1. In the three given schemes, I would choose the second one, that is, use a mutex over each hash bin. The reason is that the first one would only allow one thread to access the hash table at any given time, thus decreasing the performance dramatically. The second one would allow multiple threads to access the hash table at the same time, but if there are threads want to modify a hash bin at the same time, the mutex will guarantee these modifications executed in a well ordered way. The third one is like the second one, but would be harder to implement. Thus, I would choose the second one.

2. The given implementation is not correct. There are several reasons, first, if $q2$ is empty, it fails to return $q1$ the same as befofe the swap was attempted. Second, if $q1$ or $q2$ is empty, the lock of them will be grabbed but not released, resulting in the following accesses to these two queues impossible. Third, a deadlock would happen, if atomic_swap(q1,q2) and atomic_swap(q2,q1) are called at almost the same time.A possible modification of the given implementation could be like below.

```
1  extern Item* dequeue(Queue*);
2  // pops an item from a stack
3  extern void enqueue(Queue*, Item*);
4  // pushes an item onto a stack
5  void atomic_swap(Queue* q1, Queue* q2)
6  {
7          Item* item1;
8          Item* item2; // items being transferred
9          sort the two argumeents in some way;
10         // so when atomic_swap(q1, q2) and
11         // atomic_swap(q2, q1) are called
12         // at the same time, there would not be a
13         //deadlock
14         wait(q1->lock);
15         wait(q2->lock);
16         item1 = pop(q1);
17         if (item1 != NULL)
18         {
19                 item2 = pop(q2);
20                 if (item2 != NULL)
21                 {
22                         push(q2, item1);
23                         push(q1, item2);
```

```
                    }
                    else
                    {
                            push(q1, item1);
                    }
            }
            singal(q2->lock);
            singal(q1->lock);
}
```

3. (a) 'smokers' is used to indicate the number of smokers who are smoking now and 'nonsmokers' is used to indicate the number of nonsmokers in the lounge.

```
int smokers = 0, nonsmokers = 0;
```

(b) 'enter' is used to guarantee the executing order is the same as the arriving order of people. 'mutexSmoker' is used to protect the 'smokers' and 'mutexNonsmoker' is used to protect the 'nonsmokers'.

```
Semaphore mutexSmoker = mutexNonsmoker = 1;
Semaphore enter = smoking = 1;
```

(c) One possible solution is like below.

```
enterlounge(true)
{
        wait(enter);
        signal(enter);
}

enterlounge(false)
{
        wait(enter);
        wait(mutexNonSmoker);
        if (nonsmokers == 0)
                wait(smoking);
        nonSmokers++;
        signal(mutexNonSmoker);
        signal(enter);
}

```

```
18 smoke()
19 {
20         wait(mutexSmoker)
21         if (smokers == 0)
22                 wait(smoking);
23         smokers++;
24         signal(mutexSmoker);
25         // smoking here;
26         wait(mutexSmoker);
27         smokers--;
28         If(smokers == 0)
29                 signal(smoking);
30         signal(mutexSmoker);
31 }
32
33 leaveLounge(true)
34 {
35  //just leave
36 }
37
38 leaveLounge(false)
39 {
40         wait(mutexNonSmoker);
41         nonSmokers--;
42         if (nonSmokers == 0)
43                 signal(smoking);
44         signal(mutexNonSmoker);
45 }
```

(d)

- Mutual exclusion. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
- Progress. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- Bounded waiting. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
- Sequential Access. The sequence of accessing the critical sec-

tion follows the order of the requests raised by the processes The ‘mutexSmoker’, ‘mutexNonsmoker’ and ‘smoking’ semaphores guarantee the first three properties, and the ‘enter’ semaphore guarantees the last propertie.