CSC263 Assignment 1

Juanxi Li

May 19, 2018

Sources Consulted

- Lecture 1 Notes - Introduction, Complexity Review
- `https://medium.com/@ssbothwell/counting-inversions-with-merge-sort-4d9910dc95f0`
- Delete a node from Binary Search Tree: `https://www.youtube.com/watch?v=gcULXE7ViZw`

Problems

1. Inversions

   (a) (3,4), (3,5) and (4,5) where A[3] = 7, A[4] = 5, and A[5] = 1.

   (b) The completely reverse-sorted descending-order array with items {n, n-1, n-2, ... , 2, 1} has the most inversions. It has $1 + 2 + 3 + ... + (n-1) + n$ inversions, or $\frac{n(n-1)}{2}$. This is also the worst-case runtime of insertion sort.

   (c) Runtime of insertion sort, above the best case θ(n), is proportional to # of inversions.

   Pseudo-code of insertion sort from lecture 1 uses an outer loop j which traverses the array from 2 to n. This loop runs θ(n) times.

   Inner loop **i** begins at **j-1** and traverses backwards, swapping elements as long as A[i] > A[j] (or, in other words, if an inversion exists). Therefore the # of inversions in the array = # of swaps that must be made by the inner loop during insertion sort.

   (d) Solution is mergesort, which runs in $\theta(n\log n)$, with an added line of code to increment an inversion counter during the merge step. This line runs in constant time $\theta(1)$; therefore the entire algorithm still runs in $\theta(n\log n)$.

Recursively, total inversions = inversions encountered in sorting left half + inversions encountered in sorting right half + inversions between the two halves.

During `merge()`, the left and right halves are already sorted, and their inversions will already have been recursively counted. So additional inversions can only exist *between* the two halves. When `merge()` copies a value from the right half back to the main array, it is necessarily an inversion relative to any numbers that remain in the left half. So this is the amount by which we increase the counter.

Derived from pseudo-code in Lecture 1 slides:

```
mergeSortInversions(A,p,r) {
    inv = 0; // variable to track inversions,
                // gets passed to merge()
    if (p < r) {
        q = floor((p+r)/2)
        mergeSortInversions(A,p,q)
        mergeSortInversions(A, q+1, r)
        merge(A,p,q,r,inv)
    }
    return inv;
}

merge(A, p, q, r, inv) {
    n1 = q - p + 1
    n2 = r - q
    copy A[p,q] to L[1...n1]
    copy A[q+1,r] to R[1...n2]
    L[n1+1] = R[n1+1] = +Inf
    i = j = 1
    for (k=p to r) {
        if (L[i] < R[j]) {
            A[k] = L[i]
            i++
        }
        else {
            // a value R[j] was taken from the right
            // side in the process of merging;
            // R[j] must be an inversion relative to
            // everything remaining on the left side
            A[k] = R[j]
            inv += L.length - i
            j++
```

```
            }
        }
    }
```

2. Extract second largest

   (a) If using an implementation of nodes which store pointer to parent, we simply run `max()` and return its parent's value.

       Otherwise, we can modify `max()` while retaining the same runtime efficiency aka θ(h), or θ(logn) on a perfectly balanced binary tree.

       Add a variable which tracks the parent of the current node being checked. Return the value of this variable once max is found. Since this line runs in constant time, we still maintain θ(logn) runtime.