Assignement Report - Stacks

Sam Serbouti serbouti@kth.se 20010513-3284
TCOMK Algorithms and Data Structures ID1021 HT24

September 4, 2024

A stack is a data structure based on two operations : push and pop that follow a first-in last-out logic. Push adds a value to the top of the stack and pop takes it out and returns it. I choose to have a top pointer that points **above** the stack. This means that the actual value stored in memory where it points to is meaningless to us but this proves useful to us as when the value of top = 0 then it means that the stack is empty. I will implement this data structure in C using arrays. This will give us two versions of the structure depending on how I manage the array's size : one static and one dynamic.

1 Building a static stack

1.1 Initialization

```
stack *new_stack(int size) {
    ...
    //memory allocation

    stk->top=0;
    stk->size=size;
    stk->array=array;
    return stk;
}
```

1.2 Push

```
void push(stack *stk, int val){
    if(stk->top < stk->size){
        stk->array[stk->top]=val;
        stk->top++;}

    else{printf("Stack overflow\n");}
}
```

1.3 Pop

```
int pop(stack *stk) {
    if (stk->top > 0){
        stk->top--;
        return stk->array[stk->top];
    }
    else{
        printf("Stack underflow\n");
```

```
        return 0;
    }
}
```

1.4 Testing in a 4 items stack

I test this implementation with a stack that starts with an array of size 4, to which I push 10 items, display the stack's content and then pop until I get an underflow.

```
Stack overflow error    //6 overflows after 4 successfull pushes
Stack overflow error
Stack overflow error
Stack overflow error
Stack overflow error
Stack overflow error
stack[0] : 30           //expected contents from the stack
stack[1] : 31
stack[2] : 32
stack[3] : 33
pop : 33
pop : 32
pop : 31
pop : 30
Stack underflow         //underflow when top=0
```

2 Implementing a dynamic stack

In a dynamic implementation, initialization won't change but push and pop will.

2.1 Changing push

Whenever, after a push operation, the top pointer reaches the size of the array, the stack is full. But instead of sending a `Stack overflow` error, I create a new int array, double the size of the previous array, and copy its values. I also don't forget to add the values the user wants to push in the stack like before.

```
void push(stack *stk, int val) {
    if (stk->top == stk->size) {
        int newSize = stk->size*2;
        stk->size = newSize;
        int *copy = (int*)malloc(newSize*sizeof(int));
        for (int i = 0; i < newSize/2; i++) {
```

```
            copy[i] = stk->array[i];
        }
        free(stk->array);
        stk->array = copy;
        stk->array[stk->top] = val;
        stk->top++;
    }
    else{
        stk->array[stk->top] = val;
        stk->top++;
    }
}
```

Changing pop

We could have left the pop function unchanged. However, to optimize it, I add a mechanism that reduces the size of the array by half whenever the top pointer goes below one third of the size the array (but the array must still be larger than 4 items). I chose a one third threshold and not half because otherwise, if the user wishes to push and pop values around this half value, the machine would have to keep doubling and halfling the size of the array.

```
int pop(stack *stk) {
    if (stk->top > 0){
        if(stk->top <= 0.3*stk->size && stk->size > 4){
            int newSize=stk->size/2;
            stk->size = newSize;
            int *copy = (int*)malloc(newSize*sizeof(int));
            for (int i = 0; i<stk->top;i++){
                copy[i] = stk->array[i];
            }
            free(stk->array);
            stk->array = copy;
        }
        stk->top--;
        return stk->array[stk->top];
    }
    else{
        printf("Stack underflow\n");
        return 0;
    }
```

Here is an extract of the console when I try to push 20 items and pop 15. We can see that the size is reduced by half.

```
(push 30)        top: 1          stack[0]: 30    array size: 4
...
(push 49)        top: 20         stack[19]: 49   array size: 32
--------
(pop 49)         top: 19         stack[18]: 48   array size: 32
...
(pop 40)         top: 10         stack[9]: 39    array size: 32
(pop 39)         top: 9          stack[8]: 38    array size: 32
(pop 38)         top: 8          stack[7]: 37    array size: 16
...
(pop 35)         top: 5          stack[4]: 34    array size: 16
```

3 Implementing a reverse Polish calculator

Using our data structure, we can build a simple calculator that uses Polish reverse notations. Whenever an operation is typed ($+$, $-$, $*$, $/$) it pops two items from the stack and pushes the result of the operation.

The extract code is displayed on two columns left to right to ease reading.

```
printf(" > ");
getline(&buffer, &n, stdin);
if(strcmp(buffer,"\n")==0){
    run = false;
}

else if(strcmp(buffer,"+\n")==0){
    int a = pop(stk);
    int b = pop(stk);
    push(stk, a+b);
}

else if(strcmp(buffer,"-\n")==0){
    int a = pop(stk);
    int b = pop(stk);
    push(stk, a-b);
}
```

```
else if(strcmp(buffer,"*\n")==0){
    int a = pop(stk);
    int b = pop(stk);
    push(stk, a*b);
}

else if(strcmp(buffer,"/\n")==0){
    int a = pop(stk);
    int b = pop(stk);
    push(stk, a/b);
}

else{
    int val = atoi(buffer);
    push(stk, val);
}
```

3.1 Results

When inputing the sequence: `4 2 3 * 4 + 4 * + 2 -` whether on the static or dynamic implementation of the calculator, I obtain **-42**. It fits the result I was expecting when calculating by hand:

$$2 - ((((2 \cdot 3) + 4) \cdot 4) + 4) = -42$$