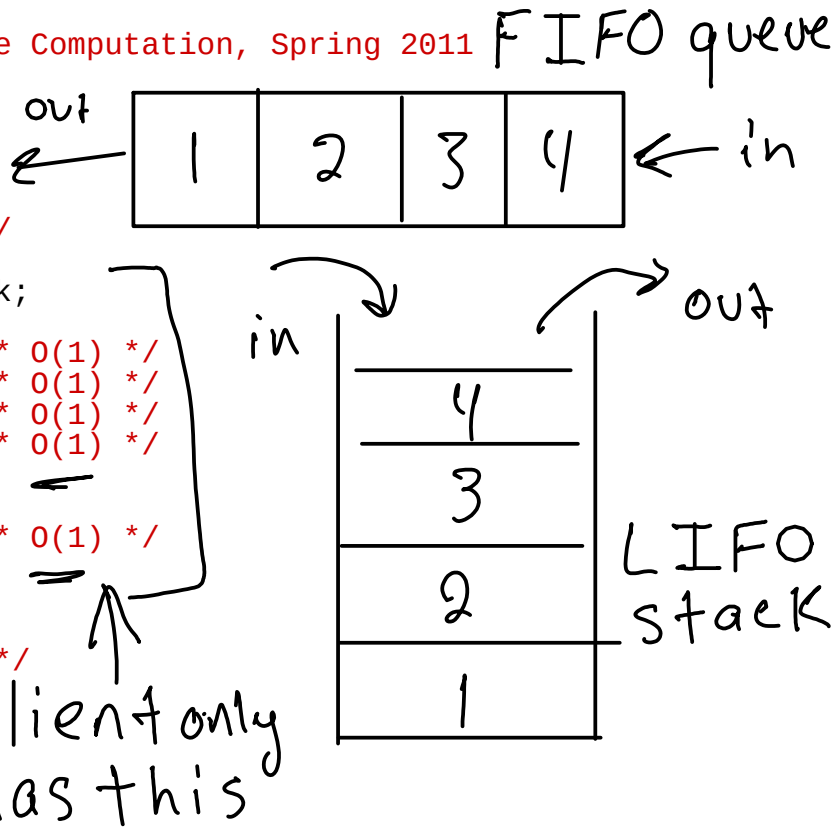
```
/* Stacks
 * 15-122 Principles of Imperative Computation, Spring 2011
 * Frank Pfenning
 */

typedef int elem;

/* Interface to stacks of elems */

typedef struct stack_header* stack;

bool stack_empty(stack S);      /* O(1) */
stack stack_new();              /* O(1) */
void push(stack S, elem e);     /* O(1) */
elem pop(stack S)               /* O(1) */
//@requires !stack_empty(S);
  ;
int stack_size(stack S);        /* O(1) */

/* Implementation of stacks */

/* Aux structure of linked lists */
struct list_node {
  elem data;
  struct list_node* next;
};
typedef struct list_node list;

/* is_segment(start, end) will diverge if list is circular! */
bool is_segment(list* start, list* end) {
  list* p = start;
  while (p != end) {
    if (p == NULL) return false;
    p = p->next;
  }
  return true;
}

int list_size(list* start, list* end)
//@requires is_segment(start, end);
{
  list* p = start;
  int n = 0;
  while (p != end)
    //@loop_invariant is_segment(p, end);
    {
      //@assert p != NULL;
      n = n+1;
      p = p->next;
    }
  return n;
}

/* Stacks */

struct stack_header {
  list* top;
  list* bottom;
  int size;    /* num of elements in stack */
```



```
};

bool is_stack(stack S) {
  if (S == NULL) return false;
  if (S->top == NULL || S->bottom == NULL) return false;
  if (!is_segment(S->top, S->bottom)) return false;
  if (S->size != list_size(S->top, S->bottom)) return false;
  return true;
}

int stack_size(stack S)
//@requires is_stack(S);
{
  return S->size;
}

bool stack_empty(stack S)
//@requires is_stack(S);
{
  return S->top == S->bottom;   /* or: S->size == 0 */
}

stack stack_new()
//@ensures is_stack(\result);
//@ensures stack_empty(\result);
{
  stack S = alloc(struct stack_header);
  list* p = alloc(struct list_node); /* does not need to be initialized! */
  S->top = p;
  S->bottom = p;
  S->size = 0;
  return S;
}

void push(stack S, elem e)
//@requires is_stack(S);
//@ensures is_stack(S);
{
  list* p = alloc(struct list_node);
  p->data = e;
  p->next = S->top;
  S->top = p;
  (S->size)++;
}

elem pop(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
//@ensures is_stack(S);
{
  elem e = S->top->data;
  S->top = S->top->next;
  (S->size)--;
  return e;
}
```