Worksheet 1 – Asymptotic Notation

Counting Steps

When analyzing the efficiency of an algorithm, typically depending on the machine, the input type and the algorithm, one might define how the steps are counted differently.

Suppose we use the convention given below:

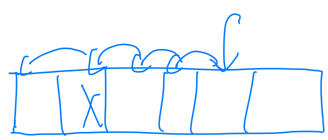
- **read, write** variables, 1 each
- **method call** 1 + steps to evaluate *each argument* + steps to *execute method*
- **return statement** 1 + steps to evaluate *return value*
- **if statement**, **while statement** (not the entire loop) 1 + steps to evaluate *exit condition*
- **assignment statement** 1 + steps to evaluate *each side*
- **arithmetic, comparison, boolean operators** 1 + steps to evaluate each *operand*
- **array access** 1 + steps to evaluate *index*
- **constants** free!

Then we could count the steps fo the algorithm INSERTION SORT:

Precondition. A is an array of integers.
Postcondition A sorted in non-decreasing order.

```
def IS (A):                                  STEPS
    i = 1                                     2
    while (i < len(A))                        5    1 extra time
        t = A[i];                             5
        j = i;                                3
        while (j > 0 AND A[j-1] > t)          10
            A[j] = A[j-1];                    8
            j = j-1;                          4
        A[j] = t;                             5
        i = i+1;                              4
```

i: 1 to n-1



Practice. Enter in the steps for each line.

Q. In terms of n, the length of the array A, how many steps does this algorithm take?

A. Outer loop: $5+5+3+5+4$ steps $n-1$ times. $= 22(n-1)$
extra while loop: $+5$. $\therefore$ total is $22n-17$

Inner: if $j=1 \to$ 1 time through + extra loop while check
$\to 1 \times 22 + 10$
$j=2 \to 2 \times 22 + 10$ ... $j=k \to k \cdot 22 + 10$

Was that fun? imagine if you had a complicated algorithm...

inner loop $\to$ we add all those lines up for $j = 1 .. n-1$

$$\sum_{j=1}^{n-1} 22j + 10 = 22\sum_{j=1}^{n-1} j + \sum_{j=1}^{n-1} 10 = \frac{22(n-1)(n)}{2} + 10(n-1)$$

Worksheet 2 – AVL Trees

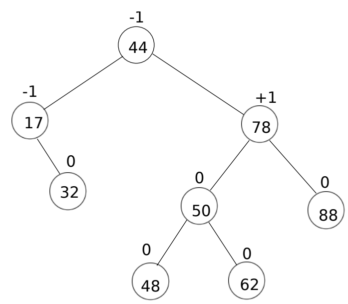
An AVL tree is similar to a *BST* in that it

- stores values in the *internal nodes* and
- has a property *relating* the values stored in a *subtree* to the values in the *parent node*.

but different from a *BST* because

- The height of an *AVL* tree is $\mathcal{O}(\log n)$.
- Each *internal node* has a *balance property* equal to -1, 0, 1.
- Balance value = *height* of the *left* subtree - *height* of the *right* subtree.

AVL Tree Operations – Insert

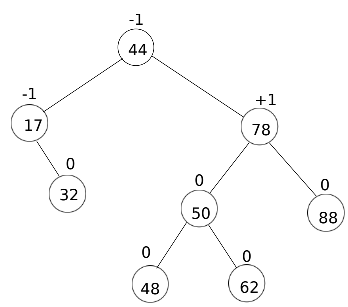


- *Searching* in an AVL tree is the *same* as a BST.
- Consider *inserting* 6 into the tree above.

Q. What are the new *balance factors* in the tree after inserting 6?

A. We will update the tree above.

Q. Let's insert 35 and update the *balance factors* on the tree below.



Q. What problem occurs? We resolve the problem by doing a **single rotation**.

A.

Week 3 Lecture 2 Worksheet Interval Trees

Collections of Intervals

Scenario. You have a set of *time intervals* representing when TA's have office hours.

Closed time intervals: $\{x \in \mathbb{R} \mid l \leq x \leq h\} = [l,h]$.

Representation: Just use l and h.

Operations:

- $insert(l,h)$: Store $[l,h]$ in the collection.
- $delete(l,h)$: Delete $[l,h]$.
- $search(l,h)$: Return a *stored interval* that *overlaps* with $[l,h]$.

  Search represents finding when a TA is available when you are.

Goal. Want $O(\lg n)$ time each.

The data structure

Q. How can we do this?

A. Use a *balanced binary search tree* (AVL, Red Black Tree, weight balanced tree ...) to store the intervals.

Q. For BST order, how do we *compare* $[l,h]$ with $[l',h']$?

- If $l < l'$, then $[l,h] < [l',h']$.
- If $l = l'$ and $h < h'$, then $[l,h] < [l',h']$.

Q. Is this *sufficient*?

A. Easy to see that *insert* and *delete* work nicely. What about *search*?

Each node x_i stores:

- l_i and h_i: interval's two ends, and the key

Example (from textbook)

Graphs - Worksheet

Draw a picture of the *directed graph* represented by the adjacency lists on the left.

1	2, 3
2	7
3	6, 7
4	1
5	4
6	8
7	5, 8
8	4

4

6 8

1 3

7 5

2

Draw a picture of the *breadth-first tree* of the graph, using the same adjacency lists.

4

6 8

1 3

7 5

2

Draw a picture of the *depth-first tree* of the graph, using the same adjacency lists but with vertex 1's adjacently list now being [3, 2] instead of [2, 3].

4

6 8

1 3

7 5

2

Add in the discovery time and finish time for each node of your DFS tree.