Suggested solutions – Exam – Datastrukturer

DIT960/DIT961, VT-19
Göteborgs Universitet, CSE

Day: 2019-08-23, *Time:* 8:30-12.30, *Place:* J

Exercise 1 (complexity)

Remember that worst-case complexity $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

a) Yes, because $n^2 + n + 1 \leq 3n^2$ for all $n \geq 1$, that is $c = 3$ and $n_0 = 1$. Alternatively, you can apply the hierarchy and multiplication rules: $O(f(n)) = O(n^2 + n + 1) = O(n^2 + n) = O(n^2)$.

b) Yes, take $c = 1$ and $n_0 = 0$.

c) Yes, use the hierarchy rules: $O(n^3) \leq O(2^n)$.

d) No, use the hierarchy rules: $f(n) = n \log n$ is not $O(n)$.

e) Yes, use the hierarchy rules: $O(n \log n) \geq O(n)$.

For a VG only:

For the `pack` function we can write the following recurrence relation:

$$T(n) = O(n) + T(n-1)$$

due to the fact that at every call we execute `takeWhile` and `dropWhile` (so $2O(n) = O(n)$) and go in recursion with the tail of the list, which is one less in size. We can expand this recurrence relation, but it is one of the standard relations and boils down to $O(n^2)$.

Although we accept the above as the correct answer, this is an overestimation we can be more precise. We start with the following recurrence relation:

$$T(n) = 2p_0 + T(n - p_0)$$

where p_0 is the number of consecutive equal elements at the start of the list, which are taken and dropped with the `takeWhile` and `dropWhile` respectively. The remainder of the list is $n - p_0$ long. We can expand and continue with the next batch of equal elements:

$$\begin{aligned} T(n) &= 2p_0 + T(n - p_0) \\ &= 2p_0 + 2p_1 + T(n - p_0 - p_1) \end{aligned}$$

and then again continue:

$$\begin{aligned} T(n) &= 2p_0 + T(n - p_0) \\ &= 2p_0 + 2p_1 + T(n - p_0 - p_1) \\ &= \ldots \\ &= 2p_0 + 2p_1 + \ldots + 2p_n + T(0) \end{aligned}$$

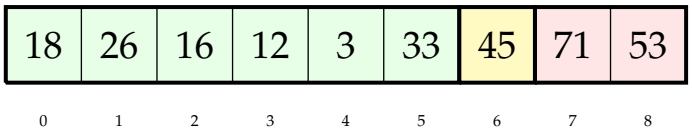
The sum of p_0 to p_n is n! And $T(0)$, the empty list case, is $O(1)$. So we can conclude:

$$T(n) = 2n + T(0) = O(n) + O(1) = O(n)$$

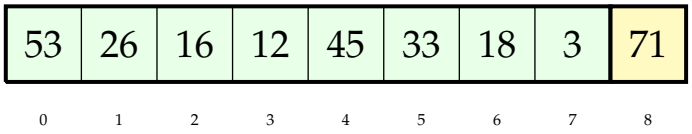
Exercise 2 (sorting)

It is important that all elements to the left of the pivot are *smaller* and to the right are *larger*, that is, the pivot should be in the right place. The elements in the to be sorted arrays should be in the correct order, reflecting how the quicksort algorithm works. The subarrays next to the pivot should *not* necessarily be sorted.

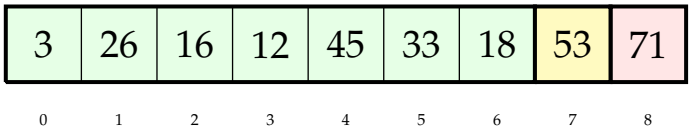
a)



b)



c)



For a VG only:

```
merge :: Ord a => [a] -> [a] -> [a]
merge []     ys     = ys
merge xs     []     = xs
merge (x:xs) (y:ys) | x < y     = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys
```

The `merge` function only uses constant time functions in each call (such as `(:)` and `(<)`) and goes in recursion once with two lists, if which one is one smaller in size. So, the recurrence relation for this function is:

$$T(n) = O(1) + T(n - 1)$$

where n is the sum of the size of the two input lists. This relation is one of the standard recurrence relations and is $O(n)$.

Exercise 3 (basic data structures)

a) All methods have $O(1)$ worst-case time complexity.

b)
```
public static <E> int size(Stack<E> s) {
  int n = 0;

  while (!s.isEmpty()) {
    s.pop();
    n++;
  }

  return n;
}
```

c) The complexity is $O(n)$.

d) You can add an instance variable that keeps track of the number of elements on the stack. The disadvantage is that it takes a bit of memory and many methods should keep this variable up to date. But you have constant time access to the size.

For a VG only:

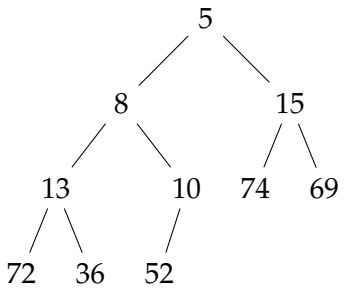
```
1  import java.util.NoSuchElementException;
2
3  public class LinkedList<E> implements Stack<E> {
4    private class Node {
5      E data;
6      Node next;
7    }
8
9    private Node top = null;
10
11   @Override
12   public void push(E elem) {
13     Node n = new Node();
14     n.next = top;
15     n.data = elem;
16     top = n;
17   }
18
19   @Override
20   public E pop() {
21     if (top == null)
```

```
22        throw new NoSuchElementException("Empty stack!");
23      else {
24        E data = top.data;
25        top = top.next;
26        return data;
27      }
28    }
29
30    @Override
31    public E top() {
32      if (top == null)
33        throw new NoSuchElementException("Empty stack!");
34      else
35        return top.data;
36    }
37
38    @Override
39    public boolean isEmpty() {
40      return top == null;
41    }
42  }
```

Exercise 4 (heaps)

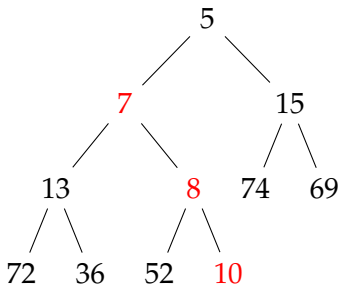
A is the only array that represents a binary heap, which can be drawn as a tree as follows:

A



B

The moved bits and new value are drawn in red.



As array: `[5, 7, 15, 13, 8, 74, 69, 72, 36, 52, 10]`

C

A complete tree means that means that all levels except the bottom one are full, and the bottom level is filled from left to right. This makes sure that the tree is balanced and that a new element has only one place to go. Moreover, we can implement it using an array!

Exercise 5 (search trees)

```
1  public class BST<E extends Comparable<? super E>> {
2    private class Node {
3      E data;
4      Node left;
5      Node right;
6
7      Node(E data) {
8        this.data = data;
9      }
10   }
11
12   private Node root;
13
14   public int height() {
15     return height(root);
16   }
17
18   private int height(Node n) {
19     if (n == null)
20       return -1;
21     else
22       return 1 + Math.max(height(n.left), height(n.right));
23   }
24
```

```
25    public void insert(E data) {
26      if (root == null)
27        root = new Node(data);
28      else
29        insert(data, root);
30    }
31
32    // Node is not null
33    private void insert(E data, Node n) {
34      int cmp = data.compareTo(n.data);
35
36      if (cmp > 0)
37        if (n.right == null)
38          n.right = new Node(data);
39        else
40          insert(data, n.right);
41      else if (cmp < 0)
42        if (n.left == null)
43          n.left = new Node(data);
44        else
45          insert(data, n.left);
46      else
47        n.data = data;
48    }
49
50    public boolean invariant() {
51      return invariant(root, null,  null);
52    }
53
54    private boolean invariant(Node n, E min, E max) {
55      if (n == null)
56        return true;
57
58      if (min != null && n.data.compareTo(min) < 0 ||
59          max != null && n.data.compareTo(max) > 0)
60        return false;
61
62      return invariant(n.left, min, n.data) && invariant(n.right, n.data, max);
63    }
64
65    public static void main(String[] args) {
66      BST<Integer> tree = new BST<>();
67
68      tree.insert(42);
69      tree.insert(20);
70      tree.insert(30);
```

```
71      tree.insert(88);
72      tree.insert(89);
73
74      System.out.println(tree.height());
75
76      tree.root.left.right.data = 44;
77
78      System.out.println(tree.invariant());
79    }
80  }
```

Exercise 6 (graphs)

a) $\{BA, AD, DF, FE, AC, CG\}$

b) G : 0, C : 2, A : 5, B : 6, D : 6, E : 7, F : 8

Depending on the priority queue implementation nodes with equal shortest distances (such as B and D) may be visited in a different order.