Programmering for computerteknologi Hand-in Assignment Exercises

Week 7: Designing sequences of program instructions for solving problems

Written by: Alexander A. Christensen (202205452)

Disclaimer: Due to errors with CMake that neither me, nor the TAs have solved, the test-cases have not been run. Instead, the functions have been manually tested.

The code can still be found at https://github.com/Aarhus-University-ECE/assignment-8-A-CHRI

Exercise 1

The following algorithm for computing factorials is given

```c
/* Factorial function definition */
int fact(int n)
{
    int i; /* counter variable */
    int f; /* factorial */

    /* pre-condition */
    assert (n >= 0);

    /* post-condition */
    f = 1;
    for(i = 1; i <= n; i = i + 1)
    {
        f = i * f;
    }
    return f;
}
```

We wish to count the amount of arithmetic operations it takes to compute `fact(5)`. We notice that each iteration in the for-loop has 2 operations being `i = i + 1` and `f = i * f`. The amount operation can then be calculated with $5 \cdot 2 = 10$. For `fact(n)`, the amount can instead be calculated by $n \cdot 2$.

Exercise 2

We wish to implement the **Insertion sort** algorithm using linked list. This can be done bu creating a sorted list, and inserting each element from the given list into the sorted list, and checking where to insert. This is done by looping through the sorted list and comparing the element to be inserted, to the already sorted elements. This has been implemented below.

```c
node *sort(node *list)
{
    /* Pre: Non-empty list */
    assert(list != NULL);

    /* Post: Sort the array using insertion sort */
    node *sorted = NULL;

    node *curr = list;
    while (curr != NULL)
    {

        /* Create a new node */
        struct node *new = (node *)malloc(sizeof(node));
```

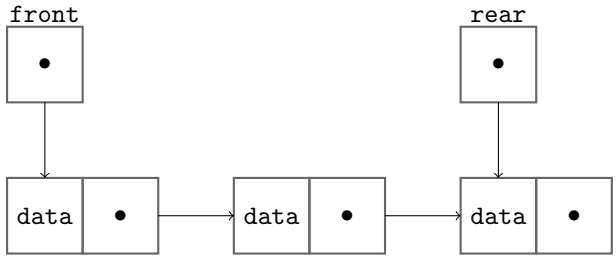
```
        new->data = curr->data;
        new->next = NULL;

        /* Special cases */
        if (sorted == NULL || sorted->data >= new->data)
        {
            new->next = sorted;
            sorted = new;
        }
        else
        {
            node *ins = sorted;
            while (ins->next != NULL && ins->next->data < new->data)
            {
                ins = ins->next;
            }
            new->next = ins->next;
            ins->next = new;
        }
        curr = curr->next;
    }
    return sorted;
}
```

Exercise 3

We wish to implement a queue data structure using a linked list. The queue can be represented using a front and rear pointer. The queue will have the following properties.

- `enqueue(q, x)`: Insert integer `x` into queue `q`
- `dequeue(q)`: Remove the front element in the queue and return it
- `init_queue(q)`: Initialise the queue
- `empty(q)`: Boolean value

Using linked lists to represent a queue is done using a front and rear pointer. When initialising it the pointers is both set to `NULL`. Whenever an element is added to the queue, it gets added to the rear end of the queue, letting the rear pointer point to it. When removing an element, we return the from element, and move the front element by one.



Implementation

```
void init_queue(queue *q)
{
    q->front = NULL;
    q->rear = NULL;
}

```

```c
void enqueue(queue *q, int val)
{
    /*Post: Add value to the queue */

    /* Create a new node */
    struct node *new = (node *)malloc(sizeof(node));
    new->data = val;
    new->next = NULL;

    /* If the queue is empty, let the rear pointer */
    if (q->rear == NULL && q->front == NULL)
    {
        q->front = new;
        q->rear = new;
    }
    q->rear->next = new;
    q->rear = new;
}

int dequeue(queue *q)
{
    /* Pre: Non-empty queue */
    assert(q->front != NULL);

    /*Post: Remove and return the rear element */
    node *t = q->front;

    q->front = q->front->next;

    int temp = t->data;
    free(t);

    return temp;
}

bool empty(queue *q)
{
    if (q->front == NULL && q->rear == NULL)
    {
        return true;
    }
    return false;
}
```

Testing

When testing the implementation we wish to test the following criteria:

1. After executing `init_queue(q);` the queue `s` must be empty
2. After executing `enqueue(q,x); y = dequeue(q);` the queue `q` must be the same as before execution of the two commands, and `x` must equal `y`.
3. After executing `enqueue(q, x0); enqueue(q, x1); y0 = dequeue(q); y1 = dequeue(q);` the queue `q` must be the same as before execution of the two commands, `x0` must equal `y0`, and `x1` must equal `y1`.

For displaying the queue at any point, the following function `printq(q)`, has been implemented.

```c
void printq(queue *q)
{
    /* Pre: Non-empty queue */
    assert(q->front != NULL);

```

```
    /* Post: Print the contents of the queue */
    node *t = q->front;
    printf("%d ", t->data);
    while (t != q->rear)
    {
        t = t->next;
        printf("%d ", t->data);
    }
    printf("\n");
}
```

In the main function, the three test scenarios has been implemented as shown below.

```
/* Create the queue*/
queue q;
init_queue(&q);

/* TEST A: After initialising the queue the queue must be empty */
if (empty(&q))
{
    printf("The queue is empty!\n");
}
else
{
    printf("The queue is NOT empty!\n");
}

/* TEST B: After queing and dequeueing an element, the element queed should match the
    one dequeued */
int x = 1;
enqueue(&q, x);
int y = dequeue(&q);
printf("x = %d should be the same as y = %d\n", x, y);

/* TEST C: After queing two elements and dequeueing them, the elements should come out
     in the right order */
int x1 = 1;
int x2 = 2;
enqueue(&q, x1);
enqueue(&q, x2);
int y1 = dequeue(&q);
int y2 = dequeue(&q);
printf("x1 = %d should be the same as y1 = %d, and x2 = %d should be the same as y2 =
    %d\n", x1, y1, x2, y2);
return 0;
```

This prints the following to the console:

```
The queue is empty!
x = 1 should be the same as y = 1
x1 = 1 should be the same as y1 = 1, and x2 = 2 should be the same as y2 = 2
```

This shows that the implementation holds up to the 3 given test scenarios