DMA Hand-in, week 4

Magnus, Mathilde og Mikkel

30. september 2020

Part A We look at a sorted doubly linked list S.

1. - In a sorted doubly linked list, each node has the fields prev, key and next, where key is a number. Next and prev are pointers.
   - This assignment is a bit unspecific. However, if the goal is to make a list with the element z, then the following code would be preferable:

```
createListWithElement(z)
    node = new Node{prev = NIL, key = z, next = NIL} // First, create
    // the node with key = z
    L = new List{head = node} // Point the head of list L to the node
    return L
```

2. The following pseudo code will solve the given problem:

```
F(S,z)
    newNode = new Node{prev = NIL, key = z, next = NIL}
    x = S.head
    while x.next != NIL and x.key < z // While loop to find the
    // position of the new node
        x = x.next
    if x.next == NIL // This will result in x being the last
    // element of the list
        newNode.prev = x // x is now the previous element
        newNode.next = NIL // newNode is the last element
        S.tail = newNode // And therefore, newNode is the tail
         return S
    else
        newNode.next = x // newNode is inserted before x and next points to x
        newNode.prev = x.prev // The previous node is what x pointed to
        x.prev.next = newNode // The previous node points towards newNode
        x.prev = newNode // newNode is the previous node to x
        return S
```

3. In the worst case, the while loop will run n times. All other statements takes constant time, therefore, the worst possible run-time will be $\Theta(n)$.
4. In the worst case, the algorithm will put the new number in the end of the list, every time. So for every time we put in a new number in the list, the iteration of finding the place to put this number, will run an extra loop. So for the first number, this loops 1 time, then for the second number, 2 times. So we essentially get the sum:

$$\sum_{k=1}^{n} k = \frac{n^2 + n}{2}$$

   This general expression is found in Theorem 13. This sum is $O(n^2)$.
5. The algorithm is very similar to insertion-sort, and they have the same worst case run-time.

Part B As in Part A, We look at a sorted doubly linked list S. We have written a pseudo code snippet to accomplish this task.

1.
```
1 Divide(S)
2        x = S.head
3        B = new List{head = S.head, size = sqrt(S.length)}
4        y = B.head
5        k = sqrt(S.length)
6        for i = 0 to k - 1
7            for j = 0 to k - 1
8                x = x.next
9            y = y.next
10           y.pa = x
11       Return B
```

First create a variable, x, that we use to loop though S. We then create a list, B, that we assume has pointers between all elements. The only thing we need to do now, is to point the pa attribute in each node, to the right value in S. We do that with the loops. The run time of the algorithm is $\Theta(k^2) = \Theta(n)$, due to the two loops.

2. It takes $\Theta(k) = \Theta(\sqrt{n})$

3. The function needs to go through the list B and check the pointers pa. Each pointer pa points towards a value which is the head of a sublist l_i. If $l_i.head > x$ then the function needs to go back to l_{i-1} and check if $l_{i-1}.tail < x$ in order to figure out where to insert x in order to keep the list sorted. If $l_{i-1}.tail! < x$ then check if $l_{i-1}.tail.prev! < x$. It then needs to continue checking the previous value of the one just checked and when it finds a value in $l_{i-1} < x$, then the function needs to insert x after that value, so x becomes its successor and thereby keeps the list sorted.
This would take $O(\sqrt{n})$ time because the function would first run through k elements in the list B, then it would enter a sublist l_i, where the function also would run through k elements. I.e. the function would run $2 \cdot k$ which is the same as $O(k)$ time which is also the same as $O(\sqrt{n})$ time, since n was defined as $k \cdot k$