About the C coding exam 2019-01-14.

`// Imperative and Object-Oriented Programming Methodology`
`// Uppsala University - Copyright © 2019 Henrik Alfken(Schulze)`

Seeing a solved coding exam does not automatically mean it is easy to understand how the coder arrived at that particular solution. Here is a rather detailed narrative about my efforts to solve the C coding exam of 2019-01-14. I use Bash on Windows (Windows Subsystem for Linux) inside ConEmu, because I find it convenient, https://conemu.github.io/en/BashOnWindows.html. As text editor I (mainly) use Geany which is easy to use and available on the department's linux servers. I have installed the Geany-plugin "Save Actions" because it is very convenient to have files saved as soon as focus is lost. (One thing less to think about when jumping back and forth between the text editor and the terminal.)

1. Check what TO DO.

Open a terminal at `uppgift1`, and type `grep -irn 'todo'`, like so:

```
<path-to-here>/uppgift1$ grep -irn 'todo'
yourcode.c:29:  /// TODO: lägg till enligt spec
yourcode.c:39:  /// TODO: lägg till enligt spec
yourcode.c:50:  /// TODO: frivillig hjälpfunktion
yourcode.c:59:  /// TODO: lägg till enligt spec
```

Taking a look inside the file `yourcode.c` shows that line 29 concerns `treemap_insert()`, line 39 is about `treemap_lookup()`, line 50 concerns `node_destroy()`, and line 59 `treemap_destroy()`.

2. Read (and copy-paste) the specifications.

A natural step to follow is to check out the specifications for these three public functions – `insert`, `lookup`, and `destroy` – as given on lines 11-22 of `treemap.h`:

```c
/// Removes the treemap from memory, but does not call
/// free on any value pointer passed to it.
void treemap_destroy(treemap_t *);

/// Insert value v under key k in the tree -- if there is
/// already a value v' with key k, return v' and replace
/// v' with v in the tree.
void *treemap_insert(treemap_t *t, int k, void *v);

/// Return the value associated with key k in the tree.
/// If key k is not in the tree, return NULL.
void *treemap_lookup(treemap_t *t, int k);
```

By copy-pasting the specifications above to `treemap.c`, it will be handy to have them easily available while writing code.

For example, `treemap_insert()` will then look like the following:

```
/// Insert value v under key k in the tree -- if there is
/// already a value v' with key k, return v' and replace
/// v' with v in the tree.
void *treemap_insert(treemap_t *t, int key, void *value)
{
  /// TODO: lägg till enligt spec
  /// Remember there are instructions in the exam text
  /// for how to implement insert

  /// Hint: use a recursive help function (or double pointers)
  return NULL;
}
```

3. Run the tests and fix the first bug encountered.

Running the tests is straight-forward:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
gcc -Wall -g yourcode.c driver.c
... yada, yada, yada ...
==146== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Makefile:8: recipe for target 'memtest' failed
make: *** [memtest] Segmentation fault (core dumped)
```

Wow! – The intimidating `Segmentation fault` has honored us with its presence. But no time for whining. Use `gdb` to help finding out what went wrong:

```
<path-to-here>/uppgift1$ gdb ./a.out
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
... yada, yada, yada ...
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./a.out...done.
(gdb)
```

Type `run` to run the program again, but this time inside `gdb`:

```
(gdb) run
Starting program: /mnt/c/sz/dit/ioopm/kp-solved/kodprov-2019-01-14/hesc0353C/uppgift1/a.out
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400847 in node_size (n=0x0, acc=0x7fffffee4e4) at yourcode.c:124
124       if (n->left != NULL) node_size(n->left, acc);
(gdb)
```

We may want to see a backtrace to help orient inside what function the crash occured:

```
(gdb) bt
#0  0x0000000000400847 in node_size (n=0x0, acc=0x7fffffee4e4) at yourcode.c:124
#1  0x00000000004008d5 in treemap_size (t=0x604010) at yourcode.c:135
#2  0x0000000000400910 in test_basic () at driver.c:24
#3  0x00000000004021a6 in main () at driver.c:232
(gdb) q
```

– Aha! Function `main()` on line 232 in `driver.c` calls `test_basic()`, which on line 24 in `driver.c` calls `treemap_size()` in `yourcode.c`. Which in turn calls `node_size()` on line 135. Finally, the program crashes on line 124 inside function `node_size(node_t *n, int *acc)` with argument values being `n=0x0 (=NULL)` and `acc=0x7fffffee4e4`.

Line 124 reads `if (n->left != NULL) ...` which explains the crash. – After all, how could the compiler tell what `n->left` is, if `n=NULL`? – `NULL->left` hardly makes any sense! As a side note, Java would have given the friendlier (and much more informative) *NullPointerException*.

So the problem is a missing check that the variable `n` may not be `NULL` when arriving at line 124.
An easy fix is to copy-paste line 100 – `if (n == NULL) return;` – to line 122.

4. Run the tests again, and again, and again – `test_basic()`.

Re-running the tests now yields:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...
==172== ERROR SUMMARY: 18 errors from 18 contexts (suppressed: 0 from 0)
```

Yay. – Still 18 errors, but at least we don't get the `Segmentation fault` any more. (Yay!)

Let us adapt a test-driven approach and comment out all tests except the first one.
The last lines of `driver.c` will then look like:

```
int main(void)
{
  test_basic();
  //test_size();
  //test_keys_1();
  //test_keys_2();
  //test_has_keys();
  //test_lookup();
  puts("\nIf no errors are printed above -- all tests pass!");
}
```

Re-running the tests again yields:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...
==184== HEAP SUMMARY:
==184==     in use at exit: 8 bytes in 1 blocks
==184==   total heap usage: 2 allocs, 1 frees, 4,104 bytes allocated
==184==
==184== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==184==    at 0x4C2F988: calloc (vg_replace_malloc.c:711)
==184==    by 0x4006DD: treemap_create (yourcode.c:50)
==184==    by 0x400908: test_basic (driver.c:23)
==184==    by 0x4021AE: main (driver.c:232)
==184==
==184== LEAK SUMMARY:
==184==    definitely lost: 8 bytes in 1 blocks
==184==    indirectly lost: 0 bytes in 0 blocks
==184==      possibly lost: 0 bytes in 0 blocks
==184==    still reachable: 0 bytes in 0 blocks
==184==         suppressed: 0 bytes in 0 blocks
==184==
==184== For counts of detected and suppressed errors, rerun with: -v
==184== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

– Valgrind says memory is allocateed which is never released again. To be more precise, the function `treemap_create()` is called on line 23 of `driver.c`, and on line 50 of `yourcode.c` memory is allocated for a treemap:

```
  return calloc(1, sizeof(treemap_t));
```

– Where is this memory deallocated / released again?

To find out:

```
<path-to-here>/uppgift1$ grep -irn 'free('
driver.c:80:  free(keys);
driver.c:119:  free(keys);
```

– So the answer is: **nowhere**!

Double-check:

```
<path-to-here>/uppgift1$ grep -irn 'alloc('
yourcode.c:21:  node_t *n = calloc(1, sizeof(node_t));
yourcode.c:50:  return calloc(1, sizeof(treemap_t));
yourcode.c:111:  int *keys = calloc(treemap_size(t) + 1, sizeof(int));
```

Line 21 of `yourcode.c` is inside the function `node_new()`, which is currently not called from anywhere. (How can you tell? [1]) And the `keys` variable allocated on line 111 of `yourcode.c` inside the `treemap_keys()` function is clearly released on lines 80 and 119 of `driver.c` as shown a few lines up from here.
The function `treemap_keys()` is called twice from `driver.c`, which can be confirmed:

```
<path-to-here>/uppgift1$ grep -irn 'treemap_keys(' driver.c
72:  int *keys = treemap_keys(t);
110:  int *keys = treemap_keys(t);
```

4a. Deallocate the memory of an empty tree.

The function `treemap_destroy()` is called on line 25 of `driver.c`, clearly with the intent of freeing up all memory allocated for the tree. Line 23 of `driver.c` just creates an empty tree, so in this case there is no need to free any memory inside the tree, only the tree itself.

This is easy to do by just adding the following after line 66:

```
  free(t);
```

Doing this, and then re-running the tests yields:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...
==238== All heap blocks were freed -- no leaks are possible
==238==
==238== For counts of detected and suppressed errors, rerun with: -v
==238== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

– *Hooray!* The messages `no leaks are possible` and `0 errors from 0 contexts` are exactly what we want to see. So `test_basic()` now passes.

5. Run the tests again – `test_size()`.

Now uncomment the next test, `test_size()`:

```
int main(void)
{
  test_basic();
  test_size();
  //test_keys_1();
  //test_keys_2();
  //test_has_keys();
  //test_lookup();
  puts("\nIf no errors are printed above -- all tests pass!");
}
```

But take one baby-step at a time – start slowly by inserting *just one* element into the tree.
The test will then look as follows:

```
void test_size()
{
  treemap_t *t = treemap_create();
  treemap_insert(t, 5, "String value");
  assert_eq(treemap_size(t), 1);
  //treemap_insert(t, 2, "String value");
  //assert_eq(treemap_size(t), 2);
  //treemap_insert(t, 8, "String value");
  //assert_eq(treemap_size(t), 3);
  //treemap_insert(t, 6, "String value");
  //assert_eq(treemap_size(t), 4);
  //treemap_insert(t, 4, "String value");
  //assert_eq(treemap_size(t), 5);
  //treemap_insert(t, 1, "String value");
  //assert_eq(treemap_size(t), 6);
  //treemap_insert(t, 32, "String value");
  //assert_eq(treemap_size(t), 7);
  treemap_destroy(t);
}
```

Re-running the tests yields:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...
Assertion failed: expected 1, got 0                              | driver.c:32 (test_size)
... yada, yada, yada ...
```

– Well. What did you expect? We have not yet implemented insertions of *any* elements into the tree. (!)

5a. Insert *just one* element into the tree.

Line 8 of `yourcode.c` tells that a treemap has a node called `root`:

```
typedef struct node node_t;

struct treemap
{
  node_t *root;
};
```

Where a `node` is defined in lines 11-17 of `yourcode.c`:

```
struct node
{
  int key;
  void *value;
  node_t *left;
  node_t *right;
};
```

As already touched upon, a helper function for creating and allocating memory for a new `node` is defined in lines 19-25 of `yourcode.c`:

```
static node_t *node_new(int key, void *value)
{
  node_t *n = calloc(1, sizeof(node_t));  // Line 21 of yourcode.c
  n->key = key;
  n->value = value;
  return n;
}
```

So to create a tree that contains *just one* element, let the root of the tree be the node that holds the element:

```
/// Insert value v under key k in the tree -- if there is
/// already a value v' with key k, return v' and replace
/// v' with v in the tree.
void *treemap_insert(treemap_t *t, int key, void *value)
{
  /// TODO: lägg till enligt spec
  /// Remember there are instructions in the exam text
  /// for how to implement insert
  /// Hint: use a recursive help function (or double pointers)
  t->root = node_new(key, value);
  return NULL;
}
```

It may be noted that the `left` and `right` attributes / pointers of the single `root node` are both `NULL`. This is because `calloc` rather than `malloc` is used to allocate memory on line 21 of `yourcode.c`.

Re-running the tests again yields:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...

If no errors are printed above -- all tests pass!
==274====274== HEAP SUMMARY:
==274==     in use at exit: 32 bytes in 1 blocks
==274==   total heap usage: 4 allocs, 3 frees, 4,144 bytes allocated
==274==
==274== 32 bytes in 1 blocks are definitely lost in loss record 1 of 1
==274==    at 0x4C2F988: calloc (vg_replace_malloc.c:711)
==274==    by 0x400683: node_new (yourcode.c:21)
==274==    by 0x4006C6: treemap_insert (yourcode.c:36)
==274==    by 0x4009BC: test_size (driver.c:31)
==274==    by 0x401FBF: main (driver.c:233)
==274==
==274== LEAK SUMMARY:
==274==    definitely lost: 32 bytes in 1 blocks
==274==    indirectly lost: 0 bytes in 0 blocks
==274==      possibly lost: 0 bytes in 0 blocks
==274==    still reachable: 0 bytes in 0 blocks
==274==         suppressed: 0 bytes in 0 blocks
==274==
==274== For counts of detected and suppressed errors, rerun with: -v
==274== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Good. – "All tests pass!" (Which refers only to the tests that are currently turned on.) The leaking memory should come as no surprise: recall from the previous section how we deliberately constructed the function `treemap_destroy()` to free up memory for the tree, *but not for any of its elements*.

This needs to change now. Free up the root as well – `free(t->root);` – and rerun the tests to see if this remedies the memory leakage. (It should!)

5b. Deallocate the memory of *all* elements in the tree.

Now try a recursive approach to free up the memory of **all** nodes in the tree.

It may come in handy with some inspiration from `treemap_size()` which recursively calls `node_size()`:

```
/// Helper function for treemap_size
static void node_size(node_t *n, int *acc)
{
  /// Postorder traversal
  if (n == NULL) return;
  /// Recurse into left subtree
  if (n->left != NULL) node_size(n->left, acc);
  /// Recurse into right subtree
  if (n->right != NULL) node_size(n->right, acc);

  /// Increment the accumulator
  *acc += 1;
}
```

Simply copy-paste the body of `node_size()` into `node_destroy()`, change the recursive calls to call `node_destroy()` rather than `node_size()`, remove the `acc` argument when calling recursively, and adjust to code to free memory of the current node instead of incrementing an accumulator:

```
/// Helper function for treemap_destroy
static void node_destroy(node_t *n)
{
  /// TODO: frivillig hjälpfunktion
  /// Hint -- use this as a helper function to treemap_destroy
  /// Implement it as a recursive function that first visits
  /// the subtrees and then destroys n
  /// Postorder traversal
  if (n == NULL) return;
  /// Recurse into left subtree
  if (n->left != NULL) node_destroy(n->left);
  /// Recurse into right subtree
  if (n->right != NULL) node_destroy(n->right);
  /// Free the memory of the current node n:
  free(n);
}
```

Oh! – And don't forget to call `node_destroy()` from within `treemap_destroy()`:

```
/// Removes the treemap from memory, but does not call
/// free on any value pointer passed to it.
void treemap_destroy(treemap_t *t)
{
  /// TODO: lägg till enligt spec
  //free(t->root);
  node_destroy(t->root);
  free(t);
}
```

Again rerun the tests to see:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...

If no errors are printed above -- all tests pass!
==330==
==330== HEAP SUMMARY:
==330==     in use at exit: 0 bytes in 0 blocks
==330==   total heap usage: 4 allocs, 4 frees, 4,144 bytes allocated
==330==
==330== All heap blocks were freed -- no leaks are possible
==330==
==330== For counts of detected and suppressed errors, rerun with: -v
==330== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

– Yay, `all tests pass!` + `no leaks are possible` + `0 errors from 0 contexts` = sweet!

Unfortunately, this proves only that `treemap_destroy()` works for *one single* element. If shit happens and the program fails later when inserting more elements, we will have to keep an open mind as to whether the error depends on `treemap_destroy()` or on `treemap_insert()` (or *both*).

5c. Insert *more than one* element into the tree.

Although the implementation of `treemap_destroy()` has not yet been confirmed to be correct for more than one element, it is reasonable to apply a similar recursive approach for `treemap_insert()`.

One possibility is to copy-paste `node_destroy()` and rename it into a completely new recursive function `node_insert()`, keeping just the skeleton, but adding the key-value pair as parameters, and having it return a node:

```
/// Helper function for treemap_insert:
static node_t *node_insert(node_t *n, int key, void *value)
{
  /// Hint -- use this as a helper function for treemap_insert().
  /// Implement it as a recursive function that either:
  /// 1) inserts a new node with the demanded key and value pointer, or
  /// 2) if the key already exists: replaces the value pointer, or
  /// 3) continues the search in the subtrees.
  if (n == NULL) ; // <-- Insert a new node here.
  else ;  // Check if key exists, if not continue searching the subtrees.
  return NULL;
}
```

Even with the strange looking code above, the program still compiles and runs.

What next? How do we check the key and then traverse the subtrees if the key is not the right one?

The observant reader will notice a hint in the code given in `yourcode.c` which is currently on line 93:

```
/// Hint: look at this function for inspiration
bool treemap_has_key(treemap_t *t, int key)
{
  node_t *n = t->root;
  while (n)
    {
      if (n->key == key) return true;  // currently line 100 of yourcode.c

      if (n->key > key)
        {
          n = n->left;
        }
      else
        {
          n = n->right;
        }                               // currently line 109 of yourcode.c
    }

  /// If we came here without finding it, it can't exist
  return false;
}
```

Of particular interest to a recursive approach are the `if` and `else` clauses, currently lines 100-109.

Copy-paste those lines into the newly created function `node_insert()`, and adjust the code slightly:

```
/// Helper function for treemap_insert:
static node_t *node_insert(node_t *n, int key, void *value)
{
  /// Hint -- use this as a helper function for treemap_insert().
  /// Implement it as a recursive function that either:
  /// 1) inserts a new node with the demanded key and value pointer, or
  /// 2) if the key already exists: replaces the value pointer, or
  /// 3) continues the search in the subtrees.
  if (n == NULL)
    {                     // If no such key, insert a new node here:
      n = node_new(key, value);
    }
  else if (n->key == key) // Else check if the key already exists ...
    {
      n->value = value;   // ... if it does, replace the value pointer ...
    }
  else if (n->key > key)  // ... if not, try inserting into the left subtree ...
    {
      n->left = node_insert(n->left, key, value);
    }
  else if (n->key < key)  // ... -- OR -- into the right subtree if key is BIG.
    {
      n->right = node_insert(n->right, key, value);
    }
  return n;
}
```

Again, remember to update `treemap_insert()` so that it calls the recursive function `node_insert()`:

```
/// Insert value v under key k in the tree -- if there is
/// already a value v' with key k, return v' and replace
/// v' with v in the tree.
void *treemap_insert(treemap_t *t, int key, void *value)
{
  /// TODO: lägg till enligt spec
  /// Remember there are instructions in the exam text
  /// for how to implement insert
  /// Hint: use a recursive help function (or double pointers)
  //t->root = node_new(key, value);
  t->root = node_insert(t->root, key, value);
  return NULL;
}
```

Next, rerun the tests to see `all tests pass!` + `no leaks are possible` + `0 errors from 0 contexts`.

OK, good. But remember – there is still only *one* element inserted in `test_size()`. To see if it works with more elements, we need to uncomment lines 33-44 of `driver.c`, like so:

```
void test_size()
{
  treemap_t *t = treemap_create();
  treemap_insert(t, 5, "String value");
  assert_eq(treemap_size(t), 1);
  treemap_insert(t, 2, "String value");
  assert_eq(treemap_size(t), 2);
  treemap_insert(t, 8, "String value");
  assert_eq(treemap_size(t), 3);
  treemap_insert(t, 6, "String value");
  assert_eq(treemap_size(t), 4);
  treemap_insert(t, 4, "String value");
  assert_eq(treemap_size(t), 5);
  treemap_insert(t, 1, "String value");
  assert_eq(treemap_size(t), 6);
  treemap_insert(t, 32, "String value");
  assert_eq(treemap_size(t), 7);
  treemap_destroy(t);
}
```

– Yay, still getting `all tests pass!` + `no leaks are possible` + `0 errors from 0 contexts` = nice!

6. Run the tests again – `test_keys_1()`, `test_keys_2()`, `test_has_keys()`.

Now uncomment the next three tests, `test_keys_1()`, `test_keys_2()`, `test_has_keys()`:

```
int main(void)
{
  test_basic();
  test_size();
  test_keys_1();
  test_keys_2();
  test_has_keys();
  //test_lookup();
  puts("\nIf no errors are printed above -- all tests pass!");
}
```

– Still getting `all tests pass!` + `no leaks are possible` + `0 errors from 0 contexts` = no problem!

7. Run the tests again – `test_lookup()`.

We have not yet implemented `treemap_lookup()` so there is no way we could expect that test to pass.

The function `treemap_lookup()` currently looks like:

```
/// Return the value associated with key k in the tree.
/// If key k is not in the tree, return NULL.
void *treemap_lookup(treemap_t *t, int key)
{
  /// TODO: lägg till enligt spec
  return NULL;
}
```

The `treemap_lookup()` function is actually rather similar to `treemap_size()` along with its helper function:

```
/// Helper function for treemap_size
static void node_size(node_t *n, int *acc)
{
  /// Postorder traversal
  if (n == NULL) return;
  /// Recurse into left subtree
  if (n->left != NULL) node_size(n->left, acc);
  /// Recurse into right subtree
  if (n->right != NULL) node_size(n->right, acc);

  /// Increment the accumulator
  *acc += 1;
}

int treemap_size(treemap_t *t)
{
  int result = 0;
  node_size(t->root, &result);
  return result;
}
```

But for the lookup, the helper function should take not the address of an accumulator, but a key and the *address* of a value pointer as parameters. There is already a function that does almost that, namely the function `node_insert()`.

So copy-paste the function `node_insert()` into a new function `node_lookup()` and adjust the code slightly. Also update `treemap_lookup()` to work in a similar way as `treemap_size()`:

```
/// Helper function for treemap_lookup:
static node_t *node_lookup(node_t *n, int key, void** adr_of_value)
{
  /// Hint -- use this as a helper function for treemap_lookup().
  /// Implement it as a recursive function that:
  /// 1) if the key is found: updates the value pointer, or
  /// 2) continues the search in the subtrees.
  if (n != NULL)
    {
      if (n->key == key)            // Check if the key is in this node,
        {
          *adr_of_value = n->value; // if it is, update the value pointer,
        }
      else if (n->key > key)        // if not, try searching the left subtree,
        {
          n->left = node_lookup(n->left, key, adr_of_value);
        }
      else if (n->key < key)        // or the right subtree.
        {
          n->right = node_lookup(n->right, key, adr_of_value);
        }
    }
  return n;
}

/// Return the value associated with key k in the tree.
/// If key k is not in the tree, return NULL.
void *treemap_lookup(treemap_t *t, int key)
{
  /// TODO: lägg till enligt spec
  void* result = NULL;
  t->root = node_lookup(t->root, key, &result);
  return result;
}
```

The time has come to uncomment the last test, `test_lookup()`:

```
int main(void)
{
  test_basic();
  test_size();
  test_keys_1();
  test_keys_2();
  test_has_keys();
  test_lookup();
  puts("\nIf no errors are printed above -- all tests pass!");
}
```

Finally run the tests again, this time literally *all* of them:

```
<path-to-here>/uppgift1$ make clean && make memtest
rm -f *.o a.out
... yada, yada, yada ...

If no errors are printed above -- all tests pass!
==139==
==139== HEAP SUMMARY:
==139==     in use at exit: 0 bytes in 0 blocks
==139==   total heap usage: 44 allocs, 44 frees, 5,328 bytes allocated
==139==
==139== All heap blocks were freed -- no leaks are possible
==139==
==139== For counts of detected and suppressed errors, rerun with: -v
==139== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

– Getting `all tests pass!` + `no leaks are possible` + `0 errors from 0 contexts` = everything is OK!

That's it!

8. Ignoring specifications – `treemap_insert()`.

Wait a minute!

What about the specifications for `treemap_insert()` given by lines 15-17 of `treemap.h`?
Those that say: "if there is already a value v' with key k, return v' and replace v' with v in the tree"?

```
/// Insert value v under key k in the tree -- if there is
/// already a value v' with key k, return v' and replace
/// v' with v in the tree.
void *treemap_insert(treemap_t *t, int k, void *v);
```

Is this really implemented?

– No. I ignored it. For two reasons. One reason is that the specification does not explicitly tell what should be returned if there is **not** already a value (pointer) with key k. (A natural assumption would be to return `NULL` in that case.) Also, there are no tests to check if this functionality has been implemented. And thirdly, this is easy to implement by using the same technique as for `treemap_lookup()` above. If you really care, I do propose a solution at http://user.it.uu.se/~hesc0353/ioopm/kodprov-2019-12-10.zip.

Cheers! /Henke, henke.uu.se@inbox.lv

--

Uppsala, 2019-12-31.

---

1. `grep -irn 'node_new(' yourcode.c` ↩