MIDTERM

QUESTION 1. (Recurrences)
(a) Solve to Θ-order:

$$T_1(n) = 8T_1(n/2) + n^4.$$

Be sure to justify the application of any known result.
(b) Suppose

$$T_2(n) = 1 + T_2(n - \frac{n}{\ln n}).$$

Give an upper bound on $T_2(n)$. HINT: It is useful to know that $\ln(1+x) \le x$ when $|x| < 1$, Expanding the recurrence once,

$$\begin{aligned} T_2(n) &= 1 + T_2\left(n\left(1 - \frac{1}{\ln n}\right)\right) \\ &\le 2 + T_2\left(n\left(1 - \frac{1}{\ln n}\right)^2\right). \end{aligned}$$

If you repeat this expansion k times, what do you get? When do you stop expanding?

ANSWER
(a) et $f(n) = n^4$. The watershed function is $f_0(n) = n^3$. Hence $f(n) = \Omega(n^{3+\epsilon})$. This suggests that we have case (+) of the Master theorem. To verify this, we need to show $a \cdot f(n/b) \le c \cdot f(n)$ for some $c < 1$. Here $a = 8, b = 2$, and hence the choice of $c = 1/2$ will lead to an equality. We conclude by the Master Theorem that

$$T_1(n) = \Theta(n^4).$$

(b) e have

$$\begin{aligned} T(n) &= 1 + T\left(n\left(1 - \frac{1}{\log n}\right)\right) \\ &\le 2 + T\left(n\left(1 - \frac{1}{\log n}\right)^2\right), \qquad (why?) \\ &\vdots \\ &\le k + T\left(n\left(1 - \frac{1}{\log n}\right)^k\right), \end{aligned}$$

using monotonicity of $T(n)$. Hence $T(n) = k$ if we assume $T(n) = 0$ for $n \le 1$ and k is chosen so that

$$\left(1 - \frac{1}{\log n}\right)^{k+1} \le 1/n < \left(1 - \frac{1}{\log n}\right)^k.$$

Taking natural logs,

$$\begin{aligned} k \ln\left(1 - \frac{1}{\ln n}\right) &> -\ln n, \\ k\left(-\frac{1}{\ln n}\right) &> -\ln n, \qquad (\text{since } \ln(1+x) \le x \text{ for } |x| < 1), \\ k &< \ln^2 n. \end{aligned}$$

NOTE: If you had guessed $O(\ln^2 n)$, you could directly verify this by induction. One can also verify by induction that this is the lower bound.

QUESTION 2. (Red-Black Trees)
Draw a red-black tree T with black height 3 and specify a key k such that deleting k from T will decrease the black height of T. Draw the red-black tree after deleting k. HINT: when does the black height decrease in the deletion procedure? Ignore this hint if it is not helpful!

ANSWER

```
          B_4                                              B_4
         /   \                                            /   \
        /     \          Delete k = 3                    /     \
    B_2        B_6      ==============>              B_2        R_6
   /   \      /   \                                 /          /   \
B_1  B_3  B_5    B_7                             R_1        B_5    B_7
```

Notes: `B_i` denotes a black node with key i.
`R_i` denotes a red node with key i.

QUESTION 3. (Greedy Algorithm)
A vertex cover for a undirected graph $G = (V, E)$ is a subset C of the vertex set V such that for all edge e in E, at least one of its two vertices is contained in the set C. A *minimum* vertex cover is a vertex cover with the smallest size among all the vertex covers for the given graph. Below is a greedy algorithm that finds a vertex cover VC:

1. Initialize VC to the empty set.
2. Choose from the graph a vertex v with the largest out-degree.
   Add vertex v to the set VC, and remove vertex v and
   all edges that are incident on it from the graph.
3. Repeat step 2 until the graph has no more edges.
4. The final set VC is a vertex cover of the original graph.

Show a graph G, for which this greedy algorithm *fails* to give a minimum vertex cover.

ANSWER

```
                 * - *
                /
    * - * - *
                \
                 * - *
```

where "*" denotes a vertex, and "-" denotes an edge. Our algorithm will initially put into VC the unique vertex of degree 3. The final vertex cover has size 4. But the optimal vertex cover size is 3, obtained by choosing the three vertices of degree 2.

QUESTION 4. (Splay Trees)
Consider the following idea as an alternative for splaying: when splaying a key K, we always keep the current node in our search path at the root. (One advantage is that this becomes a "one-pass" algorithm as opposed to the original "two-pass" algorithm.) The following recursive code TopSplay is an attempt to implement this idea:

```
topSplay(Key K, Node N):
    case of N.Key:
    (1)  N.Key = K:
             return("found").
    (2)  N.Key < K:
             if u.Right = nil, return("pred");
             else u ← u.Right; rotate(u); topSplay(K, u).
    (3)  N.Key > K:
             if u.Left = nil, return("succ");
             else u ← u.Left; rotate(u); topSplay(K, u).
```

(a) Please indicate why this solution does not work.
(b) Propose a correct solution. Instead of the program code (as in (a)), we prefer that you provide a clear verbal description. Of course, you can supplement that with code if you prefer. HINT: the original `SplayStep` algorithm may give you an idea of what is needed.
(c) Does your solution have amortized cost of $\log n$, as in the original splay algorithm? Argue why or why not.

ANSWER
(a) onsider the binary search tree

```
        u(3)
       /
    v(1)
       \
        w(2)
```

If u is the root of this tree, $\texttt{topSplay}(2, u)$ will get into an infinite loop: we first rotate v and then recursively call $\texttt{topSplay}(2, v)$. This will rotate u and recursively call $\texttt{topSplay}(2, u)$, and so on. (b) e reimplement $\texttt{topSplay}K, u$. Let u_L, u_R be the left and right children of the root u. We have 4 possible states of our algorithm:

- State 0: Both u_L and u_R have not been visited.
- State 1: u_L but not u_R has been visited.
- State 2: u_R but not u_L has been visited.
- State 3: Both u_L and u_R have been visited.

Here is the transition rule for the states. In any state, if $u.\texttt{Key} = K$, we are done. Otherwise we take the following actions.

State 0: Initially, we are in state 0. If $u.\texttt{Key} > K$, then

$$u \leftarrow .\texttt{Left}; \texttt{rotate}(u); \mathit{state} \leftarrow 1.$$

If $u.\texttt{Key} < K$, we do the symmetrical thing and move into state 2.

State 1: If $u.\texttt{Key} < K$ then we next move into state 3 and perform the actions

$$v \leftarrow u.\texttt{Right}.\texttt{Left}; rotate(v); rotate(v); \texttt{topSplay}(K, v).$$

Otherwise, we remain in state 1 and perform the actions

$$v \leftarrow u.\texttt{Left}; \texttt{rotate}(v); \texttt{topSplay}(K, v).$$

State 2: This is symmetrical to State 1.

State 3: Once we are in state 3, we remain in state 3. If $u.\texttt{Key} > K$ then $v \leftarrow u.\texttt{Left}.\texttt{Right}$ else $v \leftarrow u.\texttt{Right}.\texttt{Left}$. In any case, perform the actions

$$rotate(v); rotate(v); \texttt{topSplay}(K, v).$$

An alternative description is to perform cases I, II or III in direct analogy to SplayStep.

(c) he above algorithm performs a collection of rotations and double rotations (or zig-zags). The rotations can in turn be decomposed into a sequence of zig-zig actions and a single rotation. Then the analysis of splay trees tells us that for each of the zig-zig and zig-zag actions, the credit-potential invariant is preserved. The single rotation can be paid for directly. Hence the logarithmic amortized cost of splaying is preserved.