Programming for computerteknologi
Hand-in Assignment Exercises

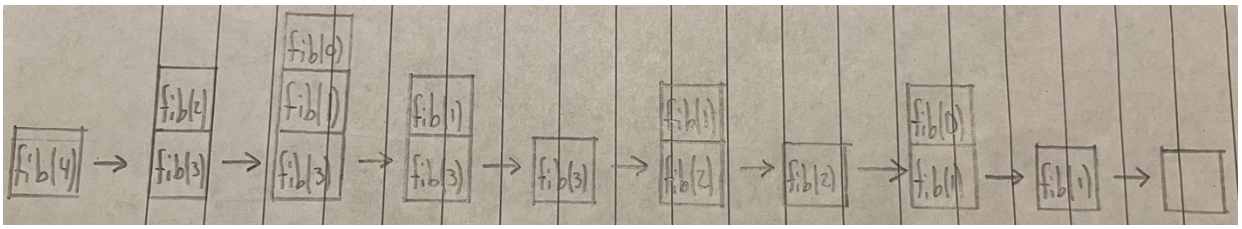
Week 9: Functions that call themselves

Exercise 1)

When running functions, we can draw a runtime stack to illustrate and calculate the different steps when working with recursive functions. In this exercise, we have been asked to look at the Fibonacci numbers which follows the following with two different base cases and one recursive step.

$$\text{fib}(n) = \begin{cases} \text{fib}(n-1) + \text{fib}(n-2), & \text{if } n > 1 \\ 1, & \text{if } n = 1 \\ 0, & \text{if } n = 0 \end{cases}$$

With all this in mind, I have made the following runtime stack.



Exercise 2)

When summing an array, it can be described recursively as follows, where a is the array and n is the length of the array.

$$sum(a, n) = \begin{cases} a[n-1] + sum(a, n-1), & \text{if } n > 0 \\ 0, & \text{if } n = 0 \end{cases}$$

With this in mind, I have implemented a recursive function that sums the integer array a.

```c
int sum(int a[], int n)
{
    /* precondition */
    assert(n >= 0);

    /* postcondition */
    if(n > 0) {
        return a[n - 1] + sum(a, n - 1);
    } else {
        return 0;
    }

    return 0;
}
```

Exercise 3)

We have been given a task, to make a recursive function that searches the integer array a for the element x. A recursive function not requiring the elements to be sorted could look like this:

$$search(a, n, x) = \begin{cases} true, & \text{if } n > 0 \text{ and } a[n-1] == x \\ search(a, n-1, x), & \text{if } n > 0 \text{ and } a[n-1] != x \\ false, & \text{if } n = 0 \end{cases}$$

With this in mind, I have implemented the following function:

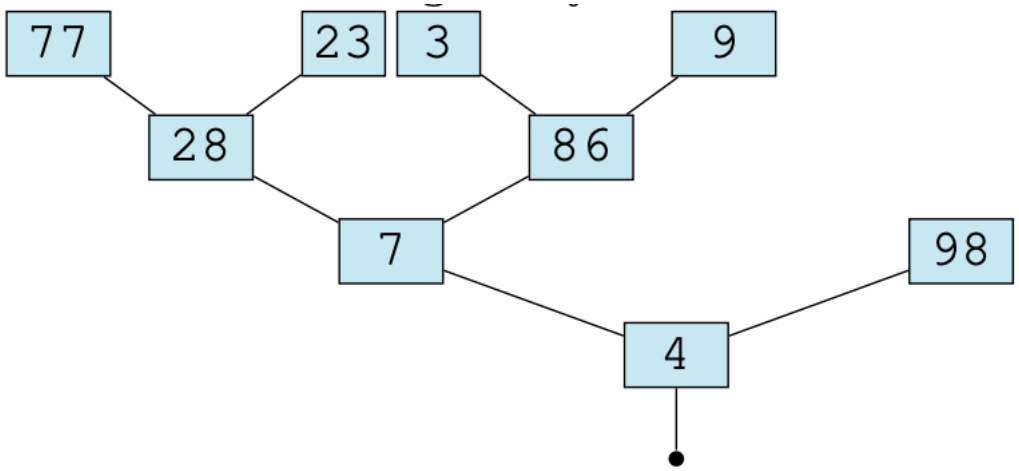
```c
bool search(int a[], int n, int x)
{
    /* precondition */
    assert(n >= 0);

    /* postcondition */
    if(n > 0 && a[n - 1] == x) {
        return true;
    } else if(n > 0 && a[n - 1] != x) {
        return search(a, n - 1, x);
    } else {
        return false;
    }

    return 0;
}
```

Exercise 4)

We have been asked to implement depth-first search using a stack. The stack is implemented as a linked list and the tree as tree nodes that each have an integer as the data item and a left and right child. The tree to be tested look like this.



With this in mind, we now implement the code.

```c
void DFT (node * root) {
  struct stack *runtimeStack = malloc(sizeof(stack)); // implementing runtime stack
  runtimeStack->node = root; // assigning the first element of the stack to the tree's root

  printf("The output is: ");
  while(isEmpty(runtimeStack) == false) { // looping through tree until every node has been visited
    struct node *tempNode = top(runtimeStack); // creating temporary node and assigning top node
    runtimeStack = pop(runtimeStack); // removing top element from the stack
    printf("%d", tempNode->num); // printing visited node
    if(tempNode->rchild) { // checking if node has a right child
      runtimeStack = push(runtimeStack, tempNode->rchild); // pushing right child to stack
    }
    if(tempNode->lchild) { // checking if node has a left child
      runtimeStack = push(runtimeStack, tempNode->lchild); // pushing left child to stack
    }
    if(isEmpty(runtimeStack) == false) { // printing out ", " if stack is not empty
      printf(", ");
    }
  }
}
```

The code prints out the following output

```
The output is: 4, 7, 28, 77, 23, 86, 3, 9, 98%
```