1. **True of False (20 pt).**
Each correct answer is worth 2 points, but 2 points will be subtracted for each wrong answer, so answer only if you are reasonably confident. Use T for True, F for False, and D for Don't know in your answers.

F **a.** $n^{\cos n+\sin n}$ is in $\mathbf{O}(n)$.

T **b.** The array [23, 15, 16, 8, 7, 10, 11, 3, 5, 2] forms a max-heap.

T **c.** A bipartite graph cannot contain a cycle of length 3.

F **d.** Suppose we have a directed acyclic graph G with n vertices and at least $n-1$ edges. A topological sort produces the vertices $v_1, \ldots, v_n$ in that order. If we add the edge (v_n, v_1), then the resulting graph is guaranteed to be strongly connected.

F **e.** Given a graph G with only positive-weight edges and all edge weights are distinct, for vertices u and v, there must be a unique shortest path from u to v.

T **f.** If a tree does not have the Binary-Search-Tree property, then any rotations will result in a tree that also does not have this property.

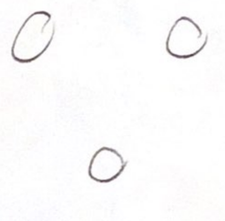
F **g.** If we have a min heap of 7 unique integers, then using a pre*order* traversal will never print the values of the heap in increasing order.

F **h.** The edge with the second smallest weight in a connected undirected graph with distinct-weight edges must be part of any minimum spanning tree of the graph. You can assume that there is at most one edge between any pair of vertices. → no dupl

F **i.** Given a directed weighted graph G, and two vertices u and v in G. The shortest path from u to v remains unchanged if we add 330 to all edge weights. You can assume that there is a unique shortest path from u to v before we add the weights.

T **j.** If an undirected graph with n vertices has k connected components, then it must have at least $n-k$ edges.

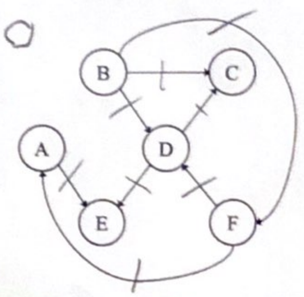
n vertices
G connected vertices



2. Short Answers (20 pt).
Show all steps for partial credits.

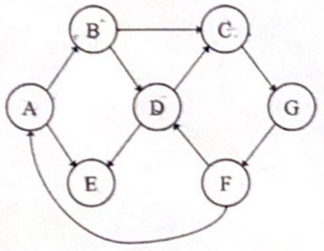
a. Write down the vertices in *topological order* for the following graph. **(3 pt)**

1) Loc vertex indegree 0
2) delete outgoing edges
3) repeat



[B, F, A, D, E, C]

b. Write down the vertices in the *Breadth First Search (BFS)* traversal order for the following graph, *starting from vertex A*. You can assume we explore neighbors of a node in alphabetical order. **(3 pt)**



[A, B, E, C, D, G, F]

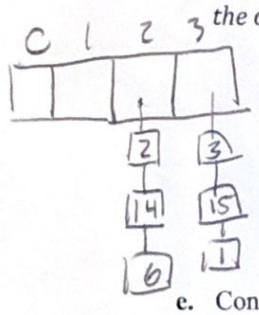
c. Write down the vertices in the *Depth First Search (DFS)* traversal order for the graph in part b, *starting from vertex D*. You can again assume we explore neighbors of a node in alphabetical order. **(3 pt)**

[D, C, G, F, A, B, E]

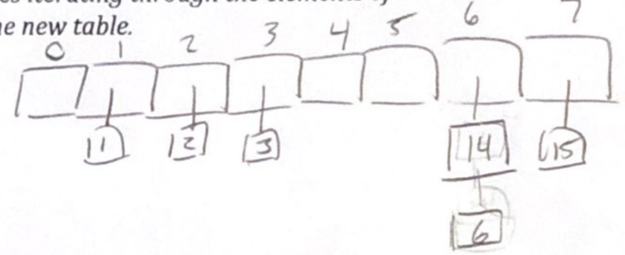
d. Suppose *H* below is a hash-table where collisions are handled by *open hashing* which uses chaining with linked lists. The default size of *H* is 4 and it automatically doubles its size whenever the load factor reaches 1.5. Draw the resulting hash-table after inserting the sequence 6, 14, 2, 1, 15, 3.

6 mod 8 = 6 1 mod 8 = 1
14 mod 8 = 6 15 mod 8 = 7
2 mod 8 = 2 3 mod 8 = 3

The hash code of an integer *x* are given by *x mod n*, where *n* is the size of *H*. You can assume that index starts at 0 for *H*. **(3 pt)**
Recall that resizing a hash table requires iterating through the elements of the old table and inserting them into the new table.



result = 7



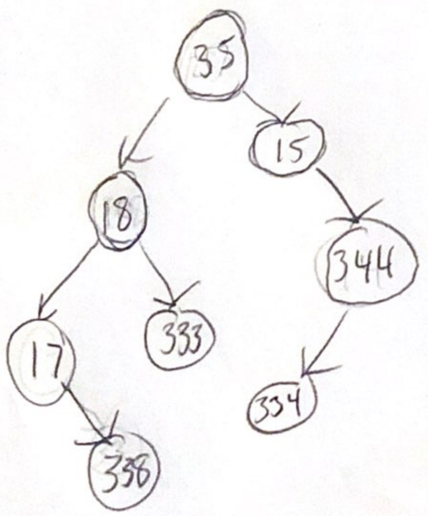
e. Consider the following pseudocode for performing an inorder traversal of a binary tree.

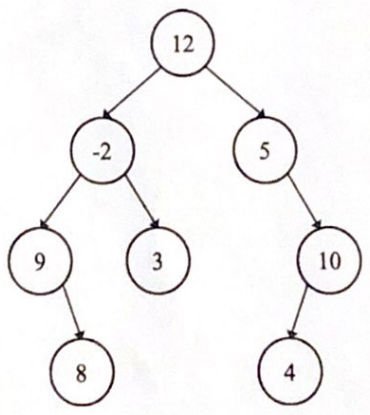
```
void inorder(Node root):
    if (root==NULL) return;
    inorder(root->left);
    visit(root);
    inorder(root->right);
}

void visit(Node n):
    if (n->left != NULL):
        n->val += n->left->val;
    else if (n->right != NULL):
        n->val += n->right->val;
    else:
        n->val += 330;
}
```

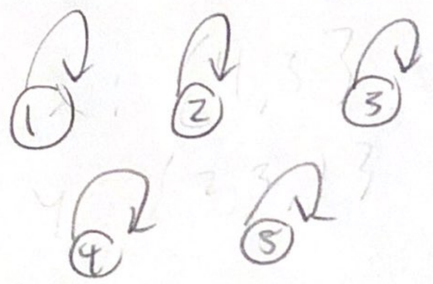
~ 9, 8, -2, 3, 12, 5, 4, 10

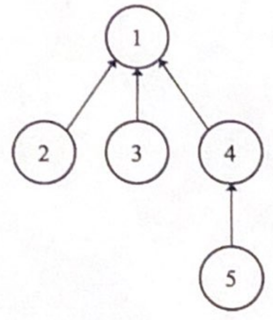
Draw the resulting tree from performing the above inorder traversal on the tree below. **(4 pt)**





f. Consider the disjoint set data structure. Give the sequence of `union(x, y)` calls (without path compression) that will produce the tree below. You can assume that if two trees with the same rank are unioned, then `x`'s root is put below `y`'s root. Each number starts in its own set at the beginning. **(4 pt)**





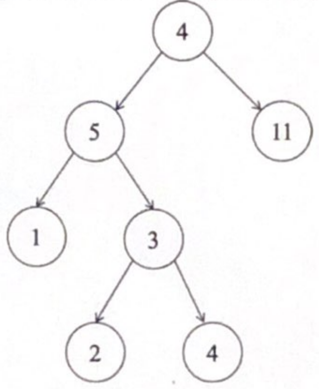
rank l < rank r,
l under right

union(2, 1)
union (3, 1)
union(5, 4)
union (1,4)

3. **Algorithm Design (30 pt).**

When you are asked to give your answer in pseudocode, you should write them clearly so that there is no ambiguity on what data structures you use and what your code does. You are strongly encouraged to write comment next to your pseudocode to explain the key step. Whenever there is a runtime requirement, you should justify the runtime of your algorithm with one or two sentences.

a. Given a *full* binary tree where each node (including the leaves) is associated with a *positive* integer value, complete the following method `twoLargestPathSums` (in pseudocode) to find the two largest path sums of all root-to-leaf paths. Note that the two largest path sums should correspond to two different root-to-leaf paths. You can assume that the tree has at least three nodes. In addition, you can assume that you can access the left child of a node `n` using `n→left`, the right child of `n` using `n→right`, and the value of at `n` using `n→val`. In the example below, the largest path sum is 16 (along the path 4 → 5 → 3 → 4) and the second largest path sum is 15 (along the path 4 → 11). Thus, your program should return (16, 15). **(6 pt)**



```
pair<int,int> twoLargestPathSums(Node root): {
    if (!root) return make_pair(root->val, root->val)
    pair<int,int> ans
    dfs(root, root->val, ans)
    return ans
}
void dfs(Node root, int pathsum, pair<int,int> ans){
    if (isLeaf) // node has no children
        potentially update ans.first & or ans.second
        if a top two pathsum is found
    if (root->left) dfs(root->left, pathsum + root->left->val, ans)
    if (root->right) dfs(root->right, pathsum + root->right->val, ans)
}
```

increment pathsum for every node in an individual root-leaf path

b. Given a binary search tree (BST), we want to delete the *largest* item in the BST. You can assume that you can access the left child of a node n using `n→left`, the right child of n using `n→right`, and the value of at n using `n→val`. Complete the method `deleteMax` below. The method should return the root of the BST after deletion. You can assume that `root` is not `NULL`. If deletion results in an empty tree, the method should ~~return NULL~~. **(6 pt)**

~~ ~~-1 subtree node ✓ can't have right subtree if its a max

```
Node deleteMax(Node root):
    Node curr = root
    while (curr->right)  curr = curr->right
    if (cur == root)  root = NULL, return root
    if (isLeaf)  curr = NULL, return root
    if (curr->left)  temp = in-order successor's ->val
        successor = NULL, curr->val = temp, return root
```

If the maximum has a left subtree, find the in-order traversal successor of that tree, store it's val as temp, delete this node, & finally, update max w/ value of successor.

c. Recall the celebrity problem from Homework 7. Suppose we know that there are exactly two celebrities in the group. They know each other but don't know anybody else in the group. Both of them are known by all non-celebrities. Given an adjacency matrix (M) representation of a graph with n nodes, where each node represents a person in the group and $M(i, j) = 1$ iff person i knows person j. Give an $\mathbf{O}(n)$ time algorithm (in pseudocode) to find the two celebrities. Your algorithm should return the ids of the two celebrities (between 0 and $n-1$). You can assume $n \geq 2$. **(6 pt)**

```
pair<int, int> findCelebrities(M):
    vector<int> candidates
    vector<int> rejected
    for u → v ∈ M
        candidates = candidates - {u}
        rejected.append(u)
        if !v ∉ R
            candidates = candidates + {v}
    return make_pair(remaining candidate 1, remaining candidate 2)
```

single pass
shortest path

d. Suppose you need to drive from city A to city B to deliver some medical supplies to city B. You are given a weighted directed graph G that models the connections between these two cities. The nodes in the graph are cities (including A and B) and there is an edge from node i to node j if there is a highway that allows us to drive from city i to city j. The weight on edge (i, j) represents the toll that we have to pay to take that highway (edge). You can assume that all the tolls are positive, which is typically the case. You are given a special pass that allows you to not pay toll on a single highway of your choice.

keep track of the largest weight edge in our minimized path, skip paying this toll

Describe an algorithm that uses Dijkstra's algorithm as a sub-procedure to find the cheapest route (in terms of the total tolls paid) from city A to city B. Your algorithm should have the same time complexity as Dijkstra's. **(6 pt)**

1) For each node that's an immediate neighbor of city A, sorted by increasing edge weights, we evaluate our neighboring nodes' neighboring nodes' edge weights, & sort for a new minimum total edge weight to be found!

2) we repeat this process per group of neighboring nodes until reaching city B, continually searching for the total minimum from A-B.
~ This can be applied even if directedness affects edge weight.

e. Consider the following turn-based game. You are given a list L of n integers (containing potentially negative numbers). In each turn, you can either (1) remove the first integer from the list and keep your current score or (2) add the first integer to your score and remove the first two integers. If there is only one integer remaining, then you can only choose option (1). You goal is to maximize your score which starts at 0.
For example, given the list [2, -1, 5, 7, 10], your optimal score is 2+7 = 9.
Give an **O**(n) time algorithm (in pseudocode) for this problem. You can assume that you can index into L in **O**(1) time. **(6 pt)**

```
int maxScore(L):
    size = L.size - 1
    for i = 0 : size
        if(L[i+1] > L[i])
            choose (1) // rm 1st integer, keep score
        else if(L[i+1] < L[i] || (i+2) > size)
            choose (2) // score += L[i], rm L[i] & L[i+1]
    return score
```