Due: Thu Apr 25th at the start of class

Homework #4

CMSC351 - Spring 2019

PRINT Name:______________________________:

- Grades depend on neatness and clarity.
- Write your answers with enough detail about your approach and concepts used, so that the grader will be able to understand it easily. You should ALWAYS prove the correctness of your algorithms either directly or by referring to a proof in the book.
- Write your answers in the spaces provided. If needed, attach other pages.
- The grades would be out of 16. Four problems would be selected and everyone’s grade would be based only on those problems. You will also get 4 bonus points for trying to solve all problems.

1. Give short answers to these questions:
   - Sum of degrees of a connected graph with n vertices is at least __2n – 2_____
   - What is the number of edges of a complete bipartite graph whose vertices is partitioned into two independent sets V₁ and V₂? (based on |V₁| and |V₂|) ___|V₁| × |V₂|______
   - What is the minimum number of vertices of a graph with 20 edges? ___7_____
   - In which dynamic data structures we can do insert and delete in average O(1) time? ___Hashing_____

-
2. Given two sets $S_1$ and $S_2$, and a real number $x$, find whether there exists an element from $S_1$ and an element from $S_2$ whose sum is exactly $x$. The algorithm should run in average time $O(n)$, where $n$ is the total number of elements in both sets. (Try to use the data structures you have learned in the lectures.)

First we should use a Hashing data structure to store all the members of $S_1$. For every member $u$ of $S_2$, we simply search for $x - u$ in the data structure. In a hash data structure the running time of insertion and search are $O(1)$ in average, thus the algorithm runs in $O(n)$ time in average.
3. A proper coloring of a graph is a way of coloring the vertices of a graph such that no two adjacent vertices have the same color. For a graph $G$, $\Delta(G)$ denotes the maximum degree of a vertex in $G$. Design an algorithm that given a graph $G = (V, E)$, finds a proper coloring of $G$ with at most $\Delta(G) + 1$ colors in time $O(|V| + |E|)$. (Hint: Use the adjacency list representation of $G$).

Consider an arbitrary sequence of vertices $v_1, \ldots, v_n$. Starting from $v_1$, we color $v_i$ with the minimum natural number $x$, such that none of the (colored) neighbors of $v_i$ are colored $x$. Since the degree of $v_i$ is at most $\Delta$, $x$ is at most $\Delta + 1$. Therefore we can color all of the vertices with at most $\Delta + 1$ colors.
4. [Prob 4.16, Pg 88] Design a data structure that supports the following operations. Each operation should take \( O(\lg n) \) time in the worst case where \( n \) is the number of elements in the data structure. Explain the required process for doing each operation and prove that it takes \( O(\lg n) \) time.

   a) \textit{Insert}(x): Insert the key \( x \) into data structure only if it is not already there.
   
   b) \textit{Delete}(x): delete the key \( x \), if it is there!
   
   c) \textit{Find}_\text{Smallest}(k): find the \( k \)th smallest key in the data structure.

A balanced binary search tree (e.g. AVL tree) can perform the first two types of commands. We modify BST such that we can also perform the third type of command in \( O(\lg n) \). In each node of the tree \( v \), store a number \( \ell_M \) which is equal to the number of vertices in the left subtree of that node. Insertion and Deletion of an AVL tree can be easily adjusted to update these values.

To perform the last command, we implement a recursive function \( f_s(v, k) \) which finds the \( k \)th element in the subtree rooted at node \( v \). Clearly the answer to operation (c) would be \( f_s(root, k) \).

Now when we want to find the \( k \)th smallest element, we simply check \( \ell_M \). If \( \ell_M = k - 1 \), then node \( v \) is the answer to \( f_s(v, k) \). If \( \ell_M < k - 1 \), the answer would be in the right subtree of \( v \). Thus the answer would be \( f_s(right\_child\_of\_v, k - \ell_M - 1) \). Similarly, if \( \ell_M > k - 1 \), the answer would be \( f_s(left\_child\_of\_v, k) \).
a) Determine the general structure of a simple binary search tree formed by inserting the numbers 1 to \( n \) in order (you may draw a general structure, but explain it properly). What is the height of this tree?

It would be a straight line of length \( n - 1 \).

b) Determine the general structure of an AVL search tree formed by inserting the numbers 1 to \( n \) in order. What is the height of this tree?

The height of tree is \([\lg n] - 1\). The tree would be an almost complete binary tree like below.
6. [Prob 6.34, Pg 179] The input is a heap of size \( n \) (in which the largest element is on top), given as an array, and a real number \( x \). Design an algorithm to determine whether the \( k \)th largest element in the heap is less than or equal to \( x \). The worst-case running time of your algorithm should be \( O(k) \), independent of the size of the heap. You can use \( O(k) \) space. (Notice that you do not have to find the \( k \)th largest element; you need only to determine its relationship to \( x \).)

Call the set of numbers in the heap which are greater than \( x \) by “high-nodes”. The question is whether the \( k \)th node is a high-node. The number stored in parent of a node \( u \), is greater than the number stored in \( u \). Thus all the nodes on the path from a high-node to the root, are also high-nodes. Now we start counting the number of high-nodes by starting from the root and walking only through the high-nodes. More precisely, by starting at root, at each vertex \( v \), we recursively go to its left child iff the left child is also a high node. After the left child finished counting and we come back to \( v \), we go to the right child of \( v \) iff the right child is a high-node. After both children are finished, we go back to the parent of \( v \).

We continue this process until we either finish visiting all the high-nodes, or end up traversing through more than \( k \) high-nodes. In the former case, the \( k \)th largest node will not be a high-node and in the latter case, the \( k \)th largest node is indeed a high-node (but we can’t say which node is the actual \( k \)th element). The running time is \( O(k) \) since we will visit at most \( k \) high nodes in either case.
7. By using the concepts of graph theory, prove that removing opposite corner squares from an $8 \times 8$ checkerboard leaves a sub-board that cannot be partitioned into $1 \times 2$ and $2 \times 1$ rectangles.

Color the board with black and white just like a chess board. We will have 32 black cells and 30 white cells (or vice versa). If we put a vertex for each cell and connect two vertices iff their corresponding cells are adjacent, then we would have a bipartite graph (it is bipartite because two adjacent cells do not have the same color). Putting dominos on the board is like choosing an edge in the graph and matching two adjacent cells (which one is black and the other is white). So if want to cover all the cells with dominos, we should partition them into adjacent pairs. One vertex of each pair would be black and the other one would be white. However, the number of black vertices is more than the number of white vertices, so we cannot cover all the vertices with dominos.
8. As we defined in the class, a bipartite graph is a graph whose vertices $V$ can be partitioned into two sets $V_1$ and $V_2$ such that there is no edge between two vertices in $V_1$ and there is no edge between two vertices in $V_2$. Using the definition above prove that a bipartite graph cannot have a simple cycle with an odd number of vertices.

We prove this by contradiction. Let $v_1, \ldots, v_{2k+1}$ be an odd cycle. We call the vertices in $V_1$ and $V_2$ black and white respectively. WLOG, we may assume that $v_1$ is black. Since there is an edge between $v_1$ and $v_2$, the color of $v_2$ is white. By continuing this process, it would be clear that $v_i$ is black iff $i$ is an odd number. Therefore $v_{2k+1}$ is also black and since $v_1$ and $v_{2k+1}$ are adjacent, this would be a contradiction. Therefore a bipartite graph cannot have an odd cycle.
9. [hard] We have a big rectangle room which is tiled by a finite number of non-overlapping rectangles. The sides of all the tiles are parallel to the sides of the room. We know that the length of at least one side of each tile is integer. Prove that at least one side of the room is also integer. (Hint: The sum of the degrees is always even in a graph!)

You can find the answer and an interesting applet here: http://www.cut-the-knot.org/Curriculum/Algebra/IntRectGraph.shtml
You can also find two different interesting solutions here: http://www.cs.toronto.edu/~mackay/rectangles/

The sketch of the proof is as follows:
Suppose that the bottom-left corner of room is on point (0,0) of the plain. We say a point (x,y) is special, iff both x and y are integer. A crucial observation is that each tile, has 0, 2, or 4 special corners. Now make a bipartite graph $G = (V_1, V_2, E)$ as follows: put a vertex in $V_1$ corresponding to the room. Also for every tile put a corresponding vertex in $V_1$. Now for every special point $(x, y)$ such that $(x, y)$ is a at least corner of one tile, put a vertex in $V_2$. Draw an edge between a vertex in $v_1 \in V_1$ and $v_2 \in V_2$, iff the point corresponding to $v_2$ is a corner of the rectangle corresponding to $v_1$. By our previous observation we know that the degrees of vertices corresponding to the tiles are always even. Also you can show that a point cannot be the corner of odd number of rectangles (i.e., the tiles and the room). Since in a graph, we cannot have just 1 odd vertex, this implies that the degree of the vertex corresponding to the room is also even. However, we know that the bottom-left corner of room is (0,0), thus a special point. This means that the room has either 2 or 4 special corners, and therefore at least one side of room is integer.