Homework 3

Problem 5

1. Assume there are two global variables: H, a hash table, and G, a graph data structure. The traverse function takes T as an input, where T is the root of an AST. Each node of the AST has left (l) and right (r) child. At leaves, l and r are both null pointers.
   traverse(T):
   1. **if** T = **null**: **return** hash(**null**)
   2. $T.l$.key := traverse($T.l$)
   3. $T.r$.key := traverse($T.r$)
   4. T.key := hash(T.attribute, $T.l$.key, $T.r$.key)
   5. H.insert(T.key)
   6. G.insert((T.key, $T.l$.key))
   7. G.insert((T.key, $T.r$.key))
   8. **return** T.key
2. Claim: Two nodes have the same hash value if and only if they have the same attribute, and their left and right children are equivalent. By mathematical induction on the depth of subtree, we hypothesize that for any two subtrees T_1 and T_2 of depth $\leq i$, T_1 and T_2 are equivalent if and only if T_1.key = T_2.key.
   Basis: Two subtrees of depth 0 are both null pointers, and have the same hash value of hash(**null**).
   IS: Given two subtrees T_1' and T_2' of depth $i+1$.
   $\Rightarrow$: If T_1' and T_2' are equivalent, then T_1' and T_2' have the same attribute, $T_1'.l$ and $T_2'.l$ are equivalent, and $T_1'.r$ and $T_2'.r$ are equivalent. Since $T_1'.l$, $T_1'.r$, $T_2'.l$ and $T_2'.r$ have depth $\leq i$, it follows that $T_1'.l$.key = $T_2'.l$.key, and $T_1'.r$.key = $T_2'.r$.key. Hence, T_1'.key = hash(T_1'.attribute, $T_1'.l$.key, $T_1'.r$.key) = hash(T_2'.attribute, $T_2'.l$.key, $T_2'.r$.key) = T_2'.key
   $\Leftarrow$: If T_1'.key = T_2'.key, then it follows that T_1'.attribute = T_2'.attribute, $T_1'.l$.key = $T_2'.l$.key, and $T_1'.r$.key = $T_2'.r$.key. Since $T_1'.l$, $T_1'.r$, $T_2'.l$ and $T_2'.r$ have depth $\leq i$, it follows that $T_1'.l$ and $T_2'.l$ are equivalent, and $T_1'.r$ and $T_2'.r$ are equivalent. Hence, T_1' and T_2' are equivalent.
   Conclusion: For any two subtrees T_1 and T_2 of arbitrary depth, T_1 and T_2 are equivalent if and only if T_1.key = T_2.key. Hence, the number of keys in H is exactly the number of different subtrees in the AST. Also, each different subtree should represent at least one unique vertex in G. Hence, the number of keys in H is exactly the minimum number of nodes for G.
3. The provided pseudocode is a post-order tree traversal algorithm (DFS on the tree). Line 2 and Line 3 traverse the entire tree recursively. Line 4 involves hash value calculation, which takes $O(1)$. Line 5 involves hash table insertion, which takes $O(1)$. Line 6 and Line 7 involve edge insertion into G, which takes $O(1)$. The time spent at each node is constant. Line 2 and Line 3 traverse each of the N nodes constant times. Hence, the time complexity is $O(N)$.

Problem 6

1. T contains $|V| - 1$ edges. If T is not a tree, then there must exist a cycle $C = (V_C, E_C)$ in T. Removing any edge $e \in C$ must yield a lower cost while T is still connected, since V_C is still connected through all the other edges, i.e., $E_C - \{e\}$.
2. Given an MST T. Removing an arbitrary edge $(u, v) \in T$ partitions T into two subtrees T_1 and T_2. The weights of T, T_1 and T_2 have the following relationship: $w(T) = w(u, v) + w(T_1) + w(T_2)$. Also, let G_1 and G_2 be the subgraph induced by the vertices of T_1 and T_2, respectively. WLOG, assume T_1' be a lower-weight spanning tree than T_1 for G_1. Then, $T' = (u, v) \cup T_1' \cup T_2$ would be a lower-weight spanning tree than T for G, which is a contradiction. Hence, any subtree of MST is also optimal.
3. Let $E = \{e_1, e_2, \ldots, e_{|E|}\}$ be a list of edges in non-increasing order of weight, and T_i be the graph after i-th iteration of while loop. First, it is hypothesized that after i-th iteration of while loop, there exists an MST $T \subseteq T_i$. Basis: T_0 is the entire graph G, and $T \subseteq T_0$ is for sure. Inductive step: At $(i + 1)$-th iteration, e_{i+1} is considered. If removing e_{i+1} would disconnect T_i, then e_{i+1} is reserved. Hence, $T \subseteq T_{i+1} = T_i$. Otherwise, e_{i+1} is removed from T_i. If $e_{i+1} \notin T$, then $T \subseteq T_{i+1} = T_i - \{e_{i+1}\}$. Otherwise, $e_{i+1} \in T$. Since removing e_{i+1} does not disconnect T_i, there must exist a cycle C such that $e_{i+1} \in C \subseteq T_i$. Also, removing e_{i+1} would partition T into T_1 and T_2. However, T_1 and T_2 can be connected through another edge $e' \in C$ such that $T' = T_1 + T_2 + \{e'\} = T - \{e_{i+1}\} + \{e'\}$. It is claimed that T' is also an MST. T' itself is a valid spanning tree since only e' but not e_{i+1} is in T'. Next, the weights of e_{i+1} and e' have the relationship: e_{i+1}.weight ≥ e'.weight since $e' \in \{e_{i+2}, \ldots, e_{|E|}\}$. Hence, T'.weight = T.weight - e_{i+1}.weight + e'.weight ≤ T.weight. T' is at least as best as T. Hence, T' is also minimum. Conclusion: after the while loop finishes, there is an MST $T \subseteq T_{|E|}$. $T_{|E|}$ is a spanning tree since any cycle will always be destroyed while $T_{|E|}$ remains connected. Hence, $T_{|E|}$ itself is an MST.
4. A complete graph of n vertices has $\frac{n(n-1)}{2}$ edges.
   Kruskal: $O(|E| \log |V|) = O(\frac{n(n-1)}{2} \log n) = O(n^2 \log n)$.
   Prim using Fibonacci heap: $O(|E| + |V| \log |V|) = O(\frac{n(n-1)}{2} + n \log n) = O(n^2)$.
   Comparison: Prim using Fibonacci heap is asymptotically better than Kruskal.
5. A planar graph of $n \geq 3$ vertices has at most $3n - 6$ edges.
   Kruskal: $O(|E| \log |V|) = O((3n - 6) \log n) = O(n \log n)$.
   Prim using Fibonacci heap: $O(|E| + |V| \log |V|) = O((3n - 6) + n \log n) = O(n \log n)$.
   Comparison: Prim using Fibonacci heap and Kruskal are asymptotically equivalent.