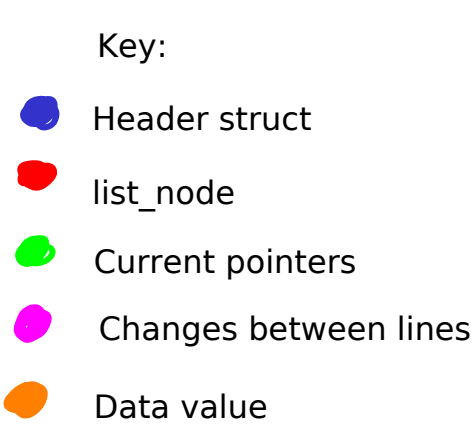


Queue and Stack Manipulation

By: Caroline Buckey, Spring 2012. Revised: Alex Cappiello, Fall 2012

September 28, 2012

Suppose you have queues and stacks implemented using linked lists as shown in lecture, with integer data. Specifically, you have the following structs:

```
struct list_node {
    int data;
    struct list_node *next;
};
typedef struct list_node list;

struct queue_header {
    list *front;
    list *back;
};
typedef struct queue_header* queue;

struct stack_header {
    list *top;
    list *bottom;
};
typedef struct stack_header* stack;
```

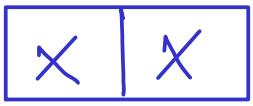
Recall from lecture that in both stacks and queues, we always keep one dummy node. In queues, `back` points to this dummy node, and in stacks, `bottom` points to this dummy node. Its fields are uninitialized, and it simply ensures that we never need to worry about `front`, `back`, `top`, or `bottom` being null.

Problem 0 (Warmup)

Recall the code for `queue_new()`:

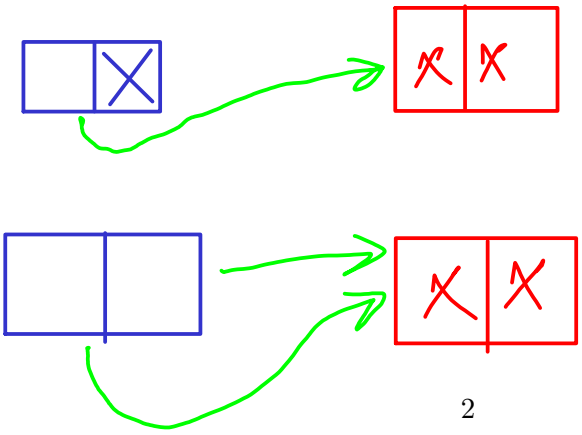
```
queue queue_new()
//@ensures is_queue(\result);
//@ensures queue_empty(\result);
{
  1: queue Q = alloc(struct queue_header);
  2: list* p = alloc(struct list_node);
  3: Q->front = p;
  4: Q->back = p;
  5: return Q;
}
```

Draw the state of the queue after each given line of code. Use `X` for uninitialized fields.

1. 

2. 

3.

4. 

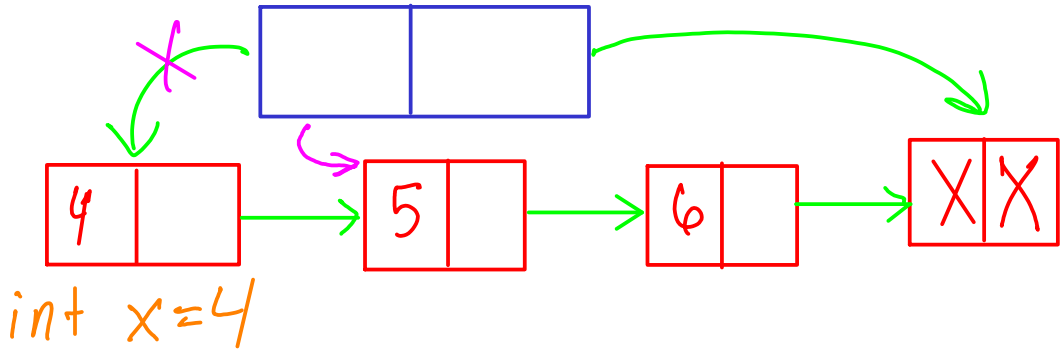
Problem 1

Recall the code for `deq`:

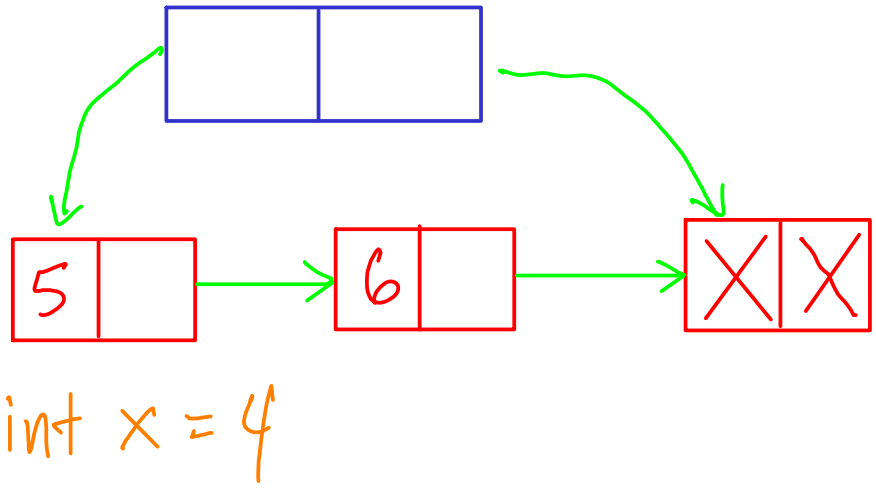
```
int deq(queue Q)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
    1: int x = Q->front->data;
    2: Q->front = Q->front->next;
    3: return x;
}
```

Suppose `deq(Q)` is called on a queue `Q` that contains before the call, from front to back, `(4, 5, 6)`. Draw the state of the queue after each given line of code. Include an indication of what data the variable `x` holds.

1.



2.



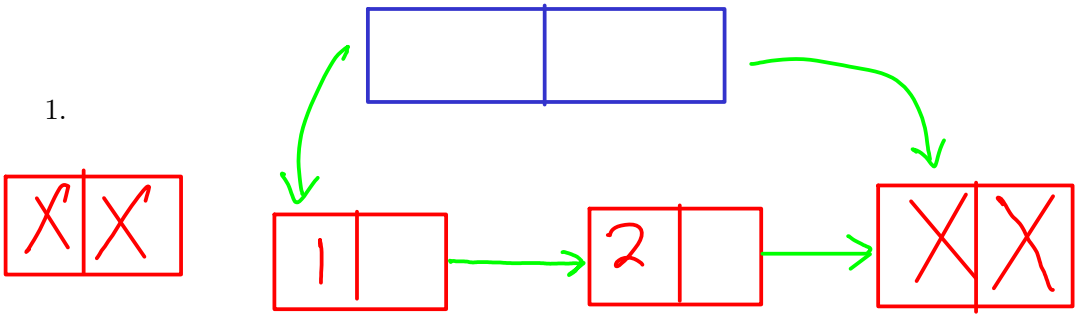
Problem 2

Recall the code for `push`:

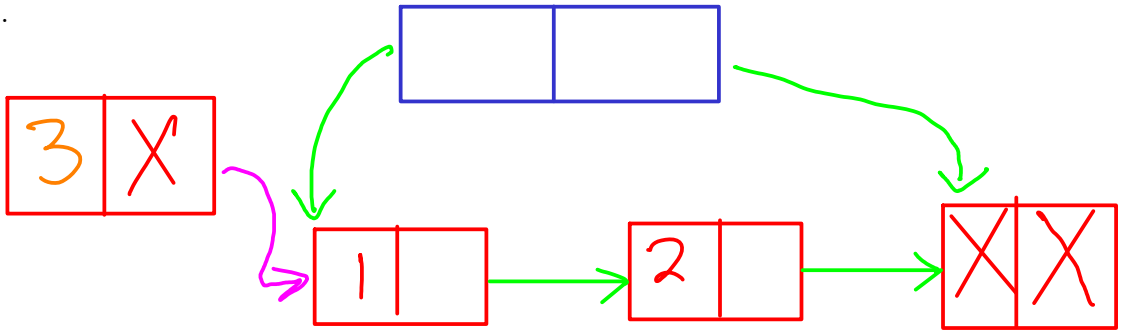
```
void push(stack S, int x)
//@requires is_stack(S);
//@ensures is_stack(S);
{
    1: list *l = alloc(struct list_node);
    2: l->data = x;
    3: l->next = S->top;
    4: S->top = l;
}
```

Suppose `push(S, 3)` is called on a stack `S` that contains before the call, from top to bottom, `(1, 2)`. On the next page, draw the state of the stack after each given line of code. Include the list struct separately before it has been added to the stack.

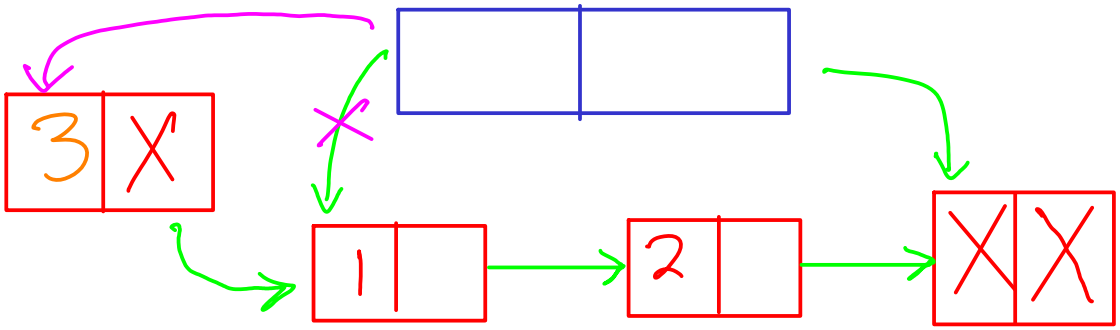
1.



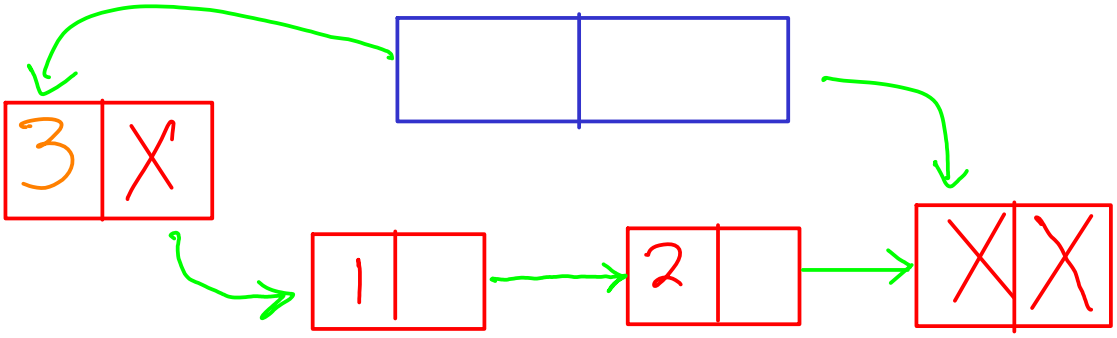
2.



3.



4.



Problem 3

Suppose we want to modify our queue data structure to support the operation `cheat()`. We allow someone to cheat by cutting in line to a position directly after their friend if they know someone in the line. They may only do this if they know a pointer to their friend's node, not just their friend's index in line.

Problem 3a

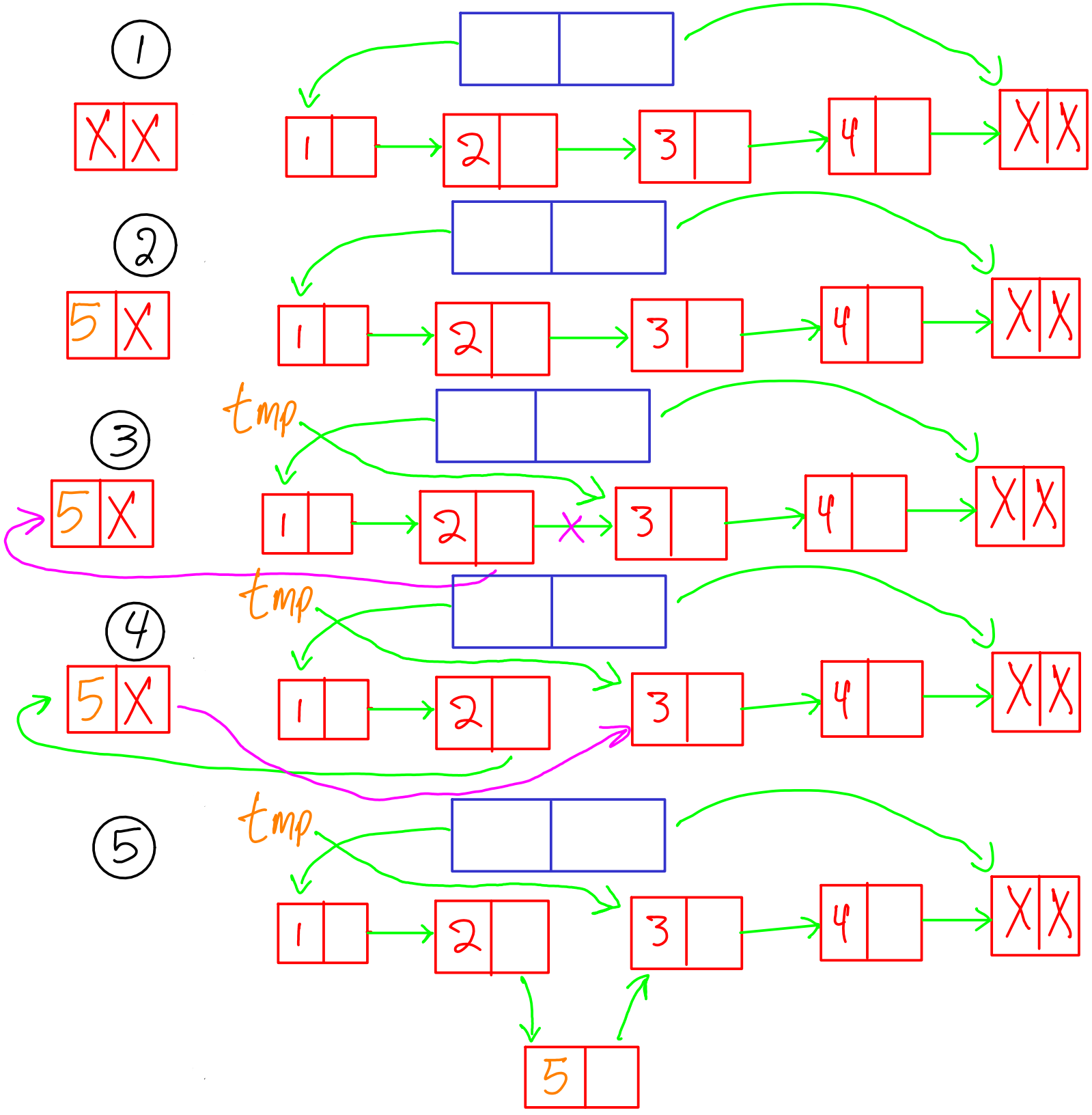
Write a function `void cheat (queue Q, list *friend, int cheater)` that creates a node for the cheater and adds them to the line directly after their friend. It is illegal to call this function if `friend` is not a node of `Q`.
Hint: Solve this problem and Problem 3b in parallel

Problem 3b

Consider a queue `Q` which contains, from front to back, `(1, 2, 3, 4)`, a cheater `5`, and a friend which is a pointer to the node containing `2`. Draw the state of the queue after each line of your `cheat()` function.

Note: There is intentionally more space given than necessary.

```
void cheat (queue Q, list *friend, int cheater)  //@requires friend != NULL;
//@requires is_queue(Q) && is_segment(Q->front,friend) && is_segment(Q->back,friend
//@ensures  is_queue(Q) && friend->next->data == cheater;
{
        list *cheat = alloc(list);
        cheat->data = cheater;
        list *tmp = friend->next;
        friend->next = cheat;
        cheat->next = tmp;









}
```


1
X X
1
2
3
4
X X
2
5 X
1
2
3
4
X X
tmp
3
5 X
1
2
X
3
4
X X
tmp
4
5 X
1
2
3
4
X X
tmp
5
1
2
3
4
X X
5