11.2-2

0	→ 28 → 19 → 10 /
1	/
2	/
3	→ 12 /
4	/
5	→ 5 /
6	→ 15 → 20 → 33 /
7	/
8	→ 17 /

11.2-4 (optional)

Under the assumption of simple uniform hashing, the running time of each operation is:

1) Unsuccessful search: $\theta(1+\alpha/2)$
2) Successful search: $\theta(1+\alpha/2)$
3) Insertion: $\theta(1+\alpha/2)$
4) Deletion: $\theta(1+\alpha/2)$

Proofs:

1)

- By assuming simple uniform hashing, any key *k* is equally likely to hash to any of the *m* lists.
- Assume that each list is sorted in ascending order. The search is unsuccessful if we compare *k* with the key of each element on a list until one key is greater than *k*. On average, this takes $\theta(\alpha/2)$.
- It takes $\theta(1)$ to compute *h(k)*.
- The total running time is the time to compute *h(k)* and to search the list.

2)

- Similar to 1), we compare *k* with each element in a list until both are equal. This also takes $\theta(\alpha/2)$ on average.
- It takes $\theta(1)$ to compute *h(k)*.
- The total running time is the time to compute *h(k)* and to search the list.

3)

- To insert *k*, compute *h(k)* and then find the correct position to insert into the chosen list.
- It takes $\theta(1)$ to compute *h(k)*.
- Similar to 1), it takes $\theta(\alpha/2)$ on average to find the inserting position.
- The total running time is the time to compute *h(k)* and to search the list.

4)

- It takes $\theta(1)$ to compute *h(k)*.
- We need to search for the key value before deletions, which takes $\theta(1+\alpha/2)$ for both successful and unsuccessful cases (see proofs 1 & 2)
- If found, there is a constant *c* time to update the linked list
- Total running time: $\theta(1+\alpha/2+c)= \theta(1+\alpha/2)$

Note: the average $\theta(\alpha/2)$ is obtained by finding the expected number of elements examined.
Let P(X = the *i*th elements) = *m/n*. Then the expected number of elements examined is:

$$E(X) = \sum x \cdot P(X = x) = \sum_{i=i}^{n/m} i \cdot (\tfrac{m}{n}) = \tfrac{1}{2}\alpha + \tfrac{1}{2}$$

11.4-1

Linear probing: $h(k) = h'(k)+i \bmod 11$
$i = 1,2,3,\ldots,10,0$

0	1	2	3	4	5	6	7	8	9	10
22	88	/	/	4	15	28	17	59	31	10

Quadratic probing: $h(k) = h'(k) + i + 3i^2 \bmod 11$
$i + 3i^2 = 0,4,3,8,8,3,4,0,2,10,2,0,4,3$

0	1	2	3	4	5	6	7	8	9	10
22	/	88	17	4	/	28	59	15	31	10

Double hashing: $h(k) = h'(k) + i\, h_2(k) \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
22	/	59	17	4	15	28	88	/	31	10

11.3-3

Let y be a concatenation of characters $k_0 \circ k_1 \circ k_2 \circ \cdots \circ k_n$, and $x = k_{i0} \circ k_{i1} \circ k_{i2} \circ \cdots \circ k_{in}$ be a permutation of y. Then the corresponding radix-2^p representations of x and y are:

$$\begin{aligned} y &= k_0 \times 2^{pn} + k_1 \times 2^{p(n-1)} + k_2 \times 2^{p(n-2)} + \cdots + k_n \times 2^{p(0)} \\ x &= k_{i0} \times 2^{pn} + k_{i1} \times 2^{p(n-1)} + k_{i2} \times 2^{p(n-2)} + \cdots + k_{in} \times 2^{p(0)} \end{aligned}, \text{ where } 0 \le k_i < 2^p .$$

$$\begin{aligned} h(y) &= y \bmod m \\ &= k_0 \times 2^{pn} + k_1 \times 2^{p(n-1)} + k_2 \times 2^{p(n-2)} + \cdots + k_n \times 2^{p(0)} \bmod m \\ &= [k_0 \times 2^{pn}]_m + [k_1 \times 2^{p(n-1)}]_m + \cdots + [k_n \times 2^{p(0)}]_m \\ &= [k_0]_m \cdot [2^{pn}]_m + [k_1]_m \cdot [2^{p(n-1)}]_m + \cdots + [k_n]_m \cdot [2^{p(0)}]_m \\ &= [k_0]_m + [k_1]_m + \cdots + [k_n]_m \\ &= [k_0 + k_1 + \cdots + k_n]_m = [k_{i0} + k_{i1} + \cdots + k_{in}]_m \\ &= [k_{i0} \times 2^{pn} + k_{i1} \times 2^{p(n-1)} + \cdots + k_{in} \times 2^{p(0)}]_m \\ &= x \bmod m = h(x) \end{aligned}$$

■

22.1-1

Assume: a directed graph; an adjacency-list representation

a) Algorithm for out-degree:

```
1   For each u in V do
2      out-deg[u] ← 0
3   For each u in V do
4      for each v in Adj[u] do
5         out-deg[u] ← out-deg[u] + 1 // count # of edges from u
```

Time for steps 1 and 2 take O(|V|) since there are |V| vertices.

Steps 3 to 5 take O(max(|V|, |E|)) = O(|V| + |E|) time. The reason is that we have to scan through |V| elements in array Adj, even if all of them point to NULL. On the other hand, the sum of the lengths of all adjacency lists is |E|. We examined each element in adjacency lists once.

∴ O(|E| + |V|) + O(|V|) = O(|E| + |V|)

b) Algorithm for in-degree:

```
1   For each u in V do
2      in-deg[u] ← 0
3   For each u in V do
4      for each v in Adj[u] do
5         in-deg[v] ← in-deg[v] + 1 // count # of edges to v
```

Steps 1 and 2 take O(|V|).
Steps 3 to 5 take O(max(|V|, |E|)) = O(|V| + |E|) time, since we have to scan all the adjacency lists (even if they may be all empty), and there are |V| lists. Moreover, the sum of the lengths of all adjacency lists is O(|E|). Each element in adjacency lists swill be examined once.

∴ O(|E| + |V|) + O(|V|) = O(|E| + |V|)

22.1-3

a) Adjacency-list algorithm:

```
1   For each u in V do
2      AdjT[u] ← NULL
3   For each u in V do
4      for each v in Adj[u] do
5         Insert(adjT[u], v) // u is now being pointed by v
```

Step 1 to 2 take O(|V|).
Step 3 to 5 take O(max(|V|, |E|)) = O(|V| + |E|).
Reason: since all adjacency lists must be scanned (even if they may all be empty), and there are |V| lists. However, the sum of lengths of all adjacency lists is |E|. Each element in adjacency list will be examined once.

∴ O(|V|) + O(|E| + |V|) = O(|E| + |V|)

b) Adjacency-matrix algorithm (base index = 1):

```
1   For i = 1 to |V| - 1 do
2      For j = i + 1 to |V| do
3         aT_ji = a_ij
4         aT_ij = a_ji
5   For i = 1 to |V| do
6      aT_ii = a_ii
```

All entries in adjacency-matrix A will be scanned.

$$\sum_{i=1}^{|V|-1}\sum_{i+1}^{|V|}1$$
$$= \sum_{i=1}^{|V|-1}(|V| - (i+1) + 1) = (|V|-1)(|V|+1) - \sum_{i=1}^{|V|-1}(i+1)$$
$$= |V|^2 + 1 - [(|V|-1)(|V|)/2 - (|V|-1)]$$
$$= \tfrac{1}{2}|V|^2 + \tfrac{3}{2}|V| \in O(\tfrac{1}{2}|V|^2) = O(|V|^2)$$

(i.e. using half of the size of |V|×|V| matrix A to access all entries in matrix A)

Wing Kuen Lee <cswkl@ust.hk>

22.3-10

Case 1:
There is one and only one vertex u in a directed graph G.

Case 2:
The vertex u has only one outgoing edge, which is (u, u), i.e. a self-loop.

22.4-5 (Optional)

a) TOPOLOGICAL SORTING algorithm.
Assume the graph is represented by an adjacency-lists.
Let R be a queue. Let L be a linked-list

```
1     For each u in V do
2        Indeg[u] ← 0
3     For each u in V do
4        For each v in adj[u] do
5           Indeg[v] ← indeg[v] + 1
```

Construct the in-degree table; takes O(|V| + |E|) time

```
6     For each u in V do
7        If indeg[u] = 0 then
8           ENQUEUE(R,u)
9           LIST-INSERT(L,u)
```

Search all vertices with in-degree = 0 and record vertices in the queue.

ENQUEUE, DEQUEUE, and LIST-INSERT all take O(1). Steps 6 – 9 take O(|V|) since there are |V| vertices, each has 1 ENQUEUE + 1 DEQUEUE operations.

```
10    While R ≠ NULL do
11       u ← DEQUEUE(R)
12       For each v in adj[u] do
13          Indeg[v] ← indeg[v] - 1
14          If indeg[v] = 0 then
15             ENQUEUE(R,v)
16             LIST-INSERT(L,v)
17    Return L
```

R queue keeps the vertices that to be removed. When a vertex is removed and causes its "neighbour" in-degree = 0, removes this "neighbour" as well.

1) In total, |V| vertices will be ENQUEUE once in R. Each of these vertices will be DEQUEUE once from R;
∴ |V|(O(1)+O(1)) = O(|V|); but
2) All elements in adjacency lists will be examined once, and the sum of length of adjacency lists is |E|. Thus, Steps 10-16 take O(max(|V|, |E|)) = O(|V| + |E|) as we have to go through all vertices, as well as elements in adjacency lists.

Note: can output each vertex with indegree=0 instead of putting into a linked-list. But their effects are same

Total time:
O(|V|+|E|) + O(|V|) + O(|V|+|E|) = O(|V|+|E|)

b) When the linked-list L (i.e. sorted result) is returned, no vertex in the cycles parts will be in the queue L. Only vertices that are not in cycles will be in L. So if none of the vertices in G has indegree = 0, then L will be empty when returned.