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) This is the same thing as finding max in array Q and deleting the larger of its two children. Here is a modified version of `extractMax()` from lecture which accompishes this.

```
extractSecondLargest(Q) {
  int secondMax;
  int maxVal;

  // max is at 1, its children are at positions 2 and 3

  if (Q[2] == NULL) {      // second largest doesn't exist
    return NULL;
  }
  else if (Q[3] == NULL) { // Q only has two items
    secondMax = 2;
  }
  else {                   // compare Q2 and Q3
    if (Q[2] >= Q[3]) { secondMax = 2; }
    else              { secondMax = 3; }
  }

  maxVal = Q[secondMax];

  // swap w last, and delete
  Q[secondMax] = Q[Q.heapsize];
  Q.heapsize = Q.heapsize - 1;
  maxHeapify(Q, secondMax);

  return maxVal;
}
```

   (b)
   - **Why element is second largest:** because in our implicit definition of the heap as an array, the maximum element is always stored in position 1, and its children are stored in positions 2 and 3.

     We compare nodes 2 and 3, returning the larger value. This is the second-largest value, behind max.

   - **Why maintains heap property:** Shape is maintained by swapping the second-largest with the last item and removing the last

item. Heap property is maintained by calling maxHeapify() on the position previously occupied by the second-largest, which bubbles that value down until heap property is restored.

- **Why worst-case run-time is O(logn):** Locating the second-largest item and swapping with last item occurs in constant time. max() is found in $\theta(1)$ since it's just the first item, and we make at most one comparision between the children at Q[2] and Q[3] if they exist. The swap takes constant time.

  `maxHeapify()` bubbles down at most h times where h is height of tree, which is $log(n)$ for binary heap.

3. Heap Delete

   (a) This will be very similar to the pseudocode from Q2, but we delete item i instead of item 2 or 3. I assume the function should return the item.

```
heapDelete(A, i) {
  int iVal = A[i];

  // swap w last, and delete
  A[i] = A[A.heapsize];
  A.heapsize = A.heapsize - 1;
  maxHeapify(A, i);

  return iVal;
}
```

   (b)
   - **Why works:** Swapping ith node with last element, and then deleting last element (by shortening the array) maintains heap shape. Calling `maxHeapify()` on the subtree rooted at i ensures that the new ith element bubbles down until the heap property is restored. Outside of this subtree, no other part of the heap is changed.
   - **Why O(logn):** Location of node to be removed is already known and supplied via integer i. Swap takes constant time. After swap, `maxHeapify()` bubbles element i down max h_i times, where h_i is height of subtree rooted at i. Worst case, i is the root and h_i = height of entire tree, which is $log(n)$ for complete binary tree.