15-122: Principles of Imperative Computation

Recitation 10 — Josh Zimmerman

Practice!

Credit for this section goes to CMU alumna Caroline Buckey. It was updated by current 122 TA Alex Cappiello to account for differences in exactly how we teach certain material.

Suppose you have queues implemented using linked lists as shown in lecture. 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;
```

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

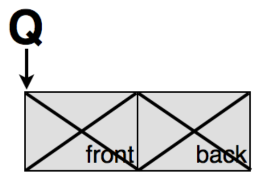
Visualizing `queue_new()`

Here's the code for `queue_new()`. For each of lines 5-8 (inclusive) draw a diagram that shows the state of the queue after that line executes. Use `X` for uninitialized fields.

```
 1 queue queue_new()
 2 //@ensures is_queue(\result);
 3 //@ensures queue_empty(\result);
 4 {
 5   queue Q = alloc(struct queue_header);
 6   list* p = alloc(struct list_node);
 7   Q->front = p;
 8   Q->back = p;
 9   return Q;
10 }
```

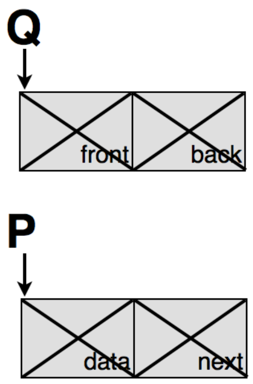
5.

Solution:



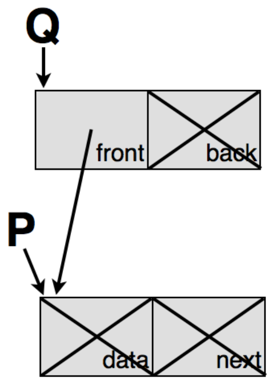
6.

Solution:



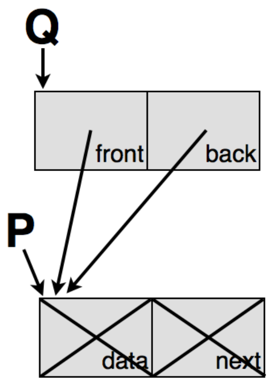
7.

Solution:



8.

Solution:



Visualizing deq

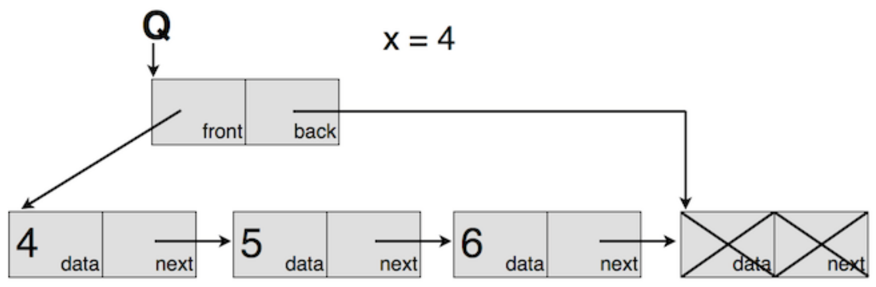
This is the code for deq:

```
int deq(queue Q)
//@requires is_queue(Q);
//@ensures is_queue(Q);
{
   int x = Q->front->data;
   Q->front = Q->front->next;
   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 lines 5 and 6 execute. Include an indication of what data the variable x holds.

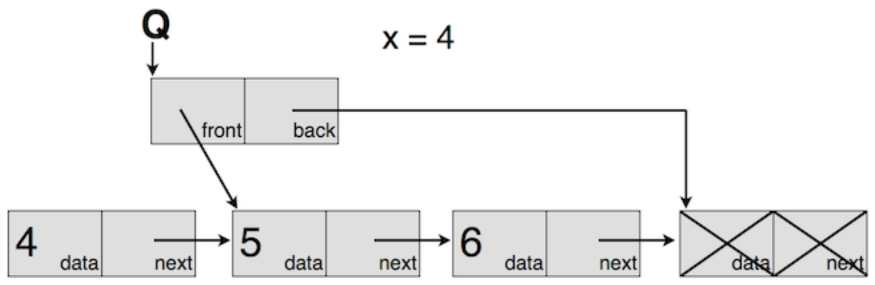
5.

Solution:



6.

Solution:



Visualizing push

This is the code for push, and the definitions of the relevant structs.

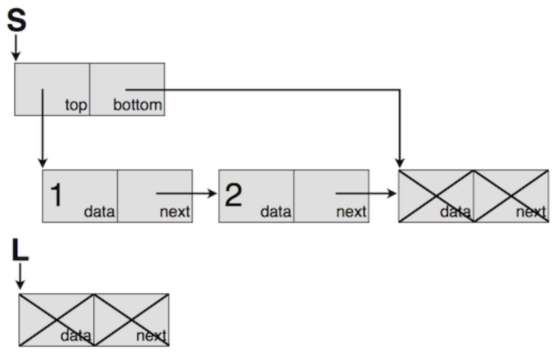
```
struct list_node {
   int data;
   struct list_node *next;
};
```

```
 5 typedef struct list_node list;
 6
 7 struct stack_header {
 8    list *top;
 9    list *bottom;
10 };
11 typedef struct stack_header* stack;
12
13 void push(stack S, int e)
14 //@requires is_stack(S);
15 //@ensures is_stack(S);
16 {
17    list *l = alloc(struct list_node);
18    l->data = e;
19    l->next = S->top;
20    S->top = l;
21 }
```

Suppose `push(S, 3)` is called on a stack `S` that contains before the call, from top to bottom, `(1, 2)`. Draw the state of the stack after each of lines 17 - 20 (inclusive). Include the list struct separately before it has been added to the stack.

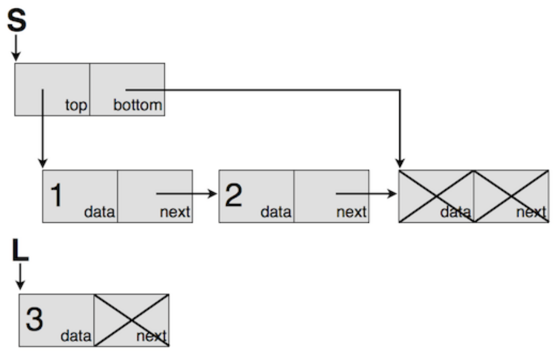
17.

Solution:

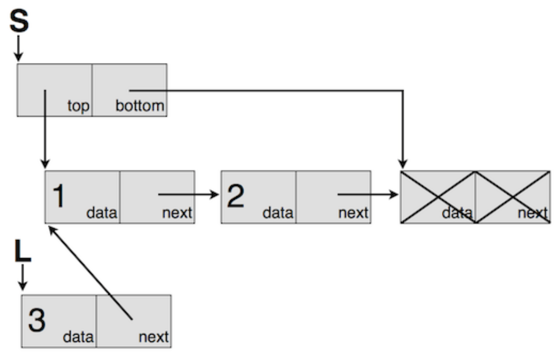


18.

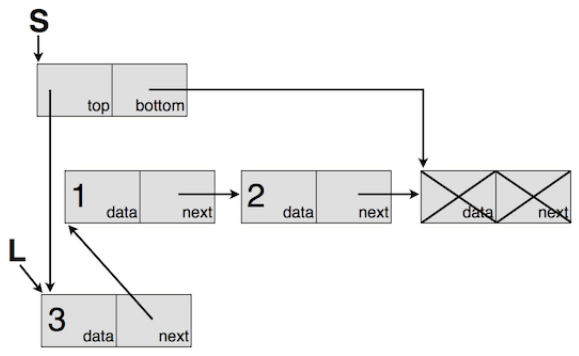
Solution:



19.

Solution: 

20.

Solution: 
S
top
bottom
1
data
next
2
data
next
data
next
L
3
data
next