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 we did for queues.

Relating the lecture to our learning goals:

Computational Thinking: We illustrate the power of abstraction by considering both client-side and library-side of the interface to a data structure.

Algorithms and Data Structures: We consider stacks.

Programming: Once again we use structs and pointers. We will also see a new pattern of loop invariants for working with linked lists.

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:

*With some contributions by André Platzer
/* type elem must be defined */

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) */

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 at the top of the file, or before this 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. We follow the basic structure of our queue implementation, except that we read off elements from the same end that we write them to. We call the pointer to this end top. Since we do not perform operations on the other side of the stack, we do not necessarily need a pointer to the other end. For structural reasons, and in order to identify the similarities with the queue implementation, we still decide to remember a pointer bottom to the bottom of the stack. With this design decision, we do not have to handle the bottom of the stack much different than any other element on the stack. The difference is that the data at the bottom of the stack is meaningless and will not be used in our implementation. A typical stack then has the following form:

![Diagram of a stack implementation]
We define:

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

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

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

```c
bool is_stack (stack S) {
    if (S == NULL) return false;
    if (S->top == NULL || S->bottom == NULL) return false;
    return is_segment(S->top, S->bottom);
}
```

To check if the stack is empty, we only need to check whether top and bottom point to the same element.

```c
bool stack_empty(stack S)
//@requires is_stack(S);
{
    return S->top == S->bottom;
}
```

For creating a new stack, we allocate the “dummy” list element for top and bottom to point to.

```c
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! */
    ```
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

![Diagram 1]

To

![Diagram 2]

In code:

```c
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;
}
```
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

```
  3
  1
  2
  top
  data
  next
  bo/om
  3
  1
  2
  top
  data
  next
  bo/om
```

to

```
  3
  1
  2
  top
  data
  next
  bo/om
  3
  1
  2
  top
  data
  next
  bo/om
```

In code:

```c
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;
    return e;
}
```

This completes the implementation of stacks, which are a very simple and pervasive data structure.
4 Abstraction

An important point about formulating a precise interface to a data structure like a stack is to achieve abstraction. This means that as a client of the data structure we can only use the functions in the interface. In particular, we are not permitted to use or even know about details of the implementation of stacks.

Let's consider an example of a client-side program. We would like to examine the element of top of the stack without removing it from the stack. Such a function would have the declaration

```c
int peek(stack S)  
//@requires !stack_empty(S);  
;
```

The first instinct might be to write it as follows:

```c
int peek(stack S)  
//@requires !stack_empty(S);  
{
    return S->top->data;
}
```

However, this would be completely wrong. Let's recall the interface:

```c
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) */
```

We don't see any top field, or any data field, so accessing these as a client of the data structure would violate the abstraction. Why is this so wrong? The problem is that if the library implementer decided to improve the code, or perhaps even just rename some of the structures to make it easier to read, then the client code will suddenly break! The interface does not serve the purpose it is intended for, namely provide a reliable way to work with a data structure.

So what can we do? It is possible to implement the peek operation without violating the abstraction! Consider how before you read on.
The idea is that we pop the top element off the stack, remember it in a temporary variable, and then push it back onto the stack.

```c
int peek(stack S)
//@requires !stack_empty(S);
{
    int x = pop(S);
    push(S, x);
    return x;
}
```

This is clearly less efficient: instead of just dereferencing two pointers we actually have to pop and element and then push it back onto the stack. However, it is still a constant-time operation ($O(1)$) since both pop and push are constant-time operations. Nonetheless, we have a possible argument to include a function `peek` in the interface.

If we are actually prepared to extend the interface, then we can go back to our original implementation.

```c
int peek(stack S)
//@requires !stack_empty(S);
{
    return S->top->data;
}
```

Is this a good implementation? Not quite. First we note that inside the library we should refer to elements as having type `elem`, not `int`. Of course, this is purely a stylistic matter because these two are synonyms.

Second we note that we are now missing a precondition. In order to even check if the stack is non-empty, we first need to be assured that it is a valid stack. On the client side, all elements of type `stack` come from the library, and any violation of data structure invariants could only be discovered when we hand it back through the library interface to a function implemented in the library. Therefore, the client can assume that values of type `stack` are valid and we don’t have explicit pre- or post-conditions for those. Inside the library, however, we are constantly manipulating the data structure in ways that break and then restore the invariants, so we should check if the stack is indeed valid.
From these two considerations we obtain the following code for \textit{inside the library}:

\begin{verbatim}
elem peek(stack S)
//@requires is_stack(S);
//@requires !stack_empty(S);
{
    return S->top->data;
}
\end{verbatim}

Why is the access to \code{S->top->data} safe? It is critical that we always convince ourselves that pointers that we use to access data are not null. Here we reason as follows:

1. Since \code{S} is a valid stack (\code{is_stack(S)}), \code{S} is not null and the dereference \code{S->top} is safe.

2. Since the stack is not empty (\code{!stack_empty(S)}), we know that \code{S->top \neq S->bottom}. Moreover, since \code{S} is a valid stack, we know \code{is_segment(S->top, S->bottom)}. From these two facts it follows that \code{S->top \neq NULL} (otherwise we could not reach \code{S->bottom} from \code{S->top}) and the dereference \code{S->top->data} is safe.

\section{Computing the Size of a Stack}

Let’s exercise our data structure once more by developing two implementations of a function that returns the size of a stack: one on the client’s side, using only the interface, and one on the library’s side, exploiting the data representation. Let’s first consider a client-side implementation, using only the interface so far.

\begin{verbatim}
int stack_size(stack S);
\end{verbatim}

Again, we encourage you to consider this problem and program it before you read on.
First we reassure ourselves that it will not be a simple operation. We do not have access to the linked lists (in fact, we cannot know how it is implemented), so the only thing we can do is pop all the elements off the stack. This can be accomplished with a prototypical while-loop that finishes as soon as the stack is empty.

```c
int stack_size(stack S) {
    int count = 0;
    while (!stack_empty(S)) {
        pop(S);
        count++;
    }
    return count;
}
```

However, this function has a big problem: in order to compute the size we have to destroy the stack! Clearly, there may be situations where we would like to know the number of elements in a stack without deleting all of its elements. Fortunately, we can use the idea from the peek function in amplified form: we maintain a new temporary stack $T$ to hold the elements we pop from $S$. Once we are done counting, we push them back onto $S$ to repair the damage.

```c
int stack_size(stack S) {
    stack T = stack_new();
    int count = 0;
    while (!stack_empty(S)) {
        push(T, pop(S));
        count++;
    }
    while (!stack_empty(T)) {
        push(S, pop(T));
    }
    return count;
}
```

The complexity of this function is clearly $O(n)$, where $n$ is the number of elements in the stack $S$, since we traverse each while loop $n$ times, and perform a constant number of operations in the body of both loops. For that, we need to know that push and pop are constant time ($O(1)$).
What about a library-side implementation of stack_size? As before, this can be done more efficiently with respect to the constant factor and space allocation, but asymptotic complexity remains the same.

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

```c
int stack_size(stack S)
//@requires is_stack(S);
{
    return list_size(S->top, S->bottom);
}
```

You should convince yourself that the precondition for list_size is satisfied where called, since we know is_stack(S). But how do we know the assertion that p != NULL? We need it in order to show that the dereference p->next is safe. Clearly, p changes in the loop by assignment, so we need a loop invariant!

Looking the definition of a list segment, we see that while we iterate through the linked list, the part of the list from p to end will shrink, but it will continue to be a valid segment. And, as before, if we have non-empty segment then its beginning cannot be null.
int list_size(list* start, list* end)
//@requires is_segment(start, end);
{
    list* p = start;
    int count = 0;
    while (p != end)
        //@loop_invariant is_segment(p, end);
        {
            //@assert p != NULL;
            count++;
            p = p->next;
        }
    return count;
}

6 Exploiting Abstraction and Data Structure Invariants

The stack_size operation from the previous section is still a linear-time operation \(O(n)\), because we have to traverse the whole linked list.

Can we do better? Yes! We can make it constant time, which may seem like magic. Think about it before you read on.
The trick is to introduce a new field into the header struct of a stack which contains the size. We initialize it to 0 when we create the stack. Every time we push an element onto the stack we increment the size field, every time we pop an element from the stack we decrement the size field. You can find the full code in the file `stacks.c0`. Here we show only the new header structure definition and the `is_stack` function which checks the invariants.

```c
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) {
    return S->size;
}
```

Note that in `is_stack` we still use the function `list_size` we wrote before, but now it is just used to check that the data structure invariants are preserved. When we actually retrieve the size we just look at the field.

Why is this correct? Here we can bring the power of abstraction to bear. Since no client can directly access the size field, or the top or bottom pointers, we only have to show that each of the operations listed in the interface maintains the invariant. And this is the case, if we have correct push and pop functions. For example,
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;
}
Exercises

**Exercise 1** Consider what would happen if we pop an element from the empty stack when contracts are not checked? When does an error arise?

**Exercise 2** Consider the client-side implementation of stack_size. We don’t have the vocabulary of predicates, but can you explain any loop invariants that might hold for the first and second loop? Are they strong enough to ensure that we really return the number of elements in S and that the stack S is the same at the end as at the beginning?

**Exercise 3** Stacks are usually implemented with just one pointer in the header, to the top of the stack. Rewrite the implementation in this style, dispensing with the bottom pointer, terminating the list with NULL instead.