Week 10 Programming Assignment

Steffen Petersen — au722120

November 14th 2022

Here is the link for my repository, in which you will find all the edited code files and such.
https://github.com/Aarhus-University-ECE/assignment-10-SirQuacc

1

Write a recursive function that prints out a linked list of integers.

Here is my code for this function, it can also be found in linked_list.c

```c
void print_list(node *p) {
    printf("-> %d ", p->value); //Print current value
    if(p->next != NULL) print_list(p->next); //If we're not at the end, recursively send the next node
  }
```

2

Write a recursive function that accepts a linked list of integers, and returns the sum of the *squares* of the integers in the list. E.g.

```c
int sum;

node* list =
  make_node(1,
    make_node(2,
      make_node(3,
        make_node(4,
          make_node(5, &SENTINEL)
        )
      )
    )
  );

sum = sum_squares(list); /* sum should equal 55 */
```

Below is the recursive function, it can also be found in linked_list.c

```c
int sum_squares(node *p) {
    if(p == NULL) return 0; //If we've gotten to the null pointer at the end of the list, or input was empty, return 0
```

```
        else return (p->value*p->value + sum_squares(p->next)); //square the input value and
            recursively add the same from the next nodes.
      }
```

3

In the lecture we discussed the *fold* (or *reduce*) function that takes a list and a *binary* operator (i.e. an operator that takes two inputs) and returns an integer. Another useful general higher order function is *map* that takes a list and a unary operator (i.e. an operator that accepts just one input), applies the operator to each element in the list, and returns a new list of the resulting values. For example, if X is a list:

```
node* X =
  make_node(1,
    make_node(2,
      make_node(3,
        make_node(4,
          make_node(5, &SENTINEL)
        )
      )
    )
  );
```

We also have a *square* function that multiplies a value by itself:

```
int square(int x) {
  return x * x;
}
```

Your task in this exercise is to implement a recursive *map* function. When you pass the list X and the function *square* to your *map* function, the result should be a linked list with the following elements: 1, 4, 9, 16, 25.

Hint: you should assign the function *square* to a variable, and pass this variable to your *map* function, e.g. consider the following code that assigns the *square* function to a variable:

```
int (*sf)(int);
sf = square;
int x;
x = sf(2);
```

The code for this is seen below, and can be found in linked_list.c

```
typedef int (*fn_int_to_int)(int);
node *map(node *p, fn_int_to_int f) {
  if(p != NULL){ //If not at the end, return a node with the applied function's value and
      next node being recursive.
    return make_node(f(p->value), map(p->next, f));
  }
```

```
    return NULL; //Terminate the list, when the end is reached.
}
```

4

A *binary tree*[1] is a container that stores items for potentially faster retrieval than a linked list. It is equipped with a search operation (called `Contains(x,t)` below). A binary tree is composed from nodes and leafs that store an item. Each node has maximally two children (a *left* child and a *right* child) each of which is either a node or a leaf. The node is called the parent node of the children. A leaf does not have children. The node without a parent is called the root node of the tree. A tree has exactly one root node. The nodes (and leafs) in a binary tree are ordered such that, the left child of a node is smaller or equal than the item of the node and the right child is larger.

The abstract operations on a binary tree include —

- `Insert(x,t)` – Insert item `x` into the tree `t`.
- `Remove(x,t)` – Remove one item `x` from tree `t`. Do nothing if `x` is not contained in the tree.
- `Contains(x,t)` – Return `true` if the tree `t` contains item `x`. Return `false` otherwise.
- `Initialize(t)` – Create an empty tree.
- `Full(t)`, `Empty(t)` – Test whether the tree can accept more insertions or removals, respectively.

(a) Implement a binary tree following the implementation of the singly-linked lists as discussed in the lecture.

(b) Test your implementation. You should expect the following “laws” to hold for any implementation of a queue:

  (A) After executing `Initialize(t);` the tree `t` must be empty.

  (B) After executing `Insert(x,t); Remove(x,t);` the tree `t` must be the same as before execution of the two commands.

  (C) After executing `Insert(x,t); y = Contains(x,t);` the value of `y` must be `true`.

  (D) After executing
  `Insert(x,t); Insert(y,t); Remove(x,t); z = Contains(y,t);`
  the value of `z` must be `true`.

  (E) After executing
  `Insert(x,t); Insert(x,t);`
  `Remove(x,t); y = Contains(x,t);`
  `Remove(x,t); z = Contains(x,t);`
  the value of `y` must be `true` and the value of `z` must be `false`.

Implement a binary tree following the implementation of the singly-linked lists as discussed in the lecture.

The solutions to this exercise are found in btree.c, and the testing of the program is done via CMake and tests.cpp

Also, the solutions are made assuming that ALL nodes on the left of a parent, are smaller than or equal to that parent.

Similarly on the right, ALL nodes to the right, also children of children, are larger than the parent.