Q1.

(a) Because isempty() does not affect the top variable in the above method, there will be no race condition. Push(item) and pop(), on the other hand, update the variable uptop. Hence, they have a race condition. As a result, the data "top" will be exposed to the race condition.

(b) A mutex lock or a semaphore can be used to solve the race condition. Semaphore was used to solve the problem: Set the global semaphore s = 1... When a process is in the crucial section, it becomes 0. When it exits the critical section, it returns to 1. The wait and signal functions are used as follows:

```
wait(S){

while(s<=0); // busy waiting

s--;

}

signal(s){

s++;

}
```

The Modified Code:

```
s = 1

push(item)

{

wait(s);

if (top < SIZE) {

stack[top] = item;

top++;

} else ERROR

signal(s);

}

pop()

{

wait(s);

if (!is empty()) {

top--;

return stack[top];
```

```
} else ERROR

signal(s);

}

// No wait and signal required here

is empty() {

if (top == 0)

return true;

else

return false;

}
```

Q2(a)
In the readers-writers dilemma, throughput is improved by prioritising numerous readers over permitting a single writer to have exclusive access to the shared values. On the other side, favouring readers may cause authors to go starving. By preserving timestamps connected with waiting processes, the readers/writers dilemma might be avoided. When a writer completes a task, it will reactivate the process that has been idle for the longest time. If a reader comes and observes another reader using the database, it will only go to the important portion if there are no waiting writers. Fairness would be ensured by these constraints.

The writer(s) may become hungry if the readers are always reading from the resource in the readers-writers issue. They'll tie up the semaphore, and the writer will never be able to write again. This is unjust to the writer, but it is ideal for reader throughput because any number of people can read at the same time without causing any problems. It should form a queue when they need to read and wait for only the now reading readers to finish and the semaphore to be accessible to be fairer to the writer(s). Any additional readers that want to read from the semaphore join the queue behind the write. The writer has a turn with the semaphore when the readers have finished. After that, any and all of the eager readers can get to work.

Q3)
Using a semaphore to fix the race condition — We can use a semaphores to replace the variable available resources. We can increase the number of accessible resources by using signal() and decrease the number of available resources by using wait ().

```
wait(s)
{
   while s <= 0
s -= count;
}
signal(s) {
s += count;
}
```