Data Structures

Stack Implementation in C

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Stack

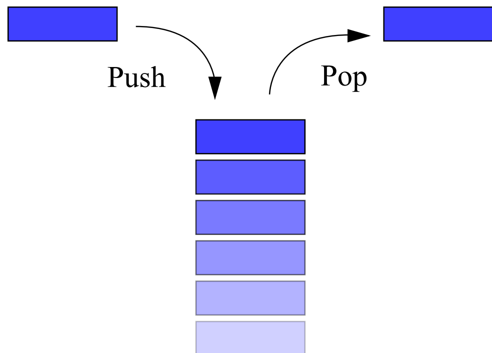
as an abstract data type

A push down stack is an abstract data type which include the following operations:

- Insert a new element, PUSH(S,x).
- Delete the last element which was added in the stack, POP(S).
- Check if the stack is empty, EMPTY(S).
- Return the size of the stack, SIZE(S).
- ..

Stack

A stack follows the **last-in, first-out** (LIFO) principle.



Stack

Some requirements:

- Separate interface from implementation
- Users are not allowed to change the internal representation (information-hiding)
- Users can create multiple stacks at the same time

Stack

interface in C (stack.h)

```
#ifndef _STACK_H
#define _STACK_H

typedef int item_type;
typedef struct _stack* stack;

stack stack_create();
void stack_destroy(stack s);

void stack_push(stack s, item_type elem);
void stack_pop(stack s);
item_type  stack_top(stack s);

int  stack_is_empty(stack s);
int  stack_size(stack s);
void stack_clear(stack s);
#endif
```

Stack Implementation

Representation (stack.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct _stack {
    item_type* array;
    int max_size;
    int top;
};
```

Stack Implementation

Create (stack.c)

```
static stack stack_create_maxsize(int max_size) {
    if (max_size <= 0) {
        fprintf(stderr, "Wrong stack size (%d)\n", max_size);
        abort();
    }

    stack s = (stack) malloc(sizeof(struct _stack));
    if (s == NULL) {
        fprintf(stderr, "Insufficient memory to initialize stack.\n");
        abort();
    }

    item_type *items;
    items = (item_type *) malloc(sizeof(item_type) * max_size);
    if (items == NULL) {
        fprintf(stderr, "Insufficient memory to initialize stack.\n");
        abort();
    }

    s->array = items;
    s->max_size = max_size;
    s->top = -1;
    return s;
}
```

Stack Implementation

Create (stack.c)

```
stack stack_create() { return stack_create_maxsize(64); }
```

Stack Implementation

Destroy (stack.c)

```
void stack_destroy(stack s) {
    if (s == NULL) {
        fprintf(stderr, "Cannot destroy stack\n");
        abort();
    }

    free(s->array);
    free(s);
}
```

Stack Implementation

Double the Capacity (stack.c)

```
static
void stack_doublesize(stack s) {
    if (s == NULL) {
        fprintf(stderr, "Cannot double stack size\n");
        abort();
    }

    // create double the stack
    int new_max_size = 2 * s->max_size;
    item_type* new_array = (item_type*) malloc(sizeof(item_type) * (new_max_size));
    if (new_array == NULL) {
        fprintf(stderr, "Insufficient memory to double the stack.\n");
        abort();
    }

    // copy elements to new array
    int i;
    for(i = 0; i <= s->top; i++) {
        new_array[i] = s->array[i];
    }

    // replace array with new array
    free(s->array);
    s->array = new_array;
    s->max_size = new_max_size;
}
```

Stack Implementation

Push (stack.c)

```
void stack_push(stack s, item_type elem) {
    // increase capacity if necessary
    if (s->top >= s->max_size-1) {
        stack_doublesize(s);
    }

    // push the element
    s->array[++s->top] = elem;
}
```

Stack Implementation

Pop and Top(stack.c)

```
void stack_pop(stack s) {
    if (stack_is_empty(s)) {
        fprintf(stderr, "Can't pop element from stack: \
            stack is empty.\n");
        abort();
    }

    s->top--;
}

item_type stack_top(stack s) {
    if (stack_is_empty(s)) {
        fprintf(stderr, "Stack is empty.\n");
        abort();
    }

    return s->array[s->top];
}
```

Stack Implementation

Empty and Size (stack.c)

```c
int stack_is_empty(stack s) {
    if (s == NULL) {
        fprintf(stderr, "Cannot work with NULL stack.\n");
        abort();
    }

    return s->top < 0;
}

int stack_size(stack s) {
    if (s == NULL) {
        fprintf(stderr, "Cannot work with NULL stack.\n");
        abort();
    }

    return s->top+1;
}
```

Stack Implementation

Clear (stack.c)

```c
void stack_clear(stack s) {
    if (s == NULL) {
        fprintf(stderr, "Cannot work with NULL stack.\n");
        abort();
    }

    s->top = -1;
}
```

Testing the Stack

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

int main() {
    stack s;
    s = stack_create();

    for(int i = 0; i < 1000; i++) {
        stack_push(s, i);
        assert(stack_top(s) == i);
        assert(stack_is_empty(s) == 0);
        assert(stack_size(s) == i+1);
    }

    for(int i = 999; i >= 0; i--) {
        assert(stack_top(s) == i);
        stack_pop(s);
    }

    stack_destroy(s);

    return 0;
}
```

Example using Stack

A classic application for a stack is to check whether the correct closing of brackets and parantheses in a written in a language like C or Java.

If we see a left bracket or a left parantheses we push it in a stack. If we see a right bracket or parantheses we must match it with one left from the top of the stack.

This simple strategy breaks down if the program contains parentheses or brackets inside characters or strings like: `printf(")");`

Example using Stack

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "stack.h"

int main() {
    stack s;
    s = stack_create();

    int is_correct, c, top;
    is_correct = 1;

    while((c=getchar())!=EOF) {
        if (c == '(' || c == '{') {
            stack_push(s, c);
        } else if (c == ')') {
            if (stack_is_empty(s)) {
                is_correct = 0;
                break;
            }
            top = stack_top(s);
            stack_pop(s);
            if (top != '(') {
                is_correct = 0;
                break;
            }
        }
```

Example using Stack

```
        else if (c == '}') {
            if (stack_is_empty(s)) {
                is_correct = 0;
                break;
            }
            top = stack_top(s);
            stack_pop(s);
            if (top != '{') {
                is_correct = 0;
                break;
            }
        }
    }

    if (!stack_is_empty(s)) {
        is_correct = 0;
    }

    if (is_correct) {
        printf("OK\n");
    } else {
        printf("ERROR\n");
    }

    stack_destroy(s);

    return 0;
}
```