**Homework 4**
Show all work.

1. **Self-organizing binary trees:** You have a fixed set of numbers arranged in a binary search tree. You will have no insertion or deletion, but you will do retrieval. On every successful retrieval, you will adjust the tree to move the retrieved number closer to the root. This strategy aims to reduce the expected retrieval time by moving more frequently retrieved items closer to the root. Write pseudocode for the operation of retrieval and tree adjustment. Your pseudocode can call as subroutines the “left-rotate” and “right-rotate” routines given in the Cormen et al. text *Introduction to Algorithms*, chapter “Red-Black Trees,” Section 2, “Rotations.”

**Solution:**

Algorithm Self-organizing-binary-tree-retrieve(Tree T, key K)
node := root(T)
do
    J := key contained at node
    if J = K
        if J < key contained in node’s parent
            rotate node to the right
        else if J > key contained in node’s parent
            rotate node to the left
        return contents of node
    else if key contained at node < K then
        node := right subchild of node
    else
        node := left subchild of node
    until node is a leaf node
return flag for unsuccessful search
end algorithm
2. Recall that a **height-balanced tree** is a binary search tree in which the difference between the height of the left subtree and the right subtree under every node is at most 1.

Let \( V(n) \) be the number of vertices in a height-balanced tree of height \( h \) with the fewest vertices. For example, we have \( V(0) = 1 \) for the tree of height 0 and 1 vertex, and \( V(1) = 2 \) because a tree of height 1 is possible with only 2 vertices.

Write a recurrence for \( V(n) \). The recurrence should have the form:

\[
V(n) = 1 + \text{number of vertices in left subtree} + \text{number of vertices in right subtree}
\]

and you are asked to replace the words with formulas.

**Solution:**

Since the tree is of height \( h \), it must have at least one subtree of height \( h - 1 \). But since we want to minimize the size of the tree, we should take the second subtree as small as possible. Since the constraint on a height-balanced tree is that the difference in height of the two subtrees is at most 1, the smallest we can take the second subtree of height \( h - 2 \). Also, both of the subtrees must also be height-balanced, because the balance constraint applies at every node. We should make these subtrees with as few nodes possible to minimize the total number of nodes. Therefore, the two subtrees are a height-balanced tree of height \( h - 1 \) with minimum number of vertices and a height-balanced tree of height \( h - 2 \) with minimum number of vertices. This gives the recurrence: \( V(h) = 1 + V(h - 1) + V(h - 2) \).

**Comment:** Note the similarity of this recurrence to the Fibonacci recurrence. The solution to this recurrence is an exponential function, which means that even with the minimum number of vertices, the number of vertices is exponential in the height, which means the height is logarithmic in the number of vertices. That makes the algorithms efficient for height-balanced trees.
3. **Transposing a directed graph:** In this exercise, we will study algorithms for reversing the directions of the edges in directed graphs using the two major representations of a graph.

(a) You are given an adjacency matrix for a directed graph \( G \), that is, a matrix \( M \) with \( M[i,j] = E \) if there is a directed edge from vertex \( i \) to vertex \( j \), otherwise it is blank. Give pseudocode for algorithm for constructing the adjacency matrix of the graph \( H \) which is \( G \) with the direction of every edge reversed. Give a big-\( \Theta \) bound for the number of times your algorithm accesses an array cell in terms of \( n \), the number of vertices in the graph.

**Solution:**

```plaintext
Algorithm transpose-edges(matrix M)
    n := number of rows in M
    Allocate a new n-by-n array N
    for i := 1 to n
        for j := 1 to n
            if M[i,j] = E
                then N[j,i] := E
            end if
        end for
    end for
end algorithm
```

This algorithm accesses each array cell one time. Since it is an n-by-n array, the number of times it access an array cell is \( n^2 \), which is \( \Theta(n^2) \).
(b) You are given an array of adjacency lists for a directed graph \( G \), that is, an array containing one list for each vertex, and the list for vertex \( i \) contains \( j \) if and only if there is an edge from \( i \) to \( j \). Give pseudocode for an algorithm to construct a new adjacency list for the graph \( H \) which is \( G \) with the direction of every edge reversed. Give a big-\( \Theta \) bound for the number of times your algorithm accesses a list element in terms of \( e \), the number of edges in the graph.

Solution:

Algorithm transpose-edges(array of adjacency lists \( A \))

\[ n := \text{size of } A \]
Allocate a new array of \( n \) adjacency lists \( B \)
for \( i := 1 \) to \( n \)
   \[ j := \text{first in list of } A[i] \]
   while \( j \) is not null
      insert \( i \) in list of \( B[j] \)
      \( j := \text{next in list of } A[i] \)
   end while
end for
end algorithm

The algorithm accesses each element in the array of adjacency lists one time. Since each element corresponds to an edge, it accesses a list element \( e \) times, which is \( \Theta(e) \).

4. A lattice is a directed acyclic graph such that there is one vertex that has a path to every other vertex (the “top”) and one vertex to which every other vertex has a path (the “bottom”). Describe an algorithm to determine whether a given DAG is a lattice. Give a big-\( \Theta \) bound for the number of times your algorithm accesses a vertex in terms of \( n \), the number of vertices in the graph.

Solution: There are many possible algorithms to solve this problem. We give a discussion of three examples.

Method 1: Brute-force search. For each vertex in the DAG, perform depth-first search to check whether it can reach each other vertex. If a vertex is found which can reach each other vertex, this can serve as the “top.” Then, for each vertex, perform a reverse depth-first search by traversing edges in the opposite direction to check whether it can be reached from every other vertex. If such a vertex is found, it is the “bottom.” If the graph has both a top and a bottom, it is a lattice. This algorithm performs depth-first search twice for each vertex in the graph. A depth-first search visits each vertex at least once plus the number of times of its degree. So the number of times it accesses a vertex is \( \sum_{i=1}^{n} (1 + \deg(v_i)) \). Using the formula relating the sum of all degrees to the number of edges, this is
\[\theta(n + e),\] where \(n\) is the number of vertices and \(e\) is the number of edges. And the number of edges may range from 0 (every vertex is isolated) to \(\frac{n(n-1)}{2}\) (we connect every edge to every other edge without introducing cycles). So in terms of \(n\), each depth-first search is bounded by \(\theta(n^2)\), and the overall algorithm doing \(2n\) depth-first searches is bounded by \(\theta(n^3)\).

**Method 2:** Topological sort. Perform a topological sort of the DAG. Then if the graph is a lattice, the top will be the first vertex in the list of topologically sorted vertices, and the bottom will be the last vertex in the list. Verify that the first vertex is the top by doing a depth-first search from it to see if it can reach every other vertex. Verify that the last vertex is the bottom by doing a reverse-order depth-first search from it to see if it can be reached from every other vertex. If both of these checks are passed, then the graph is a lattice. This algorithm requires three depth-first searches: one for the topological sort and one each for the first and last vertex in topologically sorted order. Thus, the bound is \(\theta(n^2)\).

**Method 3:** We check the indegree and outdegree of each vertex. If there is more than one vertex of indegree 0, then the graph is not a lattice, because these vertices cannot reach each other and so there is no vertex that can reach every other vertex. If there is more than one vertex of outdegree 0, then the graph is not a lattice, because these vertices cannot be reached from each other and so there is no vertex that can be reached from every other vertex. We claim that if there is exactly one vertex of indegree 0 and exactly one vertex of outdegree 0, the graph is a lattice with the vertex of indegree 0 as the top and the vertex of outdegree 0 as the bottom. We must prove this claim. Consider any vertex in the graph except the vertex of indegree 0. Choose one of its incoming edges to one of its predecessors (it must have such predecessors since it must have indegree at least 1). Then follow one of the incoming edges of the predecessor to reach the predecessor of the predecessor. By following a chain of predecessors in this way, we must eventually exhaust the graph and reach a vertex with no predecessors. This is a vertex of indegree 0. This means that any vertex is reachable from the unique vertex of indegree 0. A similar argument shows that the vertex of outdegree 0 must be reachable from every vertex in the graph. This algorithm visits each vertex once to check its indegree and outdegree, and so the big-\(\theta\) bound is \(\theta(n)\).
5. **Proof or counter-example**: For each statement, prove or give a counter-example.

(a) If $G$ is a simple graph with $n$ vertices and $n$ edges, $n \geq 3$, then $G$ is connected.

**Solution**: This statement is false. A counter example is a graph with 5 vertices $v_1, v_2, v_3, v_4$ and $v_5$ with edges $(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4)$. The vertex $v_5$ is isolated from the rest of the graph and so the graph is disconnected, although it has 5 vertices and 5 edges.

(b) **Extra Credit**: If $G$ is a simple graph with $n$ vertices and $n$ edges, $n \geq 3$, then $G$ contains a cycle.

**Solution**: This statement is true. We prove by induction. The only graph on 3 vertices with 3 edges is a triangle, which contains a cycle. This is the base case. For the induction, we assume that any graph of $n$ vertices and $n$ edges contains a cycle. Now we consider a graph of $n + 1$ vertices and $n + 1$ edges. Pick a vertex $v_i$ of this graph of minimum degree. If $v_i$ has degree 0 or degree 1, then we consider the subgraph consisting of all the other vertices in the graph. This subgraph contains $n$ vertices and $n$ edges and so contains a cycle by the inductive hypothesis, and since a subgraph of $G$ contains a cycle, $G$ must contain a cycle. If $v_i$ has degree 2 or more, then every vertex has degree at least 2. Then $\sum_{v \in V} \deg(v) = 2n$ if every vertex has degree exactly 2. And the sum cannot be more than this because the fundamental formula relating the sum of degrees to the number of edges shows this equals twice the number of edges, which is exactly $2n$. So every edge has degree exactly 2. We can now show that this graph must contain a cycle. Let $v_h$ and $v_j$ be the two neighbors of $v_i$. If $v_h$ and $v_j$ have an edge between them already, then the graph contains the cycle $v_h \rightarrow v_i \rightarrow v_j \rightarrow v_h$. If they do not, then consider the graph $G$ formed by removing $v_i$ and replacing its two edges with an edge between $v_j$ and $v_k$. $G$ is a graph with $n$ vertices and $n$ edges and so contains a cycle by the inductive hypothesis. And the process of splitting the edge from $v_h$ to $v_k$ to create $G$ would not destroy this cycle, because it merely adds a new vertex inside any path using this edge. This means $G$ must also contain a cycle. Thus the induction is established and the theorem is proven.