Programmering for computerteknologi Hand-in Assignment Exercises

Week 7: Programming larger software projects

Written by: Alexander A. Christensen (202205452)

Disclaimer: Due to errors with CMake that neither me, nor the TAs have solved, the test-cases have not been run. Instead, the functions have been manually tested.

The code can still be found at https://github.com/Aarhus-University-ECE/assignment-7-A-CHRI

Exercise 1

We wish to create a function that computes the Taylor series for a sine function. This can be written mathematically by

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

Implementation

The x can be compute as $\frac{x^1}{1!}$, which will easy implementation. To alternate between adding and subtracting, we will look at the iteration variable, and compute whether it is *even* or *odd*, adding at odd numbers and subtracting at even numbers.

```
double taylor_sine(double x, int n)
{
    /* Pre: Terms n, is a positive integer */
    assert(n > 0);

    /* Post: Compute the taylor value of sine */
    double r = 0;
    for (int i = 1; i <= n; i++)
    {
        if (i % 2 == 0)
            r -= pow(x, 2 * i - 1) / fact(2 * i - 1);
        else
            r += pow(x, 2 * i - 1) / fact(2 * i - 1);
    }
    return r;
}
```

Note: We use the `math.h` package to use the `pow` function. A script for computing the factorial has been written, and implemented as shown below.

```
double fact(double x)
{
    /* Pre: Non-negative integer */
    assert(x > 0);

    /* Post: Recursively compute the factorial of x */
    if (x == 1)
        return 1;
    return x * fact(x - 1);
}
```

The function has been implemented as a library and is linked to the test-file during compilation.

Testing

We wish to test the function using various values of `x` and `n`. Using the datatype double, for the factorial function lets us pick a higher amount of terms. Using the datatype integer, would limit `n` to a max value of 6, since $13! > 2,147,483,647$.

Testing will be done for various values of `x`, each with `n` set to 2, and 6, respectively. The chosen values of `x` is different values around a circle of various sizes.

x	n	Result	ANSI-C
$1/2$	2	0.48	0.48
$1/2$	6	0.48	0.48
$\pi/2$	2	0.92	1.00
$\pi/2$	6	1.00	1.00
$3\pi/2$	2	-12.7	-1.00
$3\pi/2$	6	-1.08	-1.00
$9\pi/2$	2	-456	1.00
$9\pi/2$	6	-69e3	1.00
$9\pi/2$	15	47.7	1.00

Tabel 1: Test-cases for the taylor_sine function

We notice that the higher the value of `x`, the more volatile the answer. Upping the terms `n` to 15 for $x = 9\pi/2$, increases the accuracy significantly. The test has been implemented as shown below.

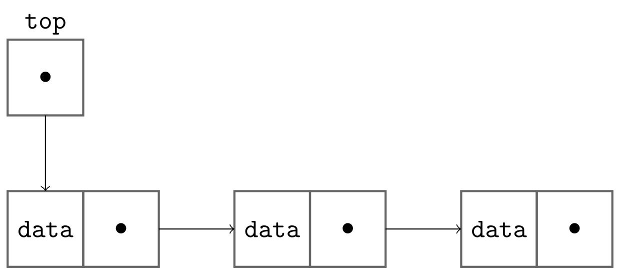
```c
#include "taylor_sine.h"
#define PI 3.14159265358979323846

int main(void)
{
    double x[9] = {0.5, 0.5, 0.5 * PI, 0.5 * PI, 1.5 * PI, 1.5 * PI, 4.5 * PI, 4.5 * PI, 4.5 * PI};
    int n[9] = {2, 6, 2, 6, 2, 6, 2, 6, 15}; // Max terms is 6 since, 13! is too big for an integer.

    for (int i = 0; i < 9; i++)
    {
        printf("\nTaylor-sine function for %f, with %d terms: %f", x[i], n[i], taylor_sine(x[i], n[i]));
        printf("\nANSI C sine function for %f: %f\n", x[i], sin(x[i]));
    }
    /* Results: For higher values of x the result is much more volitile.
     * Generally the precision n will increase the accuracy of the result
     */
    return 0;
}
```

Exercise 2

We wish to implement the stack data structure. We do this using a linked list, where the front node, acts as the top of the stack. Using a linked list, we can allow the stack to grow and shrink dynamically. Using a linked list we need to implement the following properties:

- **Initialize:** Set the top pointer to `NULL`
- **Push**: Push element `x` to stack `s`
- **Pop**: Remove and return the top element from stack `s`
- **Empty**: Boolean value, if stack `s` is empty

When using a linked list to implement a stack, we initialize the stack, by creating a `top` pointer, pointing at `NULL` – essentially an empty pointer. This pointer will always be pointing at the top element of the list. Pushing a new element to the list is then as simple as letting the node point to the previous top node, and let the `top` pointer point to the newly added node.



Implementation

When the stack is represented using a single pointer pointing to the top element, all the property functions should take in a pointer to the top pointer. This allows us to manipulate the stack using `void` functions.

```
typedef struct node
{
    int data;
    struct node *next;
} node;

void Initialize(node **top) {
    /* Set the top pointer to NULL */
    *top = NULL;
}

void Push(int x, node **top)
{
    /* Pre: Non-full stack */

    /* Post: Add element x to the top/front of the list */
    node *new = (node *)malloc(sizeof(node)); // Allocate memory for the node

    new->data = x;
    new->next = *top;

    /* Set the top node as the newly added node */
    *top = new;
}

int Pop(node **top)
{
    /* Pre: Non-empty stack */
    assert(*top != NULL);

    /* Post: Free the top node, and return its value */
    node *t = *top;
    *top = (*top)->next;

    /* Pull the data from the node, then free it*/
    int temp = t->data;
    free(t);

    /* Return the data */
    return temp;
}

bool Empty(node **top)
{
    /* Post: Return TRUE if the top node is NULL*/
    return *top == NULL;
}
```

Testing

For testing the implementation we wish to complete the following tests.

1. After executing `Initialize(s);` the stack `s` must be empty
2. After executing `Push(x,s); y = Pop(s);` the stack s must be the same as before execution of the two commands, and `x` must equal `y`
3. After executing `Push(x0,s); Push(x1,s); y0 = Pop(s); y1 = Pop(s);` the stack `s` must be the same as before execution of the two commands, `x0\verb` must equal `y1`, and `x1` must equal `y0`

For displaying the stack at any point we've written a `Display(s)` function, as shown below.

```c
void Display(node **top)
{
    /* Pre: Non-empty stack */
    assert(*top != NULL);

    /* Post: Print each element of the stack in order */
    node *t = *top;
    while (t != NULL)
    {
        printf("%d ", t->data);
        t = t->next;
    }
    printf("\n");
}
```

In the `main` function we've implemented the 3 test-scenarios. These can be seen below.

```c
int main(void)
{
    /* Initialise the stack */
    node *s;
    Initialize(&s);
    // Initialize(s);

    /* TEST A: After initialization the stack must be empty */
    if (Empty(&s))
        printf("The stack is empty after initialization.\n");
    else
        printf("The stack is NOT empty!\n");

    /* Push some elements to the stack */
    Push(1, &s);
    Push(2, &s);

    /* TEST B: After pushing an element to the stack and popping, the stack must be the same */
    Display(&s);
    Push(3, &s);
    Pop(&s);
    Display(&s);

    /* TEST C: After pushing two elements to the stack and popping twice, the two elements should be correctly distributed */
    Push(10, &s);
    Push(20, &s);
    printf("First element popped, should be latest element pushed (20): %d\n", Pop(&s));
    printf("Second element popped, should be second element pushed (10): %d\n", Pop(&s));

```

```
    return 0;
}
```

This prints the following to the console:

```
The stack is empty after initialization.
2 1
2 1
First element popped, should be latest element pushed (20): 20
Second element popped, should be second element pushed (10): 10
```

This shows that the implementation holds up to the 3 given test scenarios

Exercise 3