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.

```
25  /* print list to console */
26  void print_list(node *p) {
27    if(p != NULL) {
28      printf("%d--->", p->value);
29      print_list(p->next);
30    } else {
31      printf("NULL");
32    }
33  }
```

When running the function, we get the following output:

```
39  node* list = make_node(1, make_node(2, make_node(3, make_node(4, make_node(5, NULL)
40        )
41      )
42    )
43  );
44  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:

```
35  int sum_squares(node *p) {
36    if(p != NULL) {
37      return p->value * p->value + sum_squares(p->next);
38    }
39    return 0;
40  }
```

When running the function, we get the following output:

```
47    printf("\n\nSum of squares is: %d\n", sum_squares(list));
48
49    node* test_list = make_node(1, NULL);
50    print_list(test_list);
51    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:

```
44  node *map(node *p, fn_int_to_int f) {
45    if(p != NULL) {
46      return make_node(f(p->value), map(p->next, f));
47    }
48    return NULL;
49  }
```

When running the function, we get the following results:

```
54    printf("\n\n");
55    print_list(list);
56    printf("\n\n");
57    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:

```
203   // Initialize tree
204   struct tree_node *Initialize(struct tree_node *t) {
205     t = NULL; // creates tree
206     return t; // returns the newly created tree
207   }
```

Insert function:

```
 8  // Insert node in tree
 9  struct tree_node *Insert(int x, struct tree_node *t) {
10    struct tree_node *newNode = malloc(sizeof(struct tree_node)); // creates node to insert
11    newNode->left = NULL; // sets the node's left child to NULL
12    newNode->right = NULL; // sets the node's right child to NULL
13    newNode->item = x; // assigns the given value to the newly created node
14    if(Empty(t) == 1) { // checks whether the tree is empty or not
15      t = newNode;
16    } else if(newNode->item <= t->item) { // less than or equal (left)
17      if(t->left == NULL) { // if end of the tree
18        t->left = newNode; // assigns node to the end
19      } else {
20        t->left = Insert(x, t->left); // checks deeper into the tree
21      }
22    } else if(newNode->item > t->item) { // greater than (right)
23      if(t->right == NULL) { // if end of the tree
24        t->right = newNode; // assigns node to the end
25      } else {
26        t->right = Insert(x, t->right); // checks deeper into the tree
27      }
28    } else { // inserting the node and assigning the tree's children to the node's
29      newNode->left = t->left;
30      newNode->right = t->right;
31      t = newNode;
32    }
33    return t; // returns the tree
34  }
```

Remove function:

```
37  // Remove node from tree
38  struct tree_node *Remove(int x, struct tree_node *t) {
39    if(t->item == x) { // if the value has been found
40      if(t->left == NULL) { // checks if node has a left child or not
41        struct tree_node *tempNode = t->right;
42        return tempNode;
43      } else if(t->right == NULL) { // checks if node has a right child or not
44        struct tree_node *tempNode = t->left;
45        return tempNode;
46      }
47      struct tree_node *tempNode = t->right; // temporary node to search for minimum value
48      while(tempNode && tempNode->left != NULL) { // loops until end of left side
49        tempNode = tempNode->left; // continue search
50      }
51      t->item = tempNode->item; // moves item to removed element's spot
52      t->right = Remove(tempNode->item, t->right);
53      return t; // returns the tree
54    } else if(x < t->item && t->left != NULL) { // less than (left)
55      t->left = Remove(x, t->left); // searches for element in left side
56    } else if(x > t->item && t->right != NULL) { // greater than (right)
57      t->right = Remove(x, t->right); // searches for element in right side
58    }
59    return t; // returns the tree
60  }
```

Contains function:

```
62  // Check if tree contains value
63  int Contains(int x, struct tree_node *t) {
64    int contain = 0; // sets contains to false as default
65    if(t->item == x) { // checks if list contains element
66      contain = 1; // sets the contains to true if elements is included in list
67    } else if(x < t->item && t->left != NULL) { // less than (left)
68      contain = Contains(x, t->left); // searches further into left side
69    } else if(x > t->item && t->right != NULL) { // greater than (right)
70      contain = Contains(x, t->right); // searches further into right side
71    }
72    return contain; // returns either true or false
73  }
```

Empty function:

```
81  // Check if tree is empty
82  int Empty(struct tree_node *t) {
83      if(t == NULL) { // checks if tree is empty
84          return 1; // if tree empty returns true
85      }
86      return 0; // else returns false
87  }
```

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

```
60      struct tree_node *root = NULL;
61      Initialize(root);
62      printf("\n\n");
63      root = Insert(20, root);
64      root = Insert(10, root);
65      root = Insert(40, root);
66      root = Insert(5, root);
67      root = Insert(9, root);
68      root = Insert(3, root);
69      root = Insert(45, root);
70      root = Insert(2, root);
71      root = Insert(11, root);
72      root = Insert(8, root);
73      root = Insert(10, root);
74      print_tree(root, 2);
```

```
76    if(Contains(20, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
77    if(Contains(10, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
78    if(Contains(40, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
79    if(Contains(5, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
80    if(Contains(9, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
81    if(Contains(3, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
82    if(Contains(45, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
83    if(Contains(2, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
84    if(Contains(11, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
85    if(Contains(8, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
86    if(Contains(10, root) == 1) printf("Contains\n\n"); else printf("DONT Contains\n\n");
87
88    if(Contains(1, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
89    if(Contains(4, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
90    if(Contains(6, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
91    if(Contains(7, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
92    if(Contains(47, root) == 1) printf("Contains\n"); else printf("DONT Contains\n");
93
94    Remove(9, root);
95    Remove(7, root);
96    print_tree(root, 2);
```

```
101   root = Insert(-1, root);
102   root = Insert(-1, root);
103   printf("\n\n\n");
104   print_tree(root, 2);
105   Remove(-1, root);
106   printf("\n\n\n");
107   print_tree(root, 2);
```

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

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

```
118     // (B) and (C)
119
120     root = Insert(3, root);
121
122     REQUIRE(Contains(3, root) == 1);
123
124     root = Remove(3, root);
125
126     REQUIRE(Contains(20, root) == 1);
127     REQUIRE(Contains(5, root) == 1);
128     REQUIRE(Contains(1, root) == 1);
129     REQUIRE(Contains(15, root) == 1);
130     REQUIRE(Contains(9, root) == 1);
131     REQUIRE(Contains(7, root) == 1);
132     REQUIRE(Contains(12, root) == 1);
133     REQUIRE(Contains(30, root) == 1);
134     REQUIRE(Contains(25, root) == 1);
135     REQUIRE(Contains(40, root) == 1);
136     REQUIRE(Contains(45, root) == 1);
137     REQUIRE(Contains(42, root) == 1);
138
139     REQUIRE(Contains(2, root) == 0);
140     REQUIRE(Contains(3, root) == 0);
```

```
142     // (D) and (E)
143     root = Insert(-1, root);
144     root = Insert(-1, root);
145     root = Remove(-1, root);
146     REQUIRE(Contains(-1, root) == 1);
147     root = Remove(-1, root);
148     REQUIRE(Contains(-1, root) == 0);
149
150     root = Remove(45, root);
151     root = Remove(42, root);
152     root = Insert(16, root);
153
154     REQUIRE(Contains(20, root) == 1);
155     REQUIRE(Contains(5, root) == 1);
156     REQUIRE(Contains(1, root) == 1);
157     REQUIRE(Contains(15, root) == 1);
158     REQUIRE(Contains(9, root) == 1);
159     REQUIRE(Contains(7, root) == 1);
160     REQUIRE(Contains(12, root) == 1);
161     REQUIRE(Contains(30, root) == 1);
162     REQUIRE(Contains(25, root) == 1);
163     REQUIRE(Contains(40, root) == 1);
164     REQUIRE(Contains(45, root) == 0);
165     REQUIRE(Contains(42, root) == 0);
166     REQUIRE(Contains(16, root) == 1);
```

```
170     root = Remove(7, root);
171
172     REQUIRE(Contains(16, root) == 1);
173     REQUIRE(Contains(20, root) == 1);
174     REQUIRE(Contains(5, root) == 1);
175     REQUIRE(Contains(1, root) == 1);
176     REQUIRE(Contains(15, root) == 1);
177     REQUIRE(Contains(9, root) == 1);
178     REQUIRE(Contains(7, root) == 0);
179     REQUIRE(Contains(12, root) == 1);
180     REQUIRE(Contains(30, root) == 1);
181     REQUIRE(Contains(25, root) == 1);
182     REQUIRE(Contains(40, root) == 1);
183     REQUIRE(Contains(45, root) == 0);
184
185     root = Remove(1, root);
186     root = Remove(7, root);
187     root = Remove(12, root);
188     root = Remove(9, root);
189     root = Remove(15, root);
190     root = Remove(5, root);
191     root = Remove(42, root);
192     root = Remove(45, root);
193     root = Remove(25, root);
194     root = Remove(40, root);
195     root = Remove(30, root);
196     root = Remove(20, root);
197     root = Remove(16, root);
198
199     free(root);
200 }
```

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

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