CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Transform & conquer

- transform-and-conquer approach
- balanced search trees
  - AVL, 2-3 trees, red-black trees
  - TreeSet & TreeMap implementations
- priority queues
  - heaps
  - heap sort

1

Transform & conquer

the idea behind transform-and-conquer is to transform the given problem into a slightly different problem that suffices

in order to implement an O(log N) binary search tree, don't really need to implement add/remove to ensure perfect balance

- it suffices to ensure O(log N) height, not necessarily minimum height

transform the problem of "tree balance" to "relative tree balance"

several specialized structures/algorithms exist:

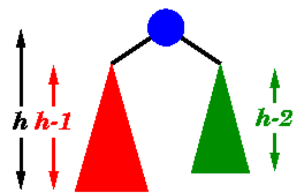
- AVL trees
- 2-3 trees
- red-black trees

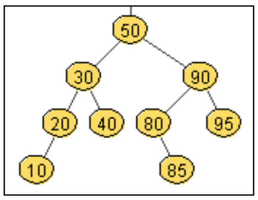
2

AVL trees

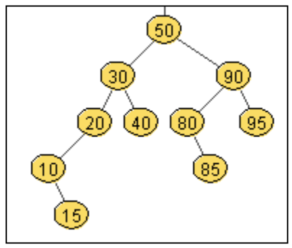
an AVL tree is a binary search tree where

- 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



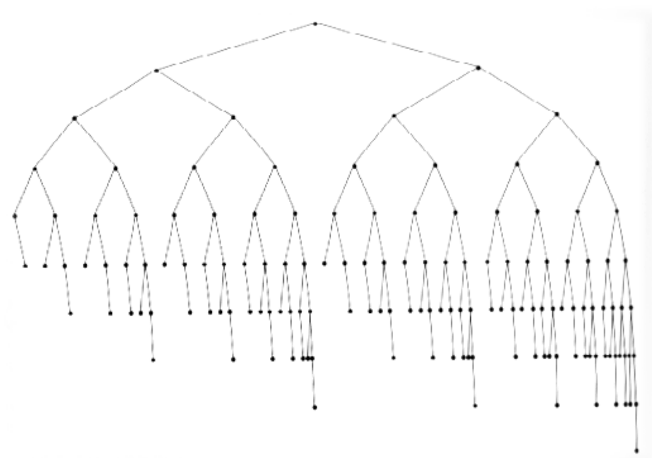
not an AVL tree – WHY?

3

AVL trees and balance

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

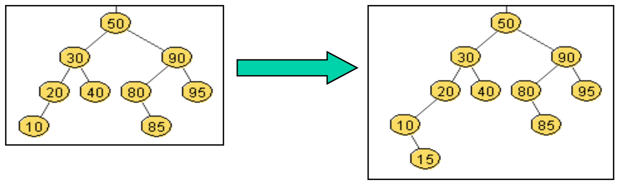
- height of AVL tree with N nodes < 2 log(N+2) → searching is O(log N)



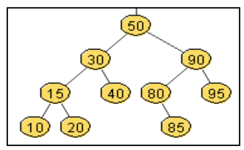
4

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

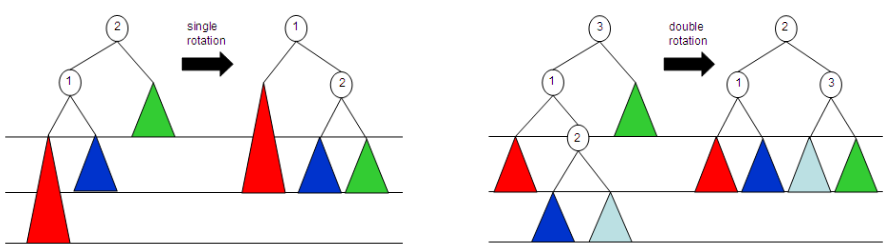


- see www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

5

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)

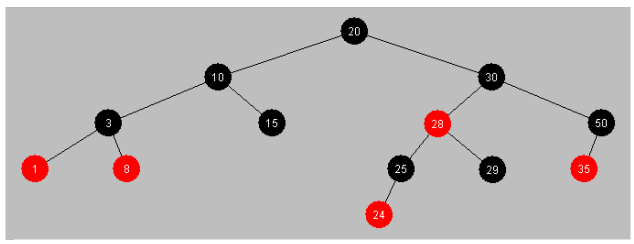
6

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

7

TreeSets & TreeMaps

`java.util.TreeSet` uses *red-black trees* to store values

➔ O(log N) efficiency on add, remove, contains

`java.util.TreeMap` uses *red-black trees* to store the key-value pairs

➔ O(log N) efficiency on put, get, containsKey

thus, the original goal of an efficient tree structure is met

- even though the subgoal of balancing a tree was transformed into "relatively balancing" a tree

8

Scheduling applications

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
- selecting the next disk sector to access from among a backlog
- multiprogramming/multitasking

what all of these applications have in common is:

- a collections of actions/options, each with a priority
- must be able to:
  - ✓ add a new action/option with a given priority to the collection
  - ✓ at a given time, find the highest priority option
  - ✓ remove that highest priority option from the collection

9

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?

10

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() { … }
    . . .
}
```

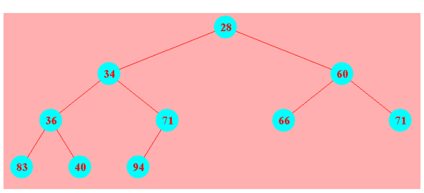
the underlying data structure is a special kind of binary tree called a *heap*

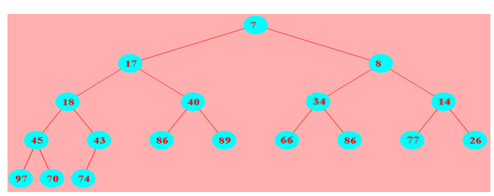
11

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)

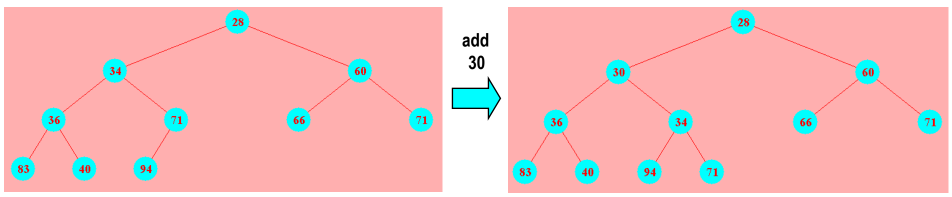
12

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

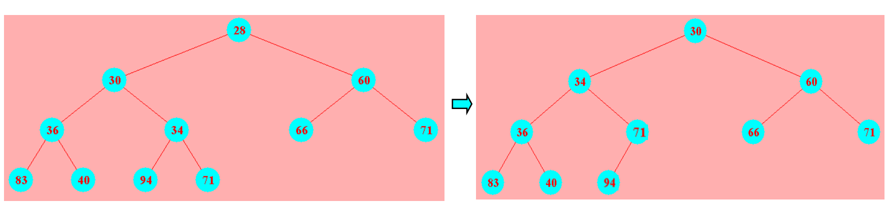
- worst case, if add smallest value, will have to swap all the way up to the root
- but only nodes on the path are swapped → O(height) = O(log N) swaps

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



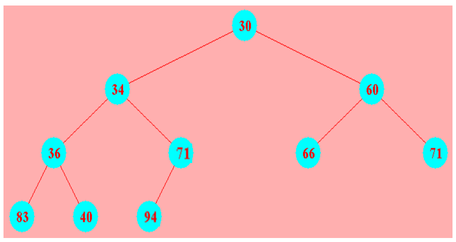
note: removing root maintains completeness and the heap property

- 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;
import java.util.NoSuchElementException;

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 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