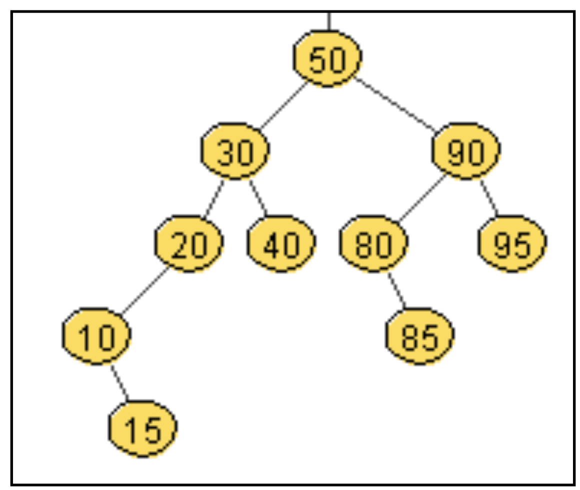
- for every node, the heights of the left and right subtrees differ by at most 1



- first self-balancing binary search tree variant
- named after Adelson-Velskii & Landis (1962)



AVL tree

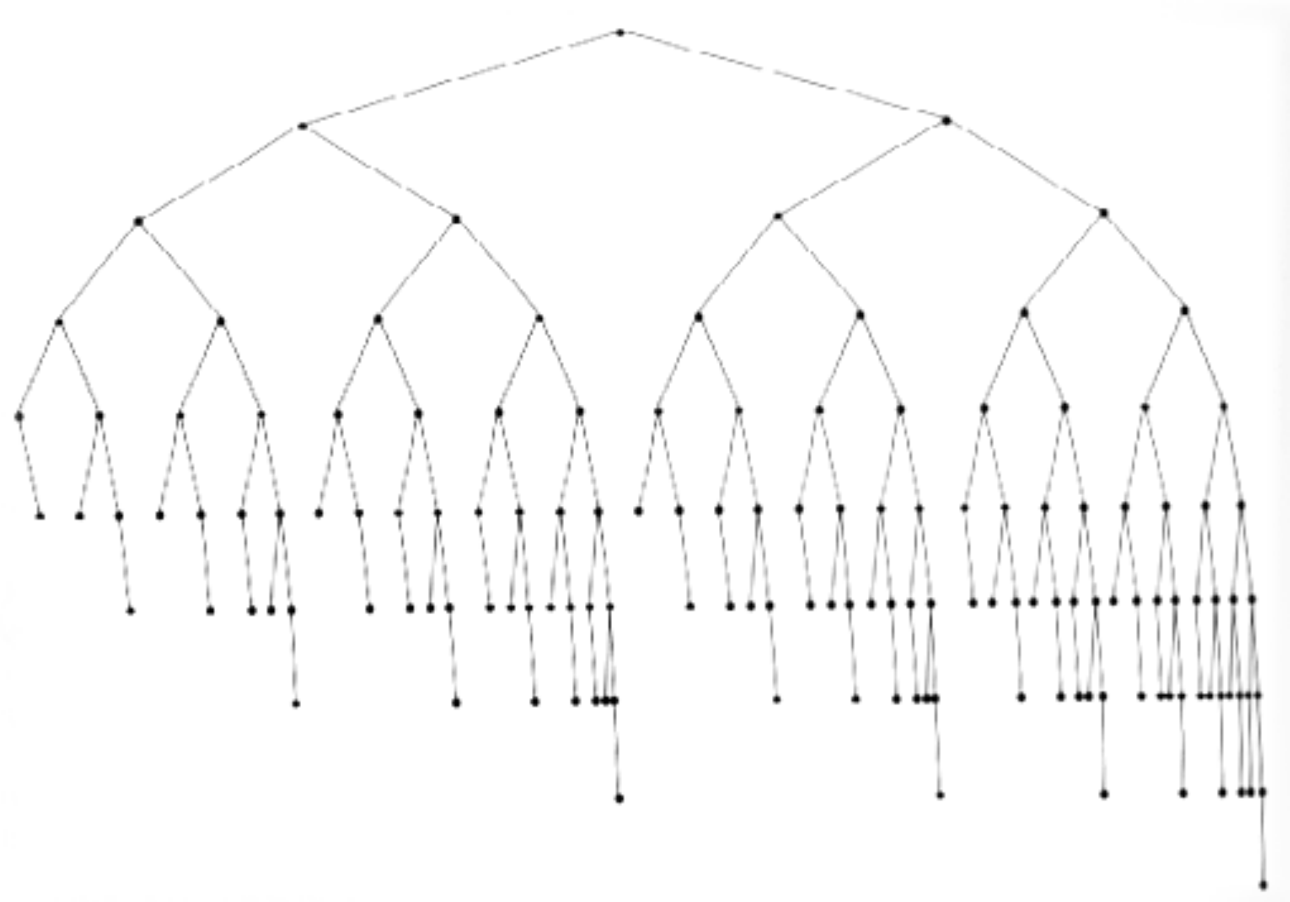


not an AVL tree – WHY?

AVL trees and balance

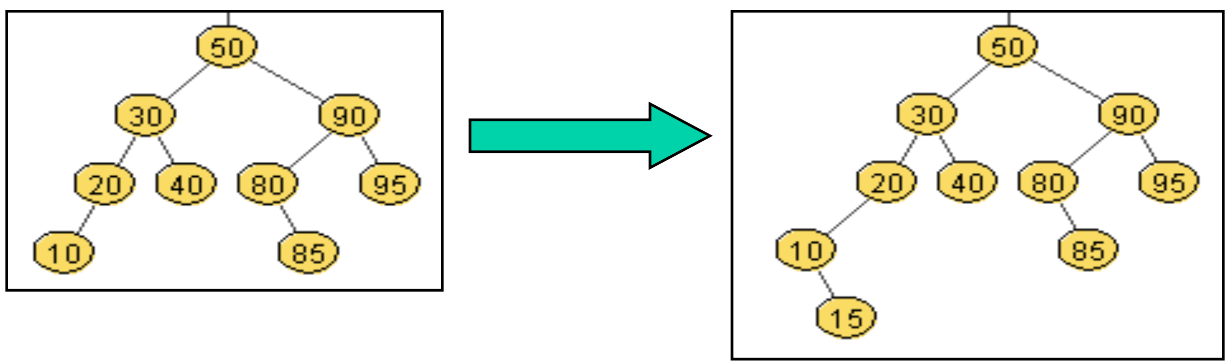
the AVL property is weaker than full balance, but sufficient to ensure logarithmic height

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

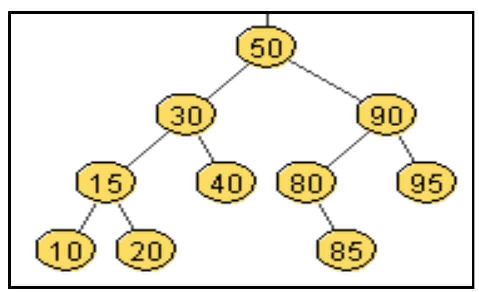


Inserting/removing from AVL tree

when you insert or remove from an AVL tree, imbalances can occur



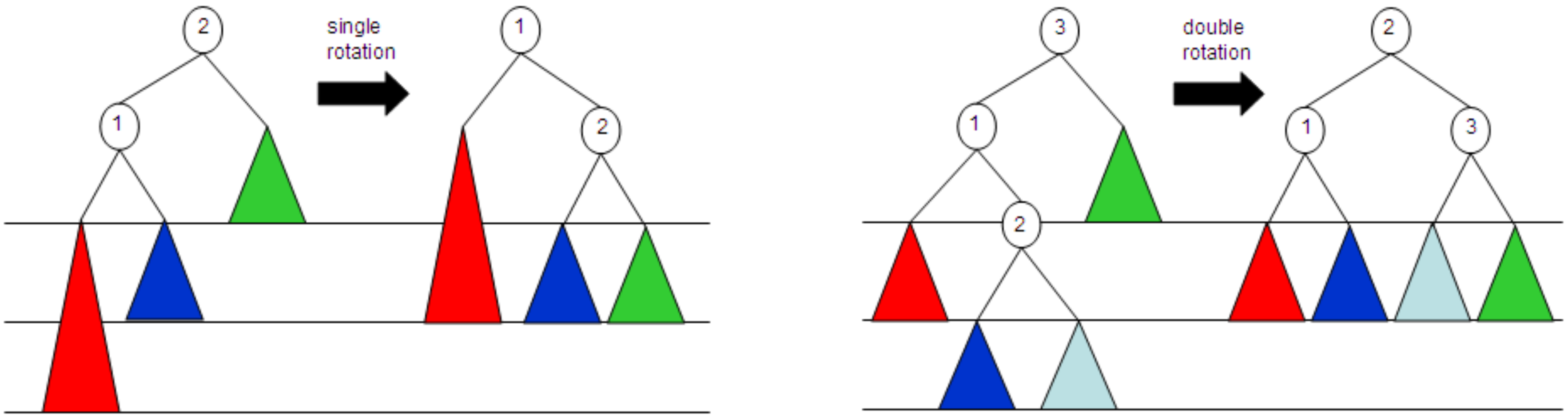
- if an imbalance occurs, must rotate subtrees to retain the AVL property



- see www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

AVL tree rotations

there are two possible types of rotations, depending upon the imbalance caused by the insertion/removal



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

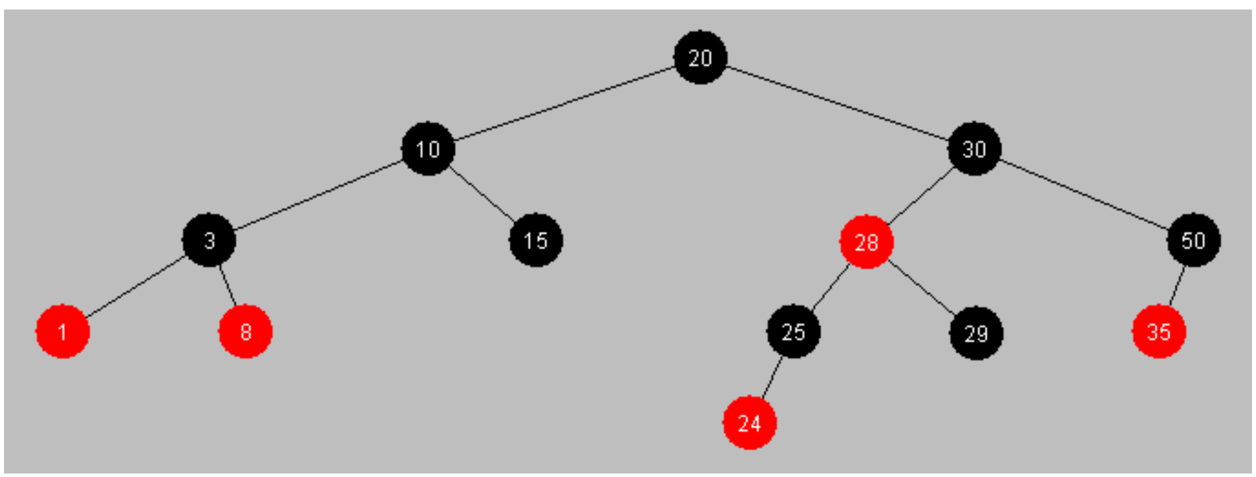
- each rotation is a constant amount of work ➔ inserting/removing is O(log N)

Red-black trees

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



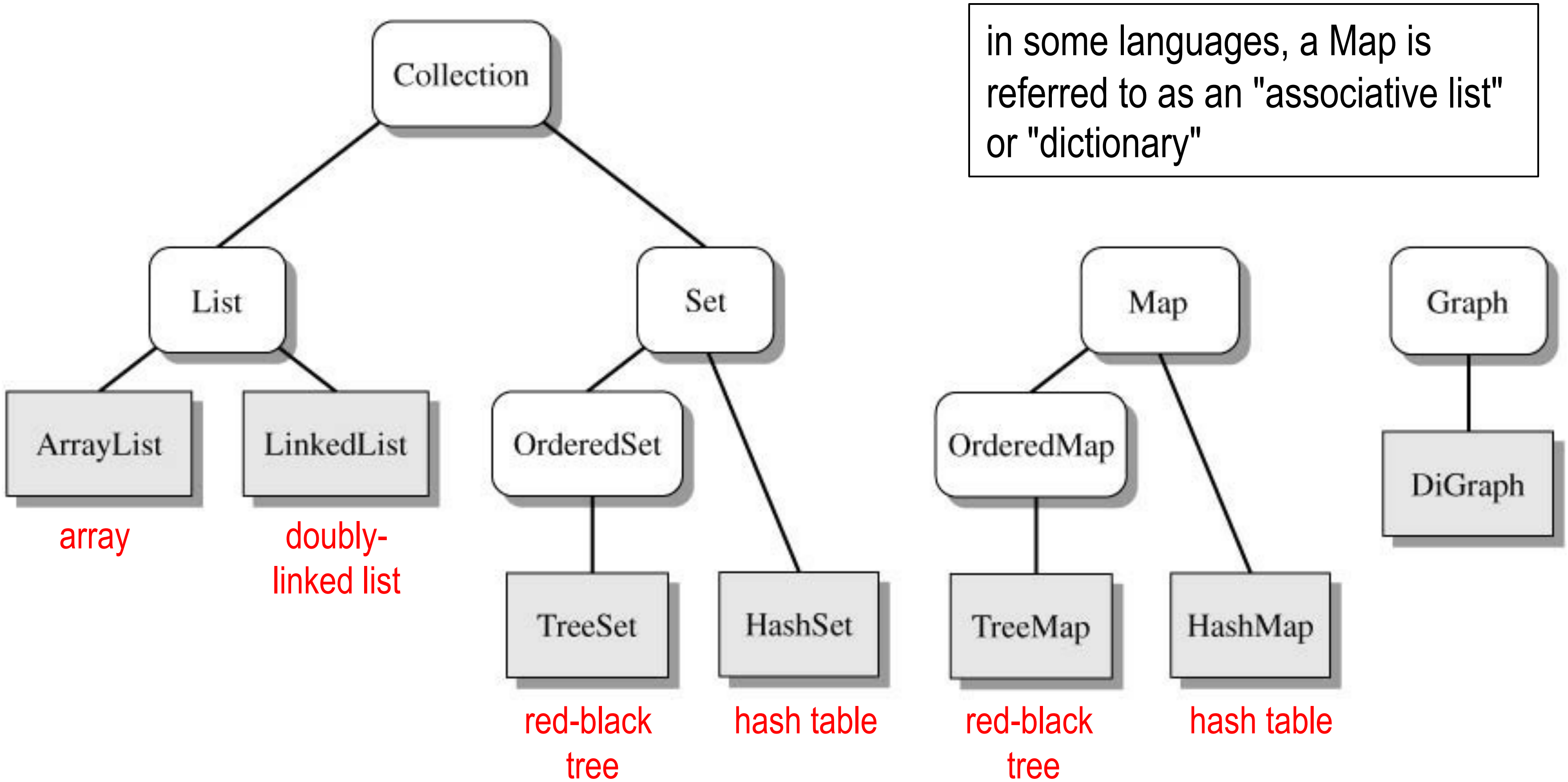
see a demo at gauss.ececs.uc.edu/RedBlack/redblack.html

Java Collection classes

recall the Java Collection Framework

- defined using interfaces abstract classes, and inheritance

in some languages, a Map is referred to as an "associative list" or "dictionary"



Sets

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 and TreeMap classes

TreeSet implementation

- ✓ implemented using a red-black tree; items stored in the nodes (must be Comparable)
- ✓ provides O(log N) add, remove, and contains (guaranteed)
- ✓ iteration over a TreeSet accesses the items in order (based on compareTo)

HashSet implementation

- ✓ HashSet utlizes a hash table data structure LATER
- ✓ HashSet provides O(1) add, remove, and contains (on average, but can degrade)

Dictionary revisited

note: our Dictionary class could have been implemented using a Set

- Strings are Comparable, so could use either implementation
- TreeSet has the advantage that iterating over the Set elements gives them in order (here, alphabetical order)

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private Set<String> words;

    public Dictionary() {
        this.words = new TreeSet<String>();
    }

    public Dictionary(String filename) {
        this();
        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.add(nextWord);
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

Maps

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 and HashMap classes

TreeMap implementation

- ✓ utilizes a red-black tree to store key/value pairs; ordered by the (Comparable) keys
- ✓ provides O(log N) put, get, and containsKey (guaranteed)
- ✓ keySet() returns a TreeSet, so iteration over the keySet accesses the key in order

HashMap implementation

- ✓ HashSet utlizes a HashSet to store key/value pairs LATER
- ✓ HashSet provides O(1) put, get, and containsKey (on average, but can degrade)

Word frequencies

a variant of Dictionary is WordFreq

- stores words & their frequencies (number of times they occur)
- can represent the word→counter pairs in a Map
- again, could utilize either Map implementation
- since TreeMap is used, showAll displays words + counts in alphabetical order

```
import java.util.Map;
import java.util.TreeMap;
import java.util.Scanner;
import java.io.File;

public class WordFreq {
    private Map<String, Integer> words;

    public WordFreq() {
        words = new TreeMap<String, Integer>();
    }

    public WordFreq(String filename) {
        this();
        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.add(nextWord);
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        String cleanWord = newWord.toLowerCase();
        if (words.containsKey(cleanWord)) {
            words.put(cleanWord, words.get(cleanWord)+1);
        }
        else {
            words.put(cleanWord, 1);
        }
    }

    public void showAll() {
        for (String str : words.keySet()) {
            System.out.println(str + ": " + words.get(str));
        }
    }
}
```

Other tree structures

a *heap* is a common tree structure that:

- can efficiently implement a priority queue (a list of items that are accessed based on some ranking or priority as opposed to FIFO/LIFO)
- can also be used to implement another O(N log N) sort

motivation: many real-world applications involve optimal scheduling

- choosing the next in line at the deli
- prioritizing a list of chores
- balancing transmission of multiple signals over limited bandwidth
- selecting a job from a printer queue
- multiprogramming/multitasking

all these applications require

- storing a collection of prioritizable items, and
- selecting and/or removing the highest priority item

Priority queue

priority queue is the ADT that encapsulates these 3 operations:

- ✓ *add item (with a given priority)*
- ✓ *find highest priority item*
- ✓ *remove highest priority item*

e.g., assume printer jobs are given a priority 1-5, with 1 being the most urgent

a priority queue can be implemented in a variety of ways

job1	job 2	job 3	job 4	job 5
3	4	1	4	2

- unsorted list
  efficiency of add? efficiency of find? efficiency of remove?

job4	job 2	job 1	job 5	job 3
4	4	3	2	1

- sorted list (sorted by priority)
  efficiency of add? efficiency of find? efficiency of remove?

- others?

java.util.PriorityQueue

Java provides a `PriorityQueue` class

```
public class PriorityQueue<E extends Comparable<? super E>> {
    /** Constructs an empty priority queue
     */
    public PriorityQueue<E>() { … }

    /** Adds an item to the priority queue (ordered based on compareTo)
     *    @param newItem the item to be added
     *    @return true if the items was added successfully
     */
    public boolean add(E newItem) { … }

    /** Accesses the smallest item from the priority queue (based on compareTo)
     *    @return the smallest item
     */
    public E peek() { … }

    /** Accesses and removes the smallest item (based on compareTo)
     *    @return the smallest item
     */
    public E remove() { … }

    public int size() { … }
    public void clear() { … }
    . . .
}
```

the underlying data structure is a special kind of binary tree called a *heap*

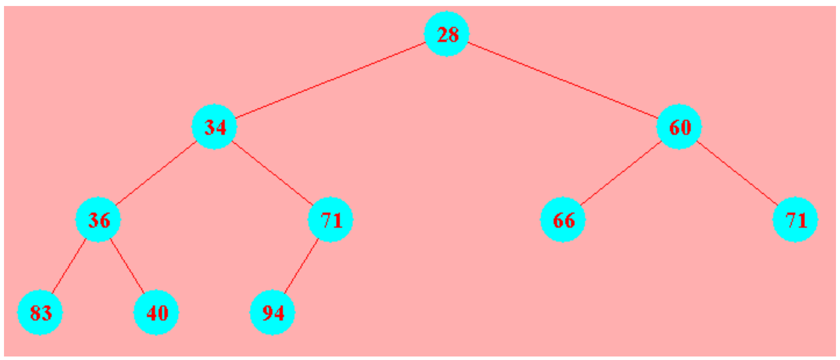
Heaps

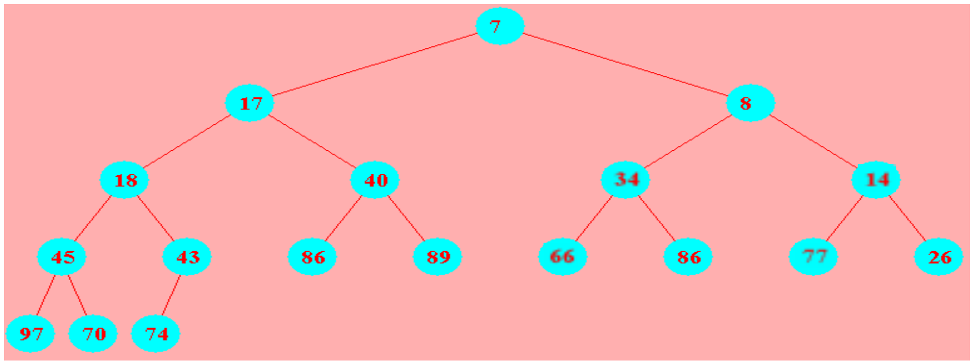
a *complete tree* is a tree in which

- all leaves are on the same level or else on 2 adjacent levels
- all leaves at the lowest level are as far left as possible

a *heap* is complete binary tree in which

- for every node, the value stored is ≤ the values stored in both subtrees
  (technically, this is a min-heap -- can also define a max-heap where the value is ≥)





since complete, a heap has minimal height = $\lfloor \log_2 N \rfloor + 1$

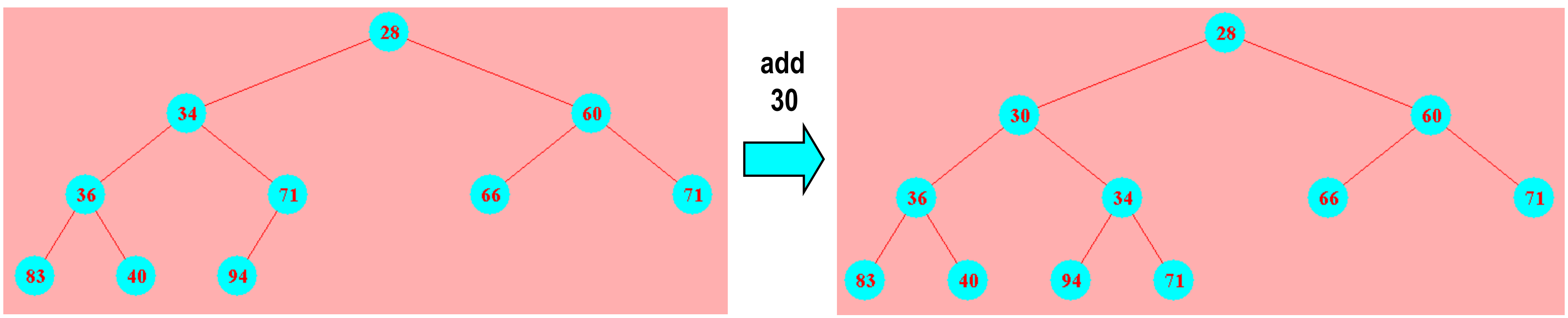
- can insert in O(height) = O(log N), but searching is O(N)
- not good for general storage, but perfect for implementing priority queues
  can access min value in O(1), remove min value in O(height) = O(log N)

Inserting into a heap

to insert into a heap

- place new item in next open leaf position
- if new value is smaller than parent, then swap nodes
- continue up toward the root, swapping with parent, until smaller parent found

see http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MinHeapAppl.html



note: insertion maintains completeness and the heap property

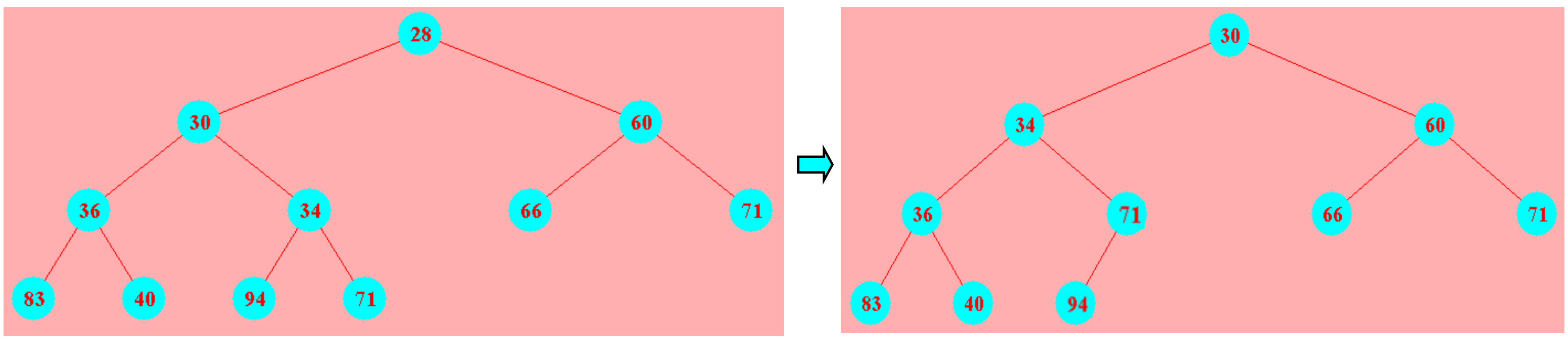
- worst case, if add smallest value, will have to swap all the way up to the root
- but only nodes on the path are swapped → O(height) = O(log N) swaps

Removing from a heap

to remove the min value (root) of a heap

- replace root with last node on bottom level
- if new root value is greater than either child, swap with smaller child
- continue down toward the leaves, swapping with smaller child, until smallest

see http://www.cosc.canterbury.ac.nz/people/mukundan/dsal/MinHeapAppl.html



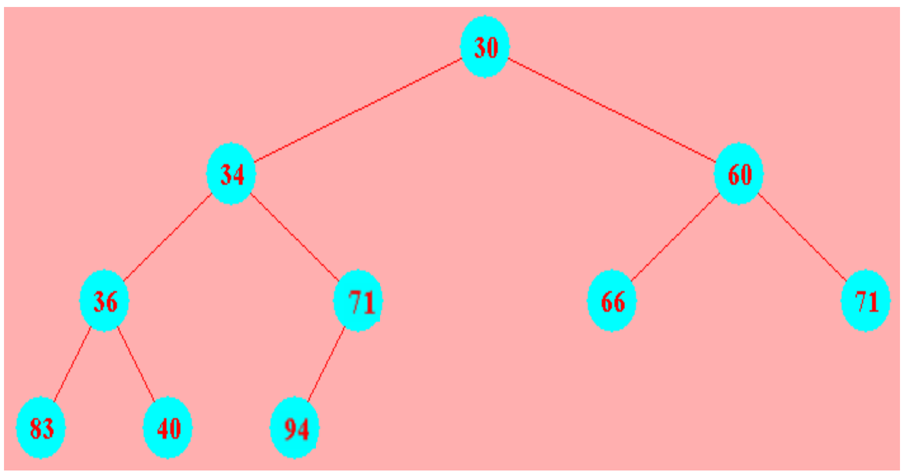
note: removing root maintains completeness and the heap property

- worst case, if last value is largest, will have to swap all the way down to leaf
- but only nodes on the path are swapped → O(height) = O(log N) swaps

Implementing a heap

a heap provides for O(1) find min, O(log N) insertion and min removal

- also has a simple, List-based implementation
- since there are no holes in a heap, can store nodes in an ArrayList, level-by-level



30	34	60	36	71	66	71	83	40	94

- root is at index `0`
- last leaf is at index `size()-1`
- for a node at index `i`, children are at `2*i+1` and `2*i+2`
- to add at next available leaf, simply add at end

`MinHeap` class

```
import java.util.ArrayList;

public class MinHeap<E extends Comparable<? super E>> {
    private ArrayList<E> values;

    public MinHeap() {
        this.values = new ArrayList<E>();
    }

    public E minValue() {
        if (this.values.size() == 0) {
            throw new java.util.NoSuchElementException();
        }
        return this.values.get(0);
    }

    public void add(E newValue) {
        this.values.add(newValue);
        int pos = this.values.size()-1;

        while (pos > 0) {
            if (newValue.compareTo(this.values.get((pos-1)/2)) < 0) {
                this.values.set(pos, this.values.get((pos-1)/2));
                pos = (pos-1)/2;
            }
            else {
                break;
            }
        }
        this.values.set(pos, newValue);
    }

    . . .
```

we can define our own simple min-heap implementation

- `minValue` returns the value at index 0
- `add` places the new value at the next available leaf (i.e., end of list), then moves upward until in position

MinHeap class (cont.)

```
. . .

public void remove() {
    E newValue = this.values.remove(this.values.size()-1);
    int pos = 0;

    if (this.values.size() > 0) {
        while (2*pos+1 < this.values.size()) {
            int minChild = 2*pos+1;
            if (2*pos+2 < this.values.size() &&
                  this.values.get(2*pos+2).compareTo(this.values.get(2*pos+1)) < 0) {
                minChild = 2*pos+2;
            }

            if (newValue.compareTo(this.values.get(minChild)) > 0) {
                this.values.set(pos, this.values.get(minChild));
                pos = minChild;
            }
            else {
                break;
            }
        }
        this.values.set(pos, newValue);
    }
}
```

- `remove` removes the last leaf (i.e., last index), copies its value to the root, and then moves downward until in position

Heap sort

the priority queue nature of heaps suggests an efficient sorting algorithm

- start with the ArrayList to be sorted
- construct a heap out of the elements
- repeatedly, remove min element and put back into the ArrayList

```
public static <E extends Comparable<? super E>>
void heapSort(ArrayList<E> items) {
    MinHeap<E> itemHeap = new MyMinHeap<E>();

    for (int i = 0; i < items.size(); i++) {
        itemHeap.add(items.get(i));
    }

    for (int i = 0; i < items.size(); i++) {
        items.set(i, itemHeap.minValue());
        itemHeap.remove();
    }
}
```

- N items in list, each insertion can require O(log N) swaps to reheapify

  →construct heap in O(N log N)

- N items in heap, each removal can require O(log N) swap to reheapify

  →copy back in O(N log N)

thus, overall efficiency is O(N log N), which is as good as it gets!

- can also implement so that the sorting is done in place, requires no extra storage