CPSC 223 Spring 2025 Exam #2 with Solutions

Tuesday, April 8, 2025

Name: _____ **NetID:** _____

Problem 1 (9 points): Hashtables

a) Given an initially empty hashtable of capacity 4 using open addressing with linear probing and a resize threshold $\alpha = 0.5$, mark an X for the operation(s) that trigger a resize:

insert 10

insert 22

X insert 31

insert 4

insert 15

Explanation (a): The load factor is the number of entries divided by capacity. After inserting 10 and 22, load = $2/4 = 0.5$, which is at threshold but does not exceed it. Inserting 31 makes load = $3/4 = 0.75 > 0.5$, so the table is resized at that insertion.

b) Suppose we delete key 22 and then search for key 15. Mark an X for each statement that is guaranteed true:

Search finds key 15 without probing past its initial slot.

X Search may require probing past deleted slots.

X Search fails if a tombstone is not handled.

Explanation (b): Deletion leaves a tombstone at 22's slot. If 15 hashed to a later slot, the search must probe past the tombstone to find it. Without special handling of tombstones, the probe might stop early and wrongly report 'not found.'

c) What assumption about the hash function ensures average-case $O(1)$ performance?

Uniform hashing (each key equally likely to map to any slot, independently).

Explanation (c): Uniform hashing ensures that, on average, probes are equally likely to land anywhere, keeping the expected probe sequence short and thus guaranteeing constant-time performance in expectation.

Problem 2 (18 points): Binary Search Trees

For each code fragment below, write the number(s) that correctly build a BST containing keys $[5, 2, 8, 1, 3]$ when *inserted in that order*.

1.
```
Node *root = NULL;
insert(&root, 5);
insert(&root, 2);
insert(&root, 8);
insert(&root, 1);
insert(&root, 3);
```

Explanation (2.1): This uses the standard insert function in the given order, so it builds the correct BST.

2.
```
Node *root = create_node(5);
root->left = create_node(2);
root->left->left = create_node(1);
root->left->right = create_node(3);
root->right = create_node(8);
```

Explanation (2.2): Nodes are manually linked in the exact BST structure matching keys 5 as root, 2–1–3 on the left, and 8 on the right.

3.
```
Node *arr[5];
arr[0]=create_node(5);
arr[1]=create_node(2);
arr[2]=create_node(8);
arr[3]=create_node(1);
arr[4]=create_node(3);
```

Explanation (2.3): Nodes are created but never linked; no tree is formed.

4.
```
Node *root = create_node(5);
insert(&root, 8);
insert(&root, 2);
insert(&root, 1);
insert(&root, 3);
```

Explanation (2.4): Inserts in the wrong order: 8 before 2 will place 8 as the right child first, then inserting 2 will still go left, but the structure/order of insert operations differs from the required sequence.

5.

```
Node *root = create_node(5);
insert(&root, 2);
insert(&root, 8);
insert(&root, 3);
insert(&root, 1);
```

Explanation (2.5): Swapping the order of 1 and 3 changes the placement: 3 is inserted before 1, altering the subtree shape incorrectly.

Answer: 1 and 2. **Explanation (2 overall):** Only fragments (1) and (2) produce the correct BST when keys $[5, 2, 8, 1, 3]$ are inserted in that specified order. The others either never link properly or violate the insertion sequence.

Problem 3 (12 points): Graph Search and Shortest Paths

Consider the undirected graph G with adjacency lists:

$$A : B, C; \quad B : A, D, E; \quad C : A, F; \quad D : B; \quad E : B, F; \quad F : C, E.$$

a) BFS from A (alphabetical neighbors): $\boxed{A,\ B,\ C,\ D,\ E,\ F}$
Explanation (3.a): BFS visits nodes level by level. From A, neighbors B and C come first; then from B we enqueue D and E; from C we enqueue F.

b) DFS preorder from A: $\boxed{A,\ B,\ D,\ E,\ F,\ C}$
Explanation (3.b): Preorder visits the node, then recursively explores its neighbors in alphabetical order.

c) DFS postorder from A: $\boxed{D,\ C,\ F,\ E,\ B,\ A}$
Explanation (3.c): Postorder explores children first, then visits the node after all deeper calls return.

d) Dijkstra distances from A (unweighted, weight=1):

$$\text{dist}(A) = 0,\ \text{dist}(B) = 1,\ \text{dist}(C) = 1,\ \text{dist}(D) = 2,\ \text{dist}(E) = 2,\ \text{dist}(F) = 2$$

Explanation (3.d): All edges cost 1, so the distance is simply the minimum number of hops from A.

e) Cycle? $\boxed{\text{Yes}}$
Explanation (3.e): A–B–E–F–C–A forms a cycle.

f) Connected? $\boxed{\text{Yes}}$
Explanation (3.f): Every vertex is reachable from A.

Problem 4 (16 points): k-d Tree Nearest Neighbor Search

Complete the code below. For each /∗ n ∗/, write the missing recursive call.

```
// in-order: axis = depth % 2
void nearest(kdnode *root, double target[2],
             int depth, kdnode **best, double *bestDist) {
    if (root == NULL) return;
    int axis = depth % 2;
    double d = (root->point[0]-target[0])*(root->point[0]-target[0])
             + (root->point[1]-target[1])*(root->point[1]-target[1]);
    if (d < *bestDist) {
        *bestDist = d;
        *best    = root;
    }
    if (target[axis] < root->point[axis]) {
        /* 1 */ nearest(root->left, target, depth+1, best, bestDist);
    } else {
        /* 2 */ nearest(root->right, target, depth+1, best, bestDist);
    }
    double diff = target[axis] - root->point[axis];
    if (diff*diff < *bestDist) {
        /* 3 */ nearest(
                  target[axis] < root->point[axis]
                    ? root->right : root->left,
                  target, depth+1, best, bestDist
                );
    }
}
```

#1 `nearest(root->left,  target, depth+1, best, bestDist);`

Explanation (4.1): Recurse into the subtree on the same side as the target.

#2 `nearest(root->right, target, depth+1, best, bestDist);`

Explanation (4.2): If the target coordinate is greater, we explore the right subtree first.

#3
```
nearest(
  target[axis] < root->point[axis]
    ? root->right : root->left,
  target, depth+1, best, bestDist
```

```
);
```

Explanation (4.3): If the hypersphere around the current best crosses the splitting plane, we must explore the opposite branch as well.

Problem 5 (6 points): Heap Runtimes

Implementations:

- A: Binary heap
- B: Fibonacci heap
- C: Unsorted array

For each operation, write the letter (once each):

1. $\Theta(1)$ *amortized* insert: $\boxed{C}$

   Explanation (5.1): Appending to an unsorted array takes $O(1)$.

2. $\Theta(1)$ *worst-case* find-min: $\boxed{B}$

   Explanation (5.2): Fibonacci heaps maintain a pointer to the minimum, so it's $O(1)$.

3. $\Theta(\log n)$ *worst-case* extract-min: $\boxed{A}$

   Explanation (5.3): Binary heap extract-min percolates down in $O(\log n)$.

Problem 6 (8 points): Depth-First Search Code Completion

```
void dfs(Graph *g, int v) {
    /* 1 */ visited[v] = true;            // mark v
    printf("%d ", v);
    for (int u = 0; u < g->n; u++) {
        if (g->adj[v][u] && !visited[u]) { // edge exists & not visited
            /* 3 */ dfs(g, u);
        }
    }
}
```

#1 `visited[v] = true;`

Explanation (6.1): We mark the current node as visited to avoid cycles.

#2 In the `if` clause: `g->adj[v][u] && !visited[u]`

Explanation (6.2): We only recurse if there's an edge and the neighbor is unvisited.

#3 `dfs(g, u);`

Explanation (6.3): Recurse on the neighbor u.

Problem 7 (15 points): AVL Tree Traversals

Insert and then give preorder and inorder of the resulting balanced tree.

a) Insert $\{30, 20, 10\}$:
Preorder: $\boxed{20,\ 10,\ 30}$
Inorder: $\boxed{10,\ 20,\ 30}$
Explanation (7.a): A right–right imbalance at 30–20–10 triggers a single right rotation at 20.

b) Insert $\{10, 20, 30\}$:
Preorder: $\boxed{20,\ 10,\ 30}$
Inorder: $\boxed{10,\ 20,\ 30}$
Explanation (7.b): A left–left imbalance at 10–20–30 triggers a single left rotation at 20.

c) Insert $\{20, 10, 30, 25, 27\}$:
Preorder: $\boxed{20,\ 10,\ 27,\ 25,\ 30}$
Inorder: $\boxed{10,\ 20,\ 25,\ 27,\ 30}$
Explanation (7.c): Insertion of 27 under the right subtree of 20 creates a left–right case at 30; fix with right rotation at 30's left child (25), then left rotation at 20.

Problem 8 (16 points): Dijkstra's Algorithm Code Completion

```
void dijkstra(Graph *g, int src) {
    int dist[MAXV];
    bool known[MAXV] = {false};
    /* 1 */ for (int v = 0; v < g->n; v++) dist[v] = INT_MAX;
    dist[src] = 0;
    for (int i = 0; i < g->n; i++) {
        /* 2 */ int v = minVertex(dist, known, g->n); // find unknown min
        known[v] = true;
        for (Edge *e = g->adj[v]; e != NULL; e = e->next) {
            int w = e->to;
            int wgt = e->weight;
            /* 3 */ if (!known[w] && dist[v] + wgt < dist[w]) {
                /* 4 */ dist[w] = dist[v] + wgt;
            }
        }
    }
}
```

#1 `for(int v=0; v<g->n; v++) dist[v]=INT_MAX;`

Explanation (8.1): Initialize all distances to "infinite."

#2 `int v = minVertex(dist, known, g->n);`

Explanation (8.2): Select the nearest unknown vertex.

#3 `!known[w] && dist[v] + wgt < dist[w]`

Explanation (8.3): Check if going through v gives a shorter path to w.

#4 `dist[w] = dist[v] + wgt;`

Explanation (8.4): Relax the edge by updating w's distance.