Part I Solution

1. Hash tables:

The expected number of probes is obtained by plugging in $\alpha = 3/4$ and $\alpha = 7/8$ into Knuth's formulas (please refer to the class slides).

2. Heapsort correctness:

For reference, the heapsort algorithm is shown below:

```
function Heapsort(arr)
    Heapify array arr;
    for $i = n - 1$ down to 1 do
        swap arr[$i$] with arr[0];
        restore heap property for the tree arr[0], ..., arr[$i-1$] by percolating down
          the root;
    end
end
```

- Recall that heapsort is similar to selection sort. At each iteration, the maximum element from the subarray under consideration is moved to its correct spot. The heap property is used to efficiently find the maximum element which is always at the root of the heap. We can therefore use the following loop invariant.

  Loop Invariant:
  Before iteration i of the for loop, the subarray ***arr***$[0, \ldots, i]$ *is a max-heap containing the $(i+1)$ smallest elements of* ***arr*** *and the subarray* ***arr***$[i+1, \ldots, n-1]$ *contains the $n-(i+1)$ largest elements of* ***arr*** *in ascending sorted order.*

- To show that the above loop invariant holds, we must show the initialization and maintenance steps.
  Initialization: Before the very first iteration, $i = n-1$. Since the entire array `arr` is heapified on line 2, we have that `arr`$[0, \ldots, n-1]$ is a max-heap. Furthermore, it contains the n smallest elements of `arr`. The subarray `arr`$[n, \ldots, n-1]$ is an empty array, so the second part of the loop invariant is vacuously true.
  Maintenance: Suppose that the loop invariant holds before iteration i and the loop runs one more time (i.e. we are just before iteration $i-1$). From the loop invariant, we know that `arr`$[0, \ldots, i] \leq$ `arr`$[i+1, \ldots, n-1]$ and that `arr`$[i+1] \leq$ `arr`$[i+2] \leq \cdots \leq$ `arr`$[n-1]$. Since `arr`$[0, \ldots, i]$ is a max-heap, we also know that `arr`$[0]$ is the largest element in `arr`$[0, \ldots, i]$.
  Line 4 swaps `arr`$[i]$ with `arr`$[0]$. Consequently, after line 4 executes, we have that `arr`$[0, \ldots, i-1] \leq$ `arr`$[i, \ldots, n-1]$ and that `arr`$[i] \leq$ `arr`$[i+1] \leq \cdots \leq$ `arr`$[n-1]$. This shows the second part of the loop-invariant.
  Line 5 restores the heap property on `arr`$[0, \ldots, i-1]$. Consequently, the subarray `arr`$[0, \ldots, i-1]$ is now a max-heap. Furthermore, since the largest element in `arr`$[0, \ldots, i]$ was moved to `arr`$[i]$, the subarray `arr`$[0, \ldots, i-1]$ now contains the i smallest elements of `arr`. This shows the first part of the loop-invariant.

- We'll show the post-condition that the array `arr` is sorted in ascending order upon termination. Upon termination, we are just before the $i = 0$ iteration. From the first part of the loop invariant, we know that `arr`$[0]$ is the smallest element in the array. Furthermore, from the second part of the loop invariant, we know that `arr`$[1] \leq$ `arr`$[2] \leq \cdots \leq$ `arr`$[n-1]$. Combining these two, we have that:
  `arr`$[0] \leq$ `arr`$[1] \leq \cdots \leq$ `arr`$[n-1]$.

3. Quicksort:

- $$T(n) \leq \begin{cases} c\,n^2, & n < K, \\ T(n-K) + T(K-1) + d\,n, & n \geq K. \end{cases}$$

- Observe that $T(K-1) = c(K-1)^2$. Thus, we can work with the following recurrence relation to obtain an upper bound.

  $$R(n) = R(n-K) + c\,(K-1)^2 + d\,n$$

  Using the substitution method, we obtain:

  $$\begin{aligned} R(n) &= R(n-K) + c\,(K-1)^2 + d\,n \\ &= R(n-2K) + c\,(K-1)^2 + d\,(n-K) + c\,(K-1)^2 + d\,n \\ &= R(n-2K) + d\big((n-K)+n\big) + 2c(K-1)^2 \\ &\vdots \\ &= R(n-iK) + d\sum_{j=0}^{i-1}(n-jK) + i\,c(K-1)^2 \end{aligned}$$

  The recurrence ends when $n - iK = 0$ (for a partition of size 0) or when $i = n/K$. Assuming that $R(0) = b$ (for a constant b), we obtain:

  $$\begin{aligned} R(n) &= b + d\frac{n}{2K}(n+K) + c\,\frac{n}{K}(K-1)^2 \\ &\le b + d\frac{n}{2K}(n+K) + c\,nK. \end{aligned}$$

  From the solution of the recurrence relation (dropping constants), we obtain:

  $$T(n) = O(\frac{n^2}{K} + nK).$$

- This will depend on the choice of K. When $K = 1$, we see that $T(n) = O(n^2)$. This is equivalent to the worst-case of quicksort. When $K = n$, $T(n) = O(n^2)$ which is what we would expect as in this case, most of the work is done by insertion sort whose worst-case complexity is $O(n^2)$. However, by choosing K to be a small value, we can reduce the impact of the quadratic term. This is a choice that is made in hybrid versions of quick sort that switch to insertion sort for small partition sizes - typically $K = 10$.

4. A tree with n vertices has $n - 1$ edges.

   We can prove this by strong induction. A tree with 1 vertex has no edges, so the base case holds. Suppose we have a tree with n vertices where $n > 1$. Pick any edge in the tree and remove it. This will create two trees. Let the number of vertices in these two trees be n_1 and n_2. Clearly, $n_1 + n_2 = n$. From the induction hypothesis we know that the number of edges in these two trees are $n_1 - 1$ and $n_2 - 1$ respectively. Now reintroduce the removed edge. The total number of edges now is $(n_1 - 1) + (n_2 - 1) + 1 = n_1 + n_2 - 1 = n - 1$.