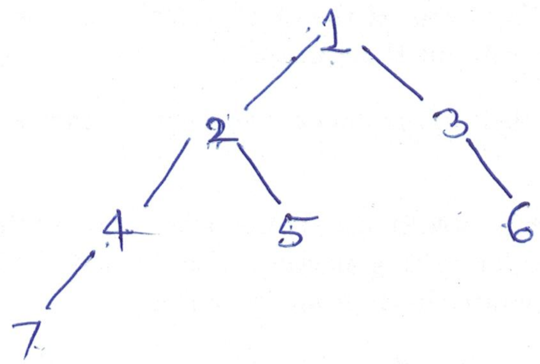
Name: ______________________ **Entry number**: ______________

COL106 Major Exam Duration: 2 hours

1. **Binary trees**

(a) (4 points) Let S be the set of binary trees (not necessarily search trees) over the set of keys $\{1, 2, 3, 4, 5, 6, 7\}$ that have in-order traversal $7, 4, 2, 5, 1, 3, 6$ and post-order traversal $7, 4, 5, 2, 6, 3, 1$. Answer the following questions.

1. What is the size of S?
Answer: 1
2. Draw any one binary tree that belongs to S.



(b) (5 points) Suppose you are given the pre-order and post-order traversals of a binary tree T over some set S of keys. Suppose $x, y \in S$. How will you determine whether x is an ancestor of y in T? Briefly explain why your answer is correct.

x is ancestor of y $\Leftrightarrow$ x appears before y in pre-order and x appears after y in post-order.
$\Rightarrow$ is obvious by def of pre and postorder
If y ancestor of x then y appears before x in preorder and y appears after x in post-order.
If y, x aren't ancestors/descendant of each other, let z be their lowest common ancestor. preorder(z) and postorder(z) visit x, y in the same order.

2. (5 points) **String Processing**
Recall the pattern matching problem discussed in class and in the fourth assignment. Let Σ be a finite alphabet and x be a string over Σ. If p is a non-empty string over Σ, then we say that the pattern p matches x at index i if $x[i \ldots (i+m-1)] = p$, where m is the length of p. If p_1 and p_2 are non-empty strings over Σ, we say that the pattern $p_1?p_2$ matches x at index i if $x[i \ldots (i+m_1-1)] = p_1$ and $x[(i+m_1+1) \ldots (i+m_1+m_2)] = p_2$, where m_1 and m_2 are the lengths of p_1 and p_2 respectively. In other words, '?' is a wildcard character which is not in Σ, and which matches every character in Σ.

Suppose you are given a text x and two patterns, p_1 and p_2, which don't contain the wildcard. You have already computed L_1 and L_2, which are sorted lists of indices at which p_1 and p_2 respectively match x. Write down an algorithm to compute L, the list of indices at which the pattern $p_1?p_2$ matches x. Your algorithm must run in time linear in the total size of L_1 and L_2.

Idea: Subtract m_1+1 from each element of L_2 to get L_3. Then find $L_1 \cap L_3$ using the "merge" step of merge-sort.

$L_3 \leftarrow \{ k-(m_1+1) \mid k \in L_2 \}$ (in sorted order)

$j_1 \leftarrow 0$, $j_2 \leftarrow 0$. $L \leftarrow$ empty list

while $j_1 < |L_1|$ and $j_2 < |L_3|$:
    if $L_1[j_1] = L_3[j_2]$
        Append $L_1[j_1]$ to L
        $j_1 \leftarrow j_1 + 1$, $j_2 \leftarrow j_2 + 1$
    else if $L_1[j_1] < L_3[j_2]$
        $j_1 \leftarrow j_1 + 1$
    else $j_2 \leftarrow j_2 + 1$

return L.

3. (4 points) **(2,4)-trees**
Recall the procedure for deleting a key from a $(2,4)$-tree: if we find the required key in a non-leaf node, then we exchange it with its successor and then delete it. But how do we find its successor in the first place? Your job is to write a `python` code to find the successor of a key in a $(2,4)$-tree that resides in a non-leaf node. A node in our $(2,4)$-tree has the following attributes.

   - `key`: a list of 3 objects. If the node contains k keys (where $1 \leq k \leq 3$), then these keys are `key[0]`, ..., `key[k-1]`, while `key[k]`, ..., `key[2]` are all `None`.
   - `child`: a list of 4 node-references. If the node has d children (where $0 \leq d \leq 4$), then the references of these children are `child[0]`, ..., `child[d-1]`, while `child[d]`, ..., `child[4]` all `None`.
   - `parent`: this is `None` if the node is the root of the tree, else the reference to the parent node.

   Complete the following function which takes a reference `p` of a non-leaf node and an integer `i` as input parameters, and returns a pair `(q,j)`, where `q` is a node-reference and `j` is an integer such that `q.key[j]` is the successor of `p.key[i]` in the $(2,4)$-tree. The function must run in time $O(\log n)$, where n is the number of keys in the $(2,4)$-tree.

```
def findSuccessor(p, i):
    q = p.child [i+1]
    while q.child[0] is not None:
        q = q.child [0]

    j = 0

    return (q, j)
```

4. **Undirected Graphs**
Recall that we defined the distance between two vertices of an undirected graph to be the length of a shortest path between those vertices (for example, the distance between a vertex and itself is 0, the distance between adjacent vertices is 1, and so on). The *diameter* of a graph is defined as the maximum, over all pairs u, v of its vertices, of the distance between u and v.

Let G be a connected undirected graph. Perform a breadth-first-traversal of G starting from some vertex s, and let T be the resulting breadth-first-traversal-tree. Suppose T has exactly $h+1$ levels $L_0 \ldots, L_h$, where $L_0 = \{s\}$.

(a) (2 points) Prove that the diameter of G is at least h.

A vertex u in L_h is at distance h from s

$\therefore \max\limits_{u,v \in V} \text{dist}(u,v) \geq \text{dist}(s,u) = h.$

(b) (4 points) Prove that the diameter of G is at most $2h$.

For any two vertices u, v,

$\text{dist}(u,v) \leq \text{dist}(s,u) + \text{dist}(s,v)$

Δ inequality

$\leq h + h = 2h$

A vertex x belongs to Li iff dist(s, x) = i.

Name: ______________________ **Entry number**: ______________

COL106 Major Exam Duration: 2 hours

5. **Directed Acyclic Graphs**
Recall the out-adjacency list representation of directed graphs, where we have one linked list for every vertex v, where the linked list contains the out-neighbors of v. Given a directed acyclic graph G in its out-adjacency list representation and a vertex t of G, we would like to compute, for every vertex v of G, the number of directed paths from v to t. Note that the number such paths could be exponential in the number of vertices, so a brute-force counting is too inefficient.

   (a) (3 points) Complete the following algorithm to compute an array `pathCount` indexed by the vertex set of G so that at the end of the run of the algorithm, `pathCount[v]` equals the number of directed paths from v to t in G. Each blank must be filled up with an expression that can be computed in $O(1)$ time.

   `findPathCounts(G,t)`

   1. Compute a topological sort of `G`.
   2. For each vertex `v` in the reverse of the topological sort order:
      2.1. If `v=t` then `pathCount[v]` $\leftarrow$ 1, else `pathCount[v]` $\leftarrow$ 0.
      2.2. For each out-neighbor `u` of `v`: `pathCount[v]` $\leftarrow$ path Count[v] + path Count[u]
   3. Return `pathCount`.

   (b) (4 points) Prove that the above algorithm runs in time $O(n+m)$, where n is the number of vertices and m is the number of edges in G.

Topological sort → $O(m+n)$ → proved in class.
Step 2.1 : once for every vertex ∴ $O(n)$
Step 2.2 : $O(d_v)$ for vertex v, where d_v is out-degree of v.
∴ Total time for 2.2. ∴ $O(\sum_v d_v) = O(m)$

(c) (4 points) Why is it necessary to iterate over the vertex set in reverse topological sort order in the algorithm of part (a)? Explain briefly.

We are using the fact: that

$$\#\, v \text{ to } t \text{ paths} = \sum_{u:\, \text{outnbr of } v} \#\, u \text{ to } t \text{ paths} \quad \text{if } v \neq t.$$

Reverse topological sort ensures that when we compute the LHS above, each term on the RHS is computed already.