A4002 - Algorithms, Data Structures and Problem Solving

-----------------------------------------------------------------------------------------------------------------

**Lab3 - Group 1**

-----------------------------------------------------------------------------------------------------------------

Eric Heindl

Esteban Rueda

1. **structs - interval.c**
   1. *The interval.c source file contains a sketch of what an interval arithmetic package might contain in C. There is a type called Interval which stores a lower and an upper bound, and functions to create, add, and print intervals. When you compile and run this program, it simply prints two intervals and their sum.*

**Output**:

aa = [ 2, 3]

bb = [ -10, 50]

cc = [ -8, 53]

* 1. *Change the Interval declaration so that it stores the bounds in double format instead of integers. Make sure to change all the other functions and variables accordingly.*

*Reminder: a good way to catch potential errors is to use the “-Wall” option of gcc, so the Makefile entry should look like this:*

*gcc -Wall -o interval interval.c*

You can replace a name in the whole Code (in Visual Code), by press F1 and then search for Replace.

#include <stdio.h>

typedef struct { **double** low, high;} Interval;

Interval create (**double** low, **double** high){

Interval ii;

ii.low = low;

ii.high = high;

return ii;

}

Interval add (Interval lhs, Interval rhs){

lhs.low += rhs.low;

lhs.high += rhs.high;

return lhs;

}

void prdouble (Interval ii){

printf ("[% **4lf**, % **4lf**]", ii.low, ii.high);

}

int main (int argc, char \*\* argv)

{

Interval aa, bb, cc;

aa = create ( 2, 3);

bb = create (-10, 50);

cc = add (aa, bb);

printf ("aa = ");

**prdouble** (aa);

printf ("\nbb = ");

**prdouble** (bb);

printf ("\ncc = ");

**prdouble** (cc);

printf ("\n");

return 0;

}

* 1. *None of the functions currently verify that low <= high but this is a fundamental property required of intervals. Write a normalize function which takes an interval and, if necessary, modifies it in-place such that it respects that property. This means that you have to add the following function definition to your source file (place it after the create function):*

**void normalize (Interval \* ii)**

**{**

**if (ii->low > ii->high){**

**double temp= ii->low;**

**ii->low = ii->high;**

**ii->high = temp;**

**}**

**}**

Interval create (double low, double high)

{

Interval ii;

ii.low = low;

ii.high = high;

**normalize(&ii);**

return ii;

}

* 1. *Change the initialization of bb to “bb = create(50, -10)” but do not call normalize on it. This should produce an erroneous result in the additon (try it out). Now make the add function robust to non-normalized inputs, by adding two calls to normalize as the first two lines in its implementation, like this:*

*Interval add (Interval lhs, Interval rhs)*

*{*

*normalize (&lhs);*

*normalize (&rhs);*

*/\* ... the rest of the add function remains the same \*/*

*}*

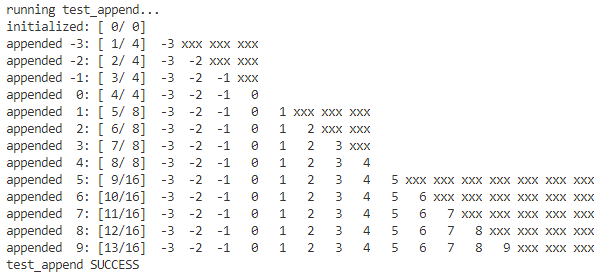
*\*\*\*We normalize in create() so we don’t need to normalize in add() but if the interval comes from a different source we need to normalize after the addition.\*\*\**

*Verify that you get the correct answer again. However, as can be seen by the program output, bb will remain non-normalized.*

1. **vectors - vector.c**

*The source file vector.c is a partial implementation of a dynamic array. It closely follows the example presented in lecture 2. There are several functions that still need to be filled in, but the source already provides an application that is ready to run.*

* 1. *Compile and run the vector program. Notice that it performs a series of checks, called unit tests, which allow to quickly determine whether an implementation is complete. Study the vector\_append function (starting on line 89 of vector.c) and make sure you understand what it does by matching it with the relevant part of the messages printed by the vector program.*

**

* 1. *Fill in the function vector\_prepend (starting on line 127 of vector.c). It is probably easiest to start by studying what vector\_append does, and following the same general implementation pattern. Verify that your implementation is correct by running the vector program and analysing the output produced by the test\_prepend function.*

int vector\_prepend (Vector \* vec, int value)

{

if (vec->len >= vec->cap) {

if (0 != vector\_grow (vec))

return -1;

}

int length = (int) vec->len;

if(length!=0){

while(length >0){

vec->arr[length]=vec->arr[length-1];

length--;

}

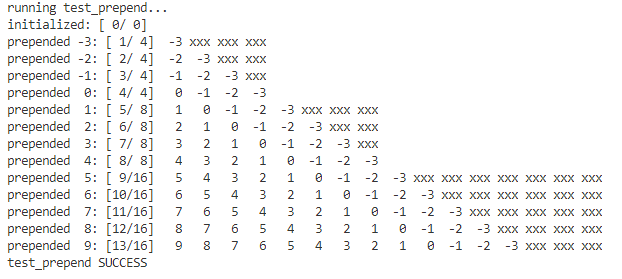
}

vec->arr[0] = value;

vec->len++;

return 0;

}

**

* 1. *Fill in the function vector\_insert (starting on line 153 of the original vector.c but that will have changed after you implemented vector\_prepend). You can get inspiration from the vector\_prepend function that you just implemented. Again, verify that your implementation is correct by analysing the unit test output produced by running the vector program.*

int vector\_insert (Vector \* vec, unsigned long pos, int value)

{

if (vec->len >= vec->cap) {

if (0 != vector\_grow (vec))

return -1;

}

int length = (int) vec->len;

printf("%d",length);

while(length >pos){

vec->arr[length]=vec->arr[length-1];

length--;

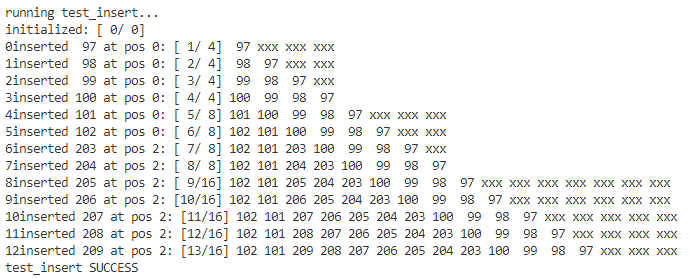
}

vec->arr[pos] = value;

vec->len++;

return 0;

}

**

* 1. *Fill in the vector\_remove function (originally starting on line 183 of vector.c). By now you should have the hang of this kind of test-driven implementation, using repeated runs of the unit tests to find out when you’re done.*

*We comment the if statement in the test\_remove*

*// if (0 == vector\_remove(&vec, 11)) {*

*, because it called the vector\_remove function without removing a value. => vec->len chnaged from 6 to 5*

*=> causes a error in the next test.*

int vector\_remove (Vector \* vec, unsigned long pos)

{

int length = (int) vec->len;

if(pos>=length){return -1;}

while(pos<length){

vec->arr[pos]=vec->arr[pos+1];

pos++;

}

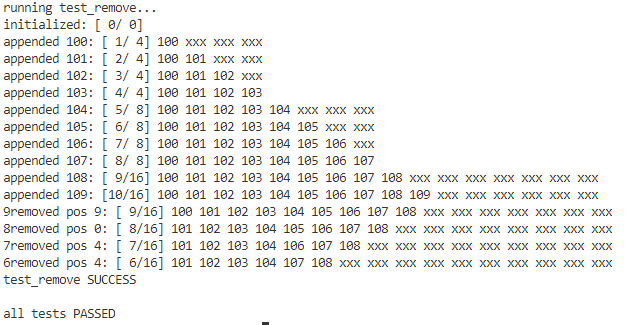
vec->arr[vec->len-1] = 0;

vec->len--;

printf("%lu",vec->len);

return 0;

}

**