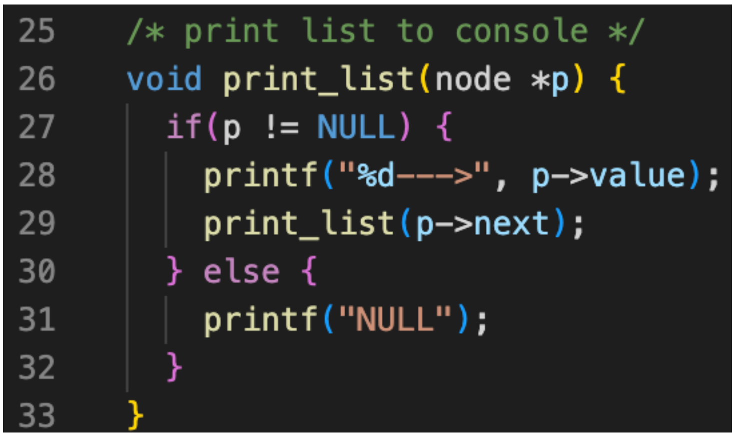
Programming for computerteknologi
Hand-in Assignment Exercises

Week 10: Passing functions as arguments to other functions

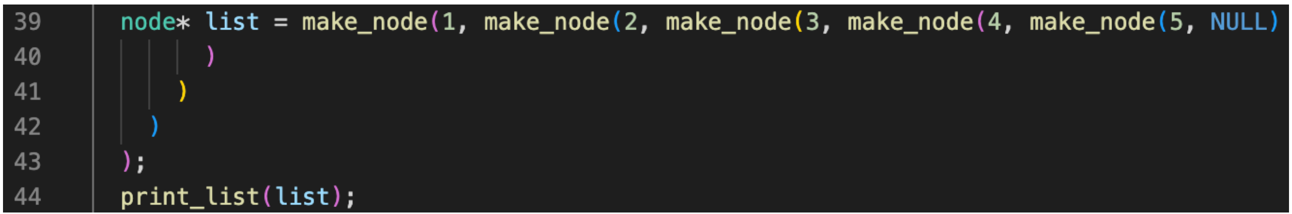
Exercise 1)

We have been given a task to implement a recursive function that prints out a linked list of integers. When implemented we get the following results.

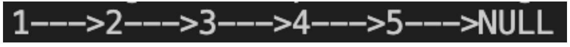


```
/* print list to console */
void print_list(node *p) {
    if(p != NULL) {
        printf("%d--->", p->value);
        print_list(p->next);
    } else {
        printf("NULL");
    }
}
```

When running the function, we get the following output:



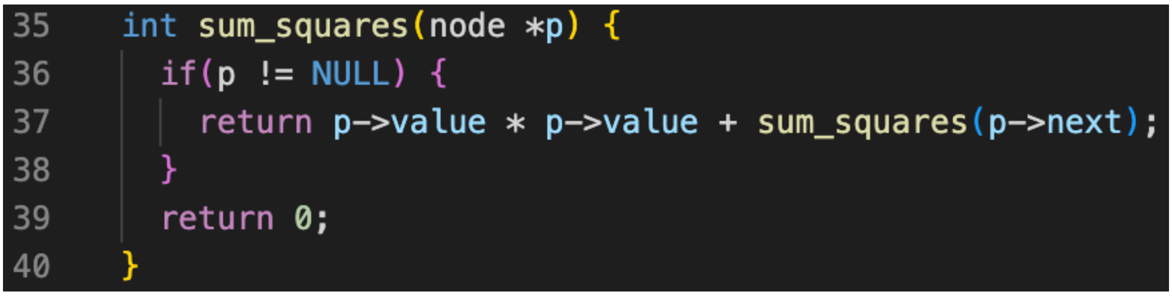
```
node* list = make_node(1, make_node(2, make_node(3, make_node(4, make_node(5, NULL)
            )
          )
        )
);
print_list(list);
```



```
1--->2--->3--->4--->5--->NULL
```

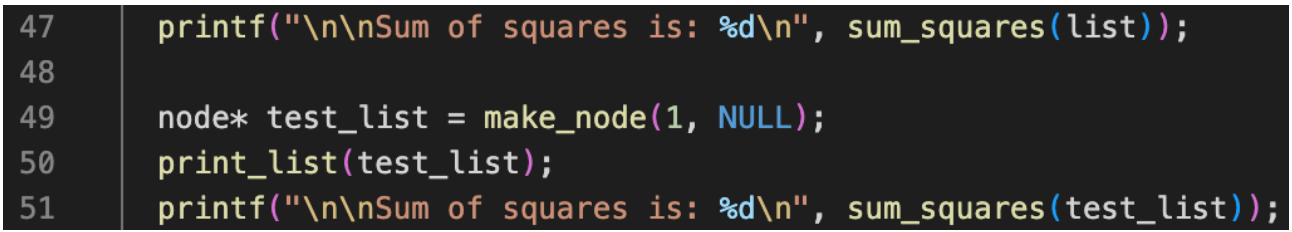
Exercise 2)

We have been given a task to write and implement a recursive function that accepts a linked list of integers and returns the sum of the squares of the integers in the list. The code will look like the following when implemented:

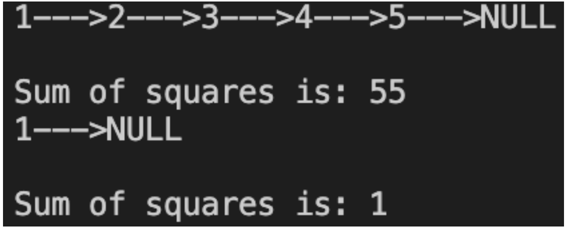


```
int sum_squares(node *p) {
    if(p != NULL) {
        return p->value * p->value + sum_squares(p->next);
    }
    return 0;
}
```

When running the function, we get the following output:



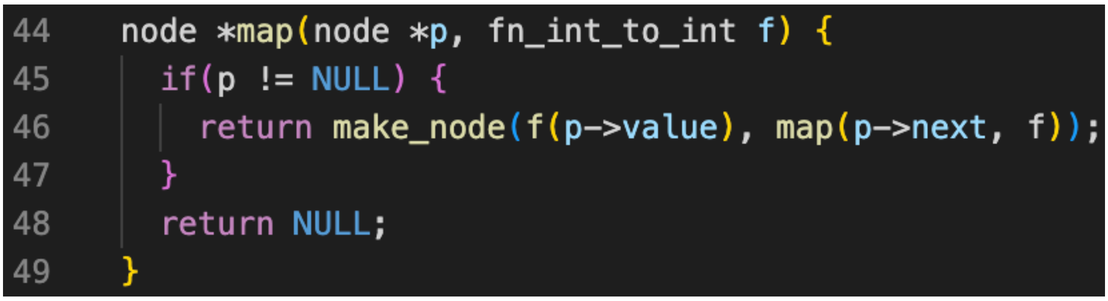
```
    printf("\n\nSum of squares is: %d\n", sum_squares(list));

    node* test_list = make_node(1, NULL);
    print_list(test_list);
    printf("\n\nSum of squares is: %d\n", sum_squares(test_list));
```



```
1--->2--->3--->4--->5--->NULL

Sum of squares is: 55
1--->NULL

Sum of squares is: 1
```

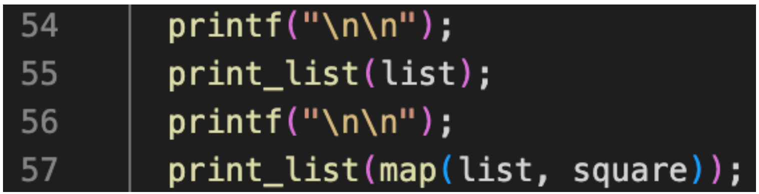
Exercise 3)

We have given the task to implement a map function that takes a list and returns a new list with each of the elements being squared from the original list. When implementing the function, it look like the following:

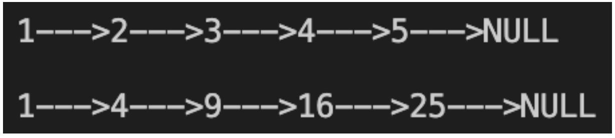


```
node *map(node *p, fn_int_to_int f) {
  if(p != NULL) {
    return make_node(f(p->value), map(p->next, f));
  }
  return NULL;
}
```

When running the function, we get the following results:



```
    printf("\n\n");
    print_list(list);
    printf("\n\n");
    print_list(map(list, square));
```



```
1--->2--->3--->4--->5--->NULL

1--->4--->9--->16--->25--->NULL
```

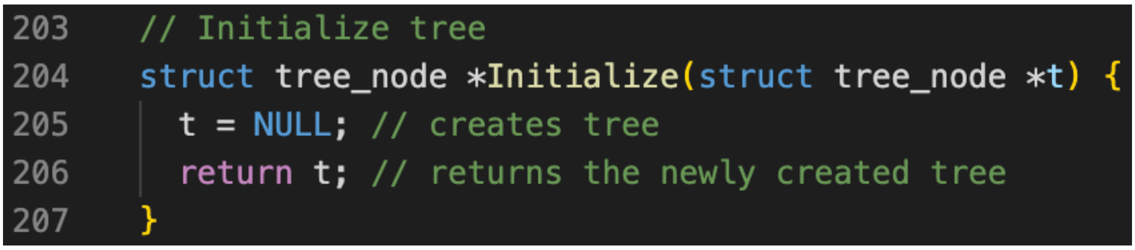
Exercise 4)

We have been given the task to implement a binary tree which includes the following functions:

- Initialize(t) - Creates a tree.
- Insert(x, t) - Inserts a node in the tree.
- Remove(x, t) - Removes a node from the tree.
- Contains(x, t) - Checks if the tree contains the given value.
- Empty(t) - Checks whether a tree is empty or not.
- Full(t) - Isn't necessary since we won't be able to fill up the tree.

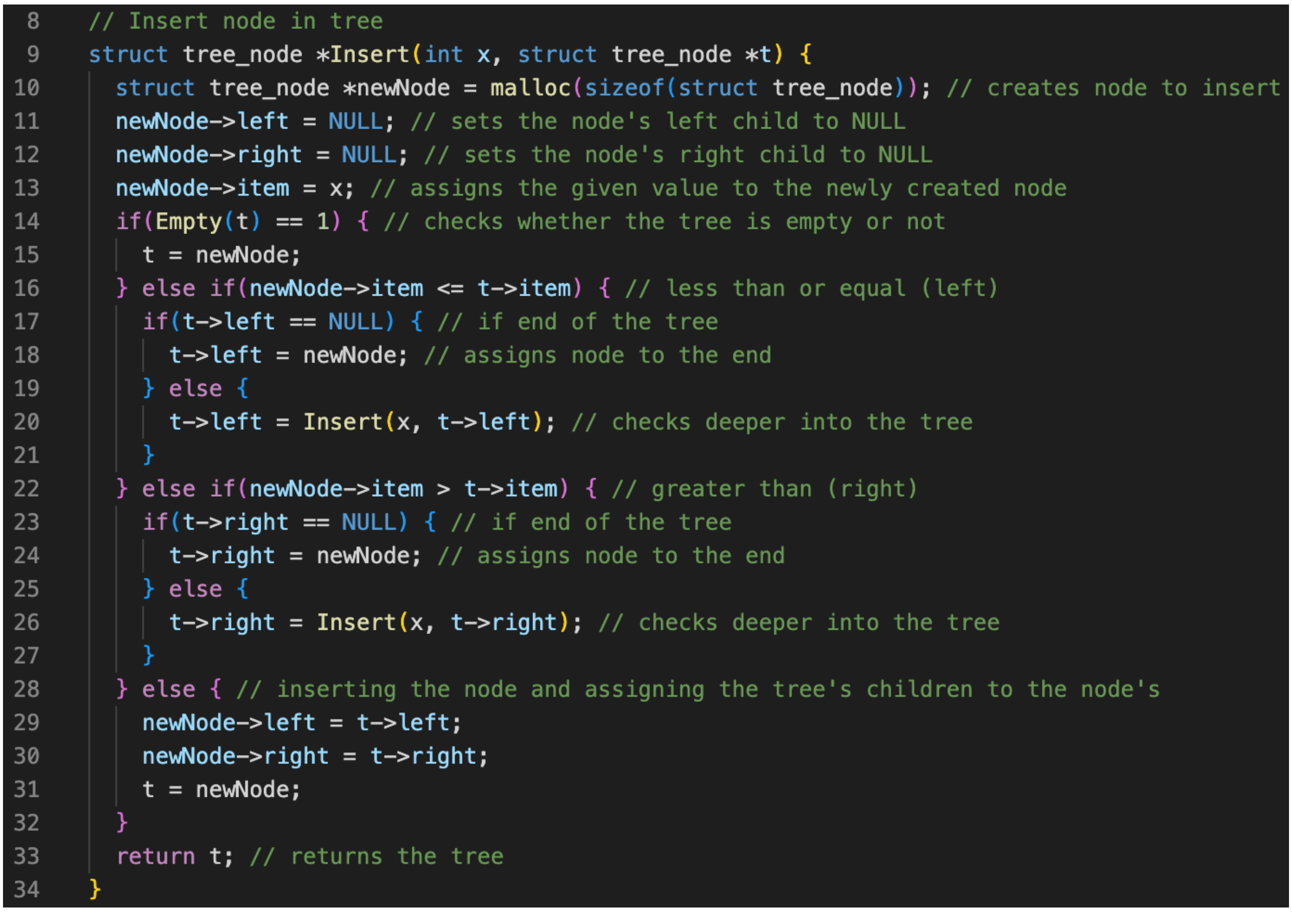
When implementing the functions, it will look like the following:

Initialize function:



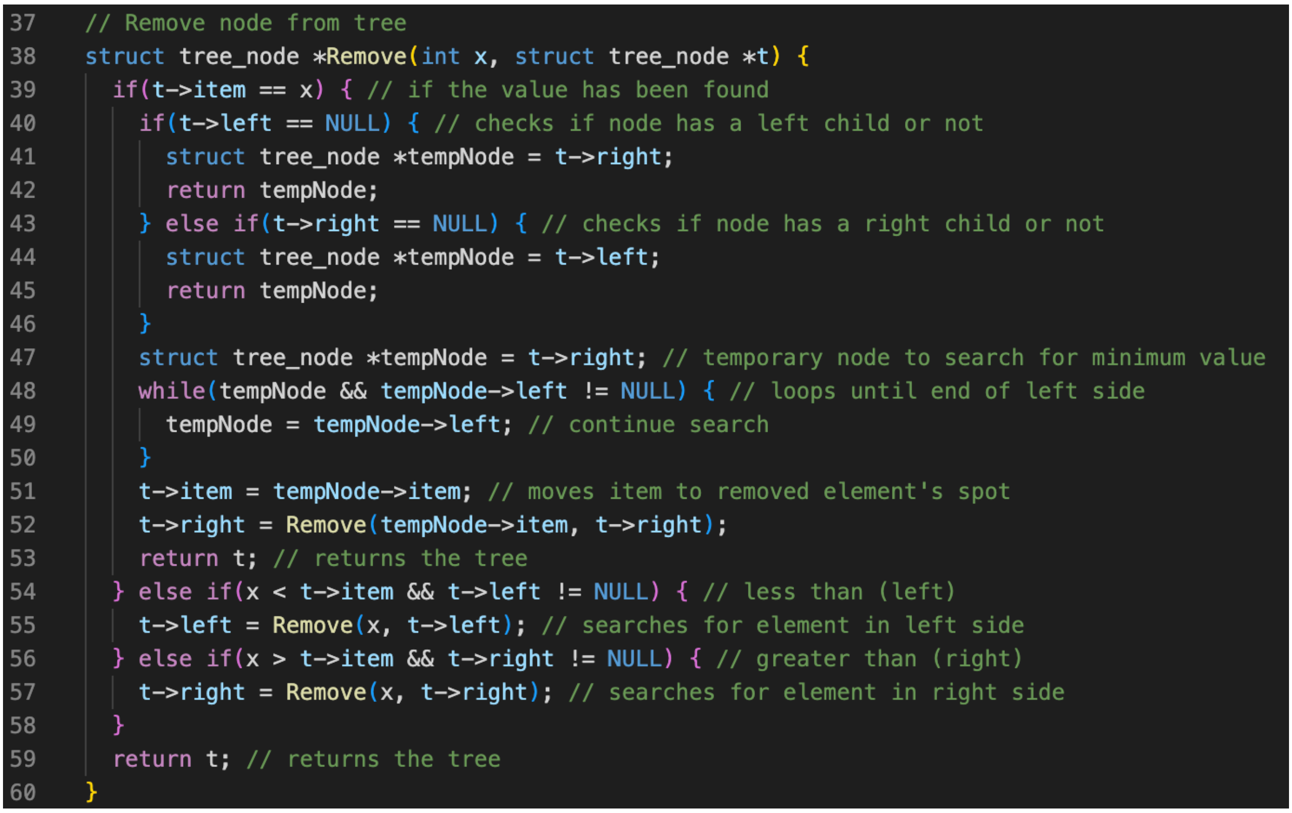
```c
// Initialize tree
struct tree_node *Initialize(struct tree_node *t) {
  t = NULL; // creates tree
  return t; // returns the newly created tree
}
```

Insert function:



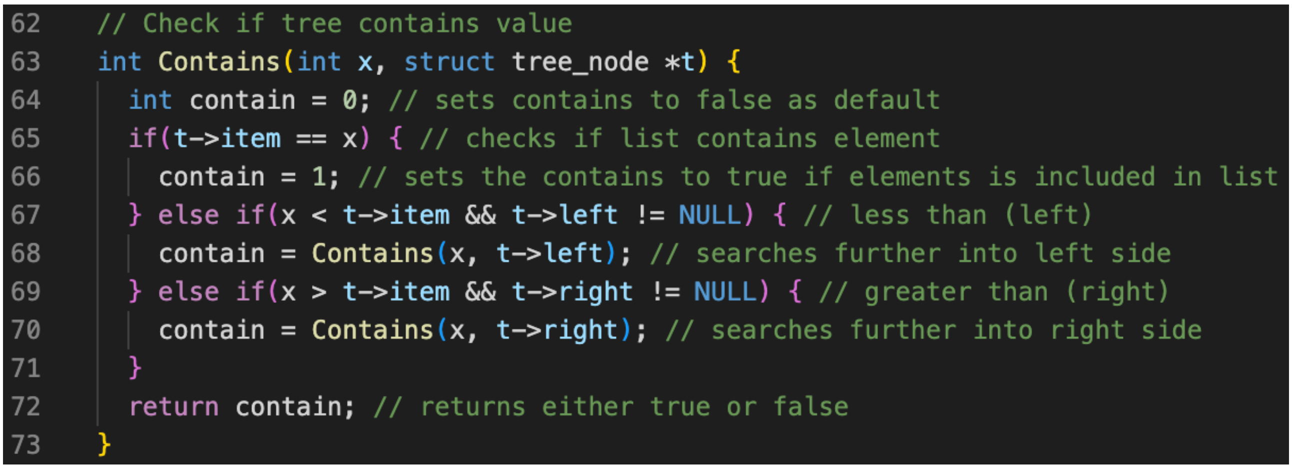
```c
// Insert node in tree
struct tree_node *Insert(int x, struct tree_node *t) {
  struct tree_node *newNode = malloc(sizeof(struct tree_node)); // creates node to insert
  newNode->left = NULL; // sets the node's left child to NULL
  newNode->right = NULL; // sets the node's right child to NULL
  newNode->item = x; // assigns the given value to the newly created node
  if(Empty(t) == 1) { // checks whether the tree is empty or not
    t = newNode;
  } else if(newNode->item <= t->item) { // less than or equal (left)
    if(t->left == NULL) { // if end of the tree
      t->left = newNode; // assigns node to the end
    } else {
      t->left = Insert(x, t->left); // checks deeper into the tree
    }
  } else if(newNode->item > t->item) { // greater than (right)
    if(t->right == NULL) { // if end of the tree
      t->right = newNode; // assigns node to the end
    } else {
      t->right = Insert(x, t->right); // checks deeper into the tree
    }
  } else { // inserting the node and assigning the tree's children to the node's
    newNode->left = t->left;
    newNode->right = t->right;
    t = newNode;
  }
  return t; // returns the tree
}
```

Remove function:



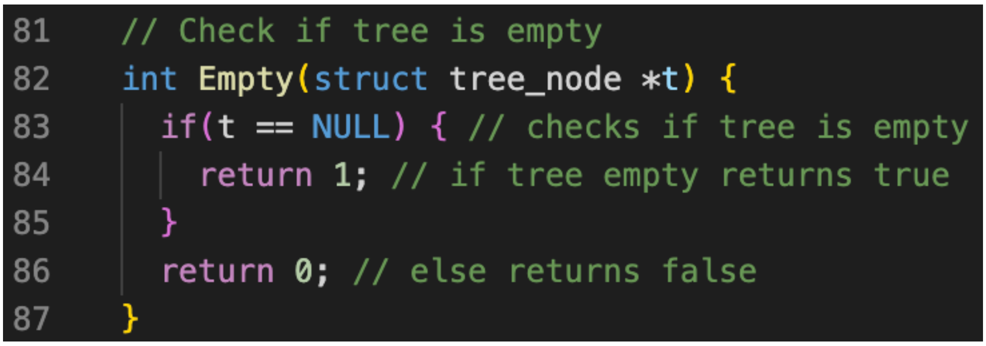
```c
// Remove node from tree
struct tree_node *Remove(int x, struct tree_node *t) {
  if(t->item == x) { // if the value has been found
    if(t->left == NULL) { // checks if node has a left child or not
      struct tree_node *tempNode = t->right;
      return tempNode;
    } else if(t->right == NULL) { // checks if node has a right child or not
      struct tree_node *tempNode = t->left;
      return tempNode;
    }
    struct tree_node *tempNode = t->right; // temporary node to search for minimum value
    while(tempNode && tempNode->left != NULL) { // loops until end of left side
      tempNode = tempNode->left; // continue search
    }
    t->item = tempNode->item; // moves item to removed element's spot
    t->right = Remove(tempNode->item, t->right);
    return t; // returns the tree
  } else if(x < t->item && t->left != NULL) { // less than (left)
    t->left = Remove(x, t->left); // searches for element in left side
  } else if(x > t->item && t->right != NULL) { // greater than (right)
    t->right = Remove(x, t->right); // searches for element in right side
  }
  return t; // returns the tree
}
```

Contains function:



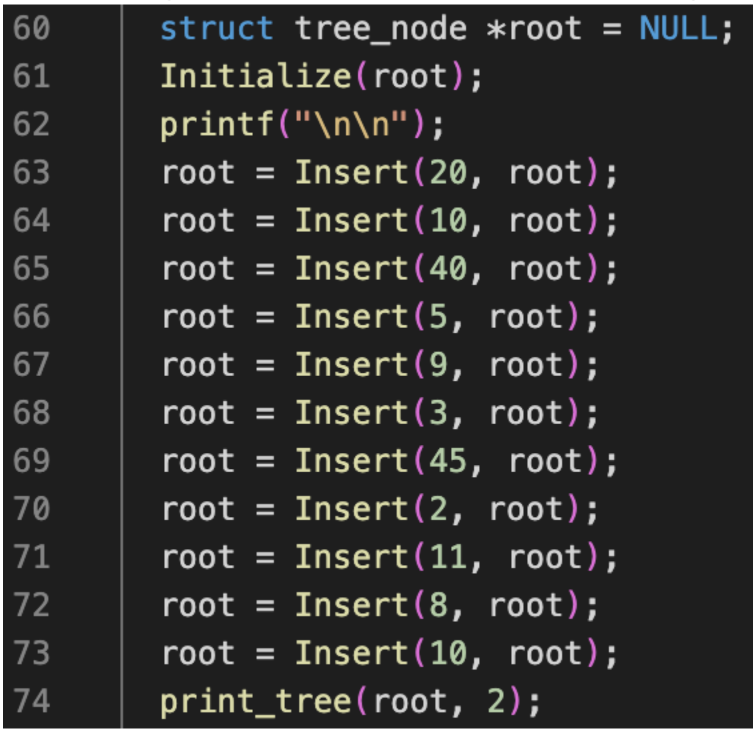
```c
// Check if tree contains value
int Contains(int x, struct tree_node *t) {
  int contain = 0; // sets contains to false as default
  if(t->item == x) { // checks if list contains element
    contain = 1; // sets the contains to true if elements is included in list
  } else if(x < t->item && t->left != NULL) { // less than (left)
    contain = Contains(x, t->left); // searches further into left side
  } else if(x > t->item && t->right != NULL) { // greater than (right)
    contain = Contains(x, t->right); // searches further into right side
  }
  return contain; // returns either true or false
}
```

Empty function:

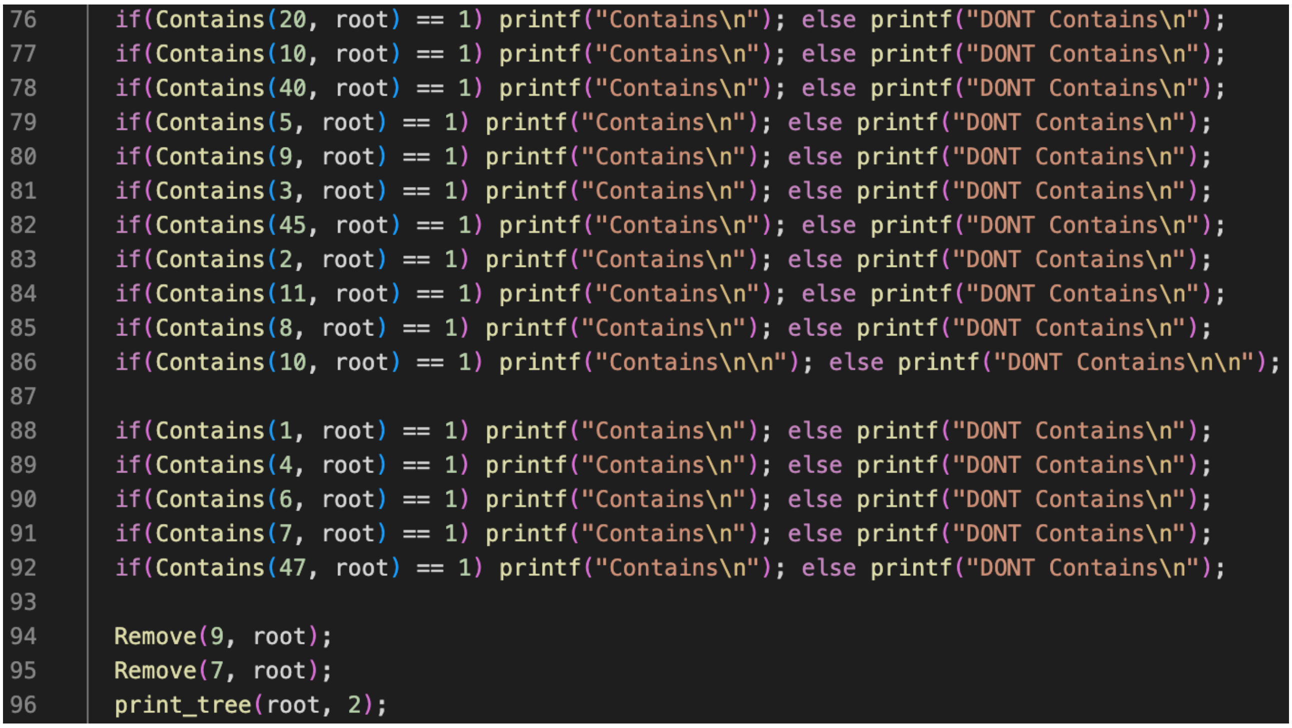


```c
// Check if tree is empty
int Empty(struct tree_node *t) {
    if(t == NULL) { // checks if tree is empty
        return 1; // if tree empty returns true
    }
    return 0; // else returns false
}
```

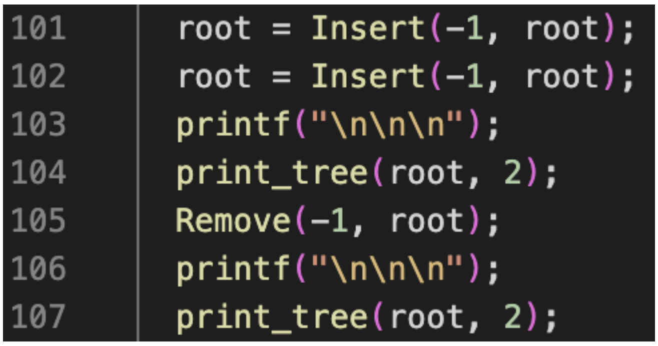
When testing the functions, I have done the following inside main:



```c
    struct tree_node *root = NULL;
    Initialize(root);
    printf("\n\n");
    root = Insert(20, root);
    root = Insert(10, root);
    root = Insert(40, root);
    root = Insert(5, root);
    root = Insert(9, root);
    root = Insert(3, root);
    root = Insert(45, root);
    root = Insert(2, root);
    root = Insert(11, root);
    root = Insert(8, root);
    root = Insert(10, root);
    print_tree(root, 2);
```

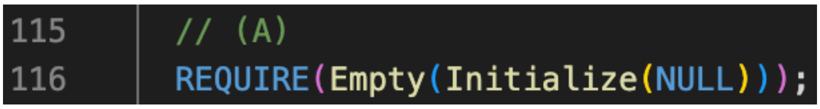


```
    if(Contains(20, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(10, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(40, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(5, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(9, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(3, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(45, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(2, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(11, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(8, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(10, root) == 1) printf("Contains\n\n"); else printf("DONT Contains\n\n");

    if(Contains(1, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(4, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(6, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(7, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
    if(Contains(47, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");

    Remove(9, root);
    Remove(7, root);
    print_tree(root, 2);
```

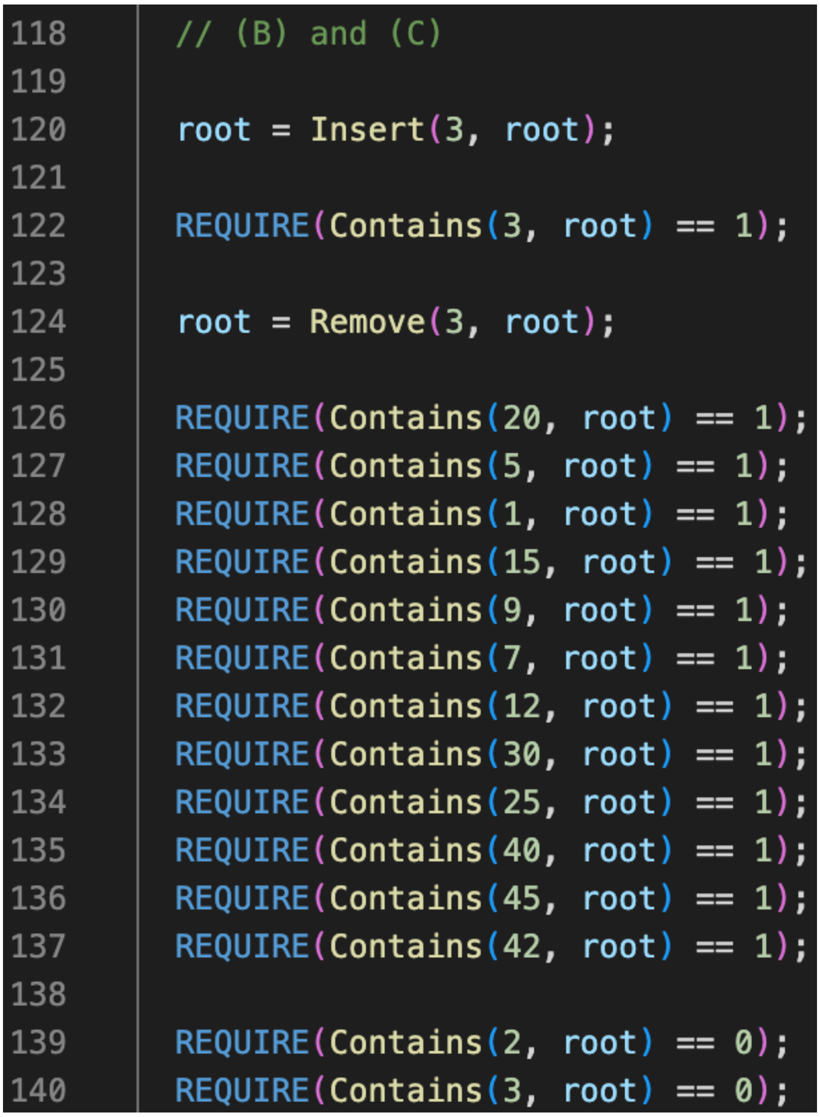


```
    root = Insert(-1, root);
    root = Insert(-1, root);
    printf("\n\n\n");
    print_tree(root, 2);
    Remove(-1, root);
    printf("\n\n\n");
    print_tree(root, 2);
```

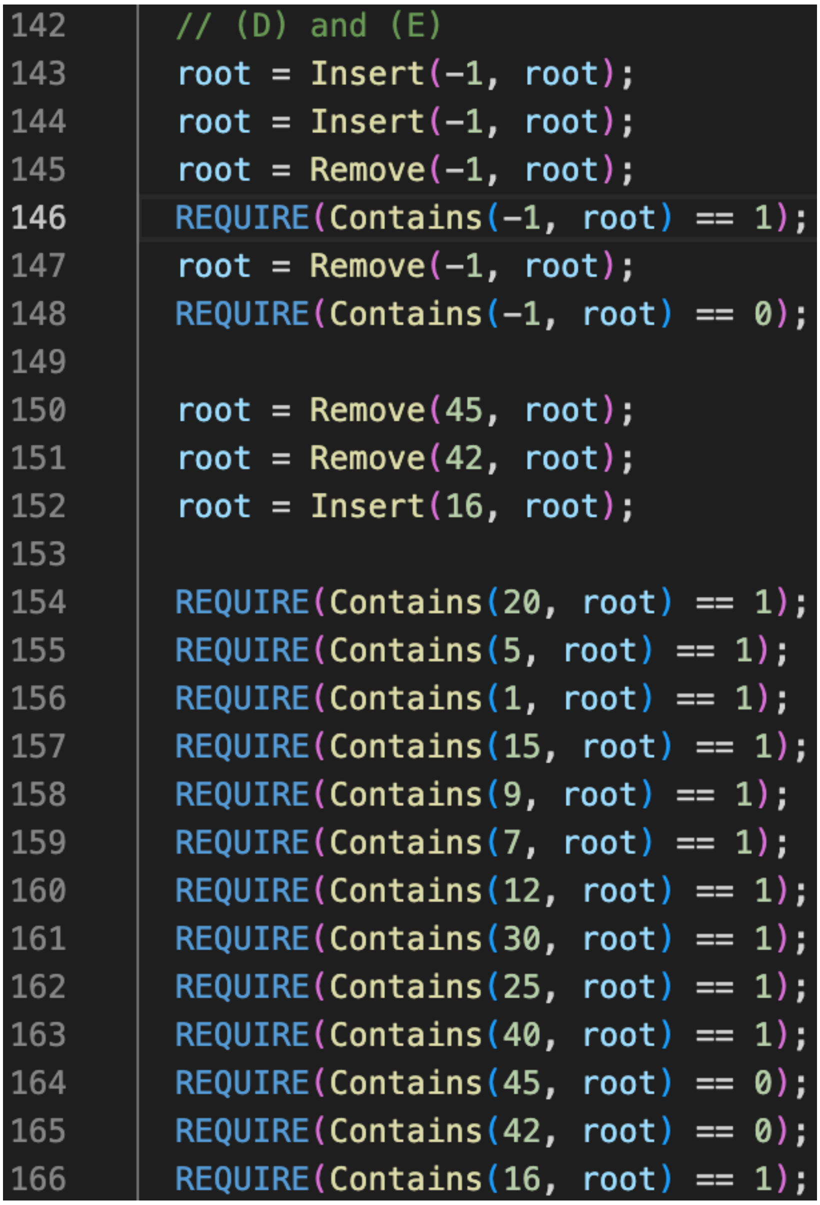
For more specific test cases, we can take a look at the given test code “tests.cpp”.



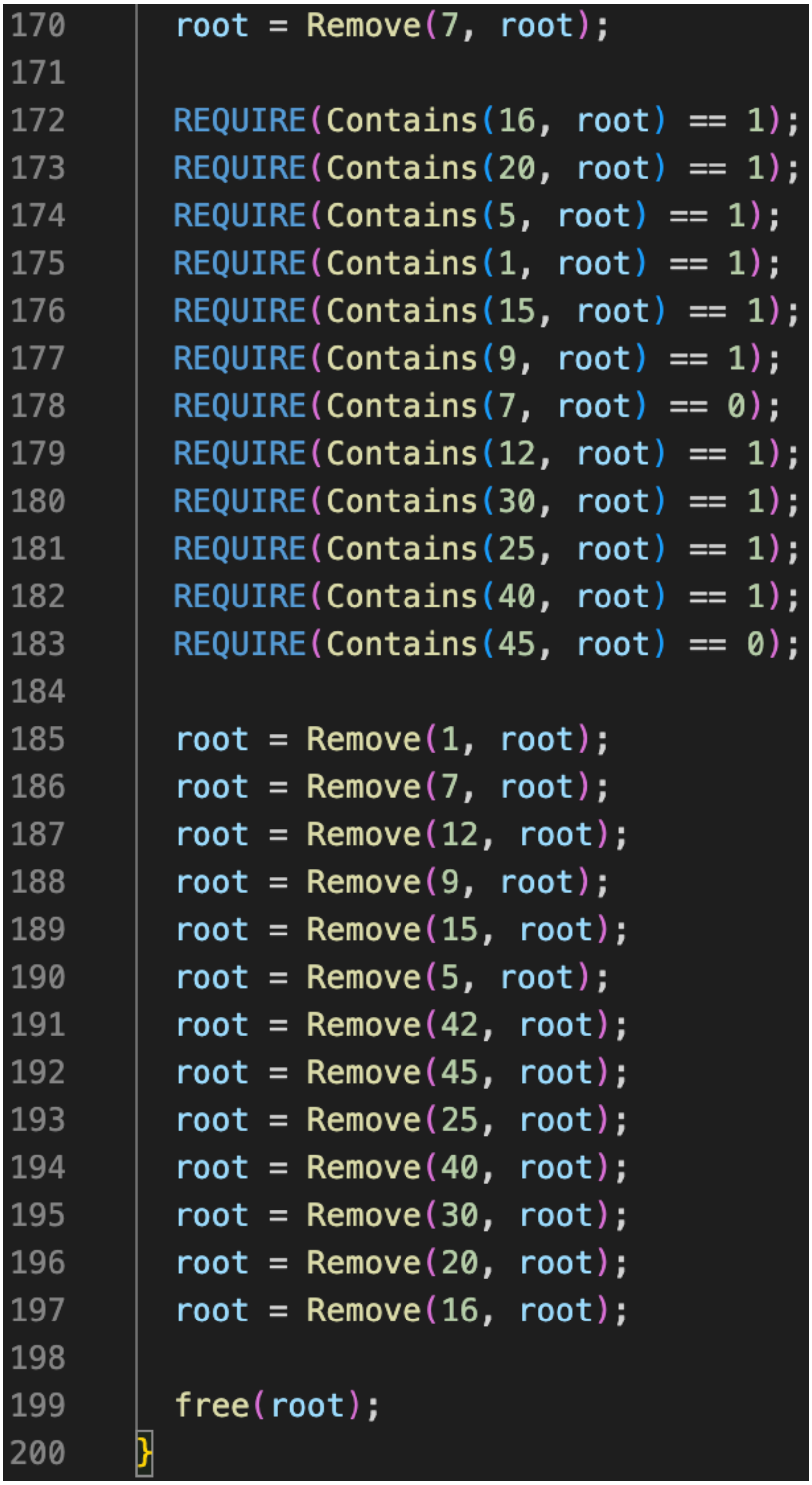
```
    // (A)
    REQUIRE(Empty(Initialize(NULL)));
```



```
// (B) and (C)

root = Insert(3, root);

REQUIRE(Contains(3, root) == 1);

root = Remove(3, root);

REQUIRE(Contains(20, root) == 1);
REQUIRE(Contains(5, root) == 1);
REQUIRE(Contains(1, root) == 1);
REQUIRE(Contains(15, root) == 1);
REQUIRE(Contains(9, root) == 1);
REQUIRE(Contains(7, root) == 1);
REQUIRE(Contains(12, root) == 1);
REQUIRE(Contains(30, root) == 1);
REQUIRE(Contains(25, root) == 1);
REQUIRE(Contains(40, root) == 1);
REQUIRE(Contains(45, root) == 1);
REQUIRE(Contains(42, root) == 1);

REQUIRE(Contains(2, root) == 0);
REQUIRE(Contains(3, root) == 0);
```

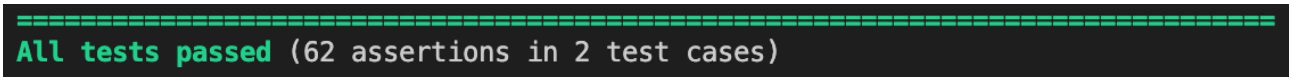


```
    // (D) and (E)
    root = Insert(-1, root);
    root = Insert(-1, root);
    root = Remove(-1, root);
    REQUIRE(Contains(-1, root) == 1);
    root = Remove(-1, root);
    REQUIRE(Contains(-1, root) == 0);

    root = Remove(45, root);
    root = Remove(42, root);
    root = Insert(16, root);

    REQUIRE(Contains(20, root) == 1);
    REQUIRE(Contains(5, root) == 1);
    REQUIRE(Contains(1, root) == 1);
    REQUIRE(Contains(15, root) == 1);
    REQUIRE(Contains(9, root) == 1);
    REQUIRE(Contains(7, root) == 1);
    REQUIRE(Contains(12, root) == 1);
    REQUIRE(Contains(30, root) == 1);
    REQUIRE(Contains(25, root) == 1);
    REQUIRE(Contains(40, root) == 1);
    REQUIRE(Contains(45, root) == 0);
    REQUIRE(Contains(42, root) == 0);
    REQUIRE(Contains(16, root) == 1);
```



```
    root = Remove(7, root);

    REQUIRE(Contains(16, root) == 1);
    REQUIRE(Contains(20, root) == 1);
    REQUIRE(Contains(5, root) == 1);
    REQUIRE(Contains(1, root) == 1);
    REQUIRE(Contains(15, root) == 1);
    REQUIRE(Contains(9, root) == 1);
    REQUIRE(Contains(7, root) == 0);
    REQUIRE(Contains(12, root) == 1);
    REQUIRE(Contains(30, root) == 1);
    REQUIRE(Contains(25, root) == 1);
    REQUIRE(Contains(40, root) == 1);
    REQUIRE(Contains(45, root) == 0);

    root = Remove(1, root);
    root = Remove(7, root);
    root = Remove(12, root);
    root = Remove(9, root);
    root = Remove(15, root);
    root = Remove(5, root);
    root = Remove(42, root);
    root = Remove(45, root);
    root = Remove(25, root);
    root = Remove(40, root);
    root = Remove(30, root);
    root = Remove(20, root);
    root = Remove(16, root);

    free(root);
}
```

As seen, all tests are passed which means that the program has been implemented successfully.



```
===============================================================================
All tests passed (62 assertions in 2 test cases)
```