Lecture Notes on Stacks

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 8
September 16, 2010

1 Introduction

In this lecture we introduce another commonly used data structure called a stack. We practice again writing an interface, and then implementing the interface using linked lists as for queues. We also discuss how to check whether a linked list is circular or not.

2 Stack Interface

Stacks are similar to queues in that we can insert and remove items. But we remove them from the same end that we add them, which makes stacks a LIFO (Last In First Out) data structure.

Here is our interface

```
// type elem must be defined
typedef struct stack* stack;
bool is_empty(stack S);         /* O(1) */
stack s_new();                  /* O(1) */
void push(elem x, stack S);     /* O(1) */
elem pop(stack S);              /* O(1) */
```

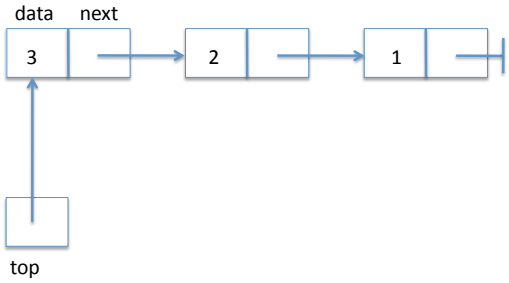
We want the creation of a new (empty) stack as well as pushing and popping an item all to be constant-time operations.

We are being slightly more abstract here than in the case of queues in that we do not write, in this file, what type the elements of the stack have to be. Instead we assume that before the file is read, we have already defined a

type `elem` for the type of stack elements. We say that the implementation is generic or polymorphic in the type of the elements. Unfortunately, neither C nor C0 provide a good way to enforce this in the language and we have to rely on programmer discipline.

3 Stack Implementation

The idea is to reuse linked lists. But since all operations work on one end of the list, we do not need two pointers but just one which we call `top`. A typical stack then has the following form:



Note that the end of the linked list is marked with the special `NULL` pointer that cannot be dereferenced. We define:

```
struct list {
  elem data;
  struct list* next;
};
typedef struct list* list;

struct stack {
  list top;
};
```

To test if some structure is a valid stack, we only need to check that the list starting at `top` ends in `NULL`, which is the same as checking that this is a list segment (as introduced in the last lecture).

```
bool is_stack (stack S) {
  return is_segment(S->top, NULL);
}
```

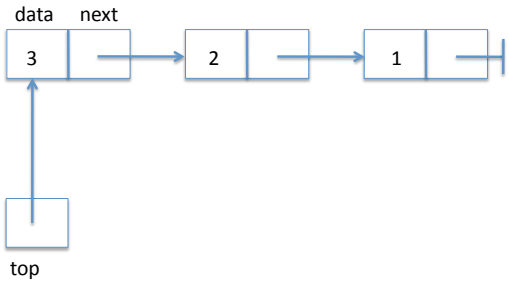
To check if the stack is empty, we only need to verify that `top` is `NULL`.

```
bool is_empty(stack S)
//@requires is_stack(S);
{
  return S->top == NULL;
}
```

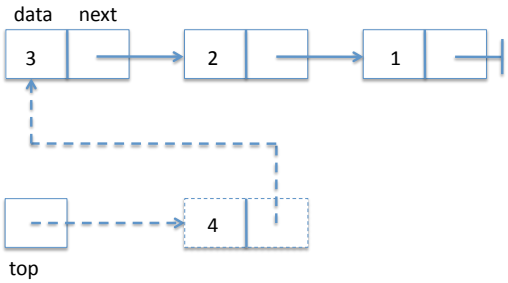
Creating a new stack is very simple, since we only need to set `top` to `NULL` after allocating it.

```
stack s_new()
//@ensures is_stack(\result);
//@ensures is_empty(\result);
{
  stack S = alloc(struct stack);
  S->top = NULL;
  return S;
}
```

To push an element onto the stack, we create a new list item, set its data field and then its next field to the current top of the stack. Finally, we need to update the `top` field of the stack to point to the new list item. While this is simple, it is still a good idea to draw a diagram. We go from



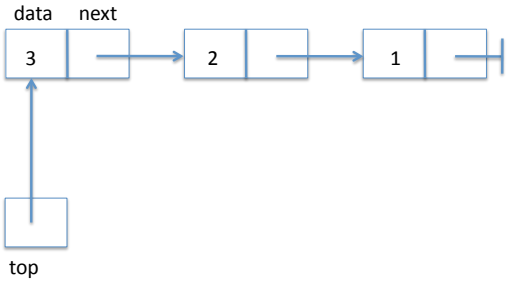
to



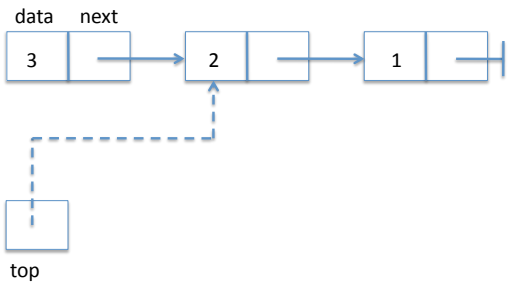
In code:

```
void push(elem x, stack S)
//@requires is_stack(S);
//@ensures !is_empty(S);
{
  list first = alloc(struct list);
  first->data = x;
  first->next = S->top;
  S->top = first;
}
```

Finally, to pop an element from the stack we just have to move the top pointer to follow the next pointer from the top of the stack. As in the case of dequeuing an element from the previous lecture, the list item that previously constituted the top of the stack will no longer be accessible and be garbage collected as needed by the runtime system. We go from



to



In code:

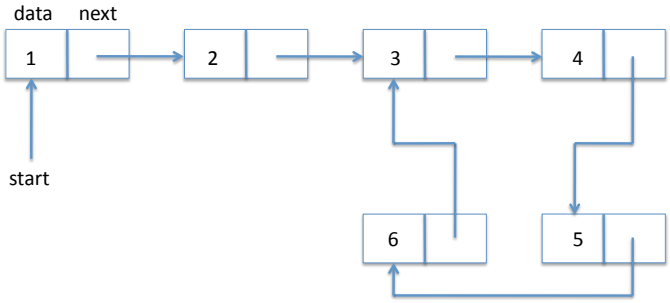
```
elem pop(stack S)
//@requires is_stack(S);
//@ensures is_stack(S);
{ assert(S->top != NULL, "cannot pop empty stack");
  { elem x = S->top->data;
    S->top = S->top->next;
    return x;
  }
}
```

This completes the implementation of stacks, which are a very simple and pervasive data structure. We will exercise them in the next lecture when we implement a virtual machine.

4 Detecting Circularity

Checking whether a stack or a queue satisify their data structure invariant raises an interesting question: what if, somehow, we created a list that

contains a cycle, such as



In that case, following next pointers until we reach `NULL` actually never terminates. The program for checking a segment will get into an infinite loop.

In general, contracts should terminate and have no effects. It is marginally acceptable if a contract diverges, because it will not incorrectly claim that the contract it satisfied, but it would clearly be better if it explicitly rejected a circular list. But how do we check that? Before you read on, you should seriously think about the problem, like our class did in lecture.

Here is the original `is_segment` predicate.

```
bool is_segment(list start, list end)
{ list p = start;
  while (p != end) {
    if (p == NULL) return false;
    p = p->next;
  }
  return true;
}
```

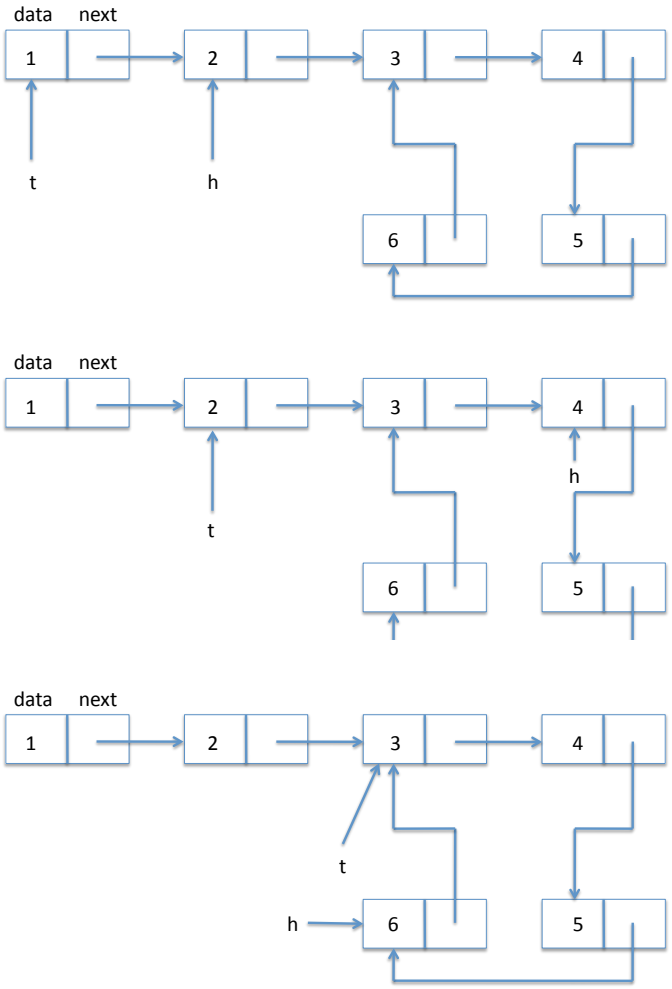
One the simplest solutions proposed in class keeps a copy of the `start` pointer. Then when we advance p we run through an auxiliary loop to check if the next element is already in the list. The code would be something like

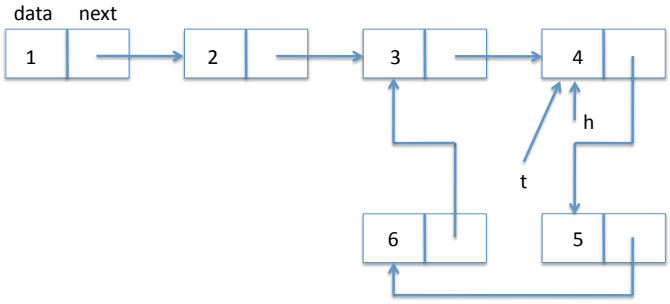
```
bool is_segment(list start, list end)
{ list p = start;
  while (p != end) {
    if (p == NULL) return false;
    { list q = start;
      while (q != p) {
        if (q == p->next) return false; // circular
        q = q->next;
      }
    }
    p = p->next;
 }
 return true;
}
```

Unfortunately this solution requires $O(n^2)$ time for a list with n elements, whether it is circular or not.

Again, consider if you can find a better solution before reading on.

The idea for a more efficient solution is to create *two* pointers, let's name them t and h. t traverses the list like the pointer p before, in single steps. h, on the other hand, skips two elements ahead for every step taken by p. If the slower one t ever gets into a loop, the other pointer h will overtake it from behind. And this is the only way that this is possible. Let's try it on our list. We show the state of t and h on every iteration.





In code:

```
bool is_circular(list l)
{ if (l == NULL) return false;
  { list t = l;       // tortoise
    list h = l->next; // hare
    while (t != h)
      //@loop_invariant is_segment(t, h);
      { if (h == NULL || h->next == NULL) return false;
        t = t->next;
        h = h->next->next;
      }
    return true;
  }
}
```

A few points about this code: in the condition inside the loop we exploit the short-circuiting evaluation of the logical or '||' so we only follow the next pointer for h when we know it is not `NULL`. Guarding against trying to dereference a `NULL` pointer is an extremely important consideration when writing pointer manipulation code such as this.

This algorithm has been called the *tortoise and the hare* and is due to Floyd. We have chosen t to stand for *tortoise* and h to stand for *hare*.

This algorithm has complexity $O(n)$. An easy way to see this was suggested by a student in class: when there is no loop, the hare will stumble over `NULL` after $O(n)$ steps. If there is a loop, then consider the point when the tortoise enters the loop. At this point, the hare must already be somewhere in the loop. Now for every step the tortoise takes in the loop the hare takes two, so on every iteration it comes one closer. The hare will catch the tortoise after at most half the size of the loop. Therefore the overall complexity of $O(n)$: the tortoise will not complete a full trip around the loop.