Suggested solutions – Re-exam – Datastrukturer

DIT961, VT-22
Göteborgs Universitet, CSE

Day: 2022-08-18, *Time:* 8:30-12.30, *Place:* Johanneberg

Exercise 1 (complexity)

a) Both loops iterate over each element in the array, so n times. The first loop calls `map.put` for each element, and in the worst case BST is unbalanced and so put has linear complexity.

So the first loop is $O(n) * O(n) = O(n^2)$, and the second is $O(n)$. In the end the complexity of `sort_values` is quadratic, $O(n^2)$

b) This time the tree will always be balanced, so adding elements to it is logarithmic in the worst case. Therefore the first loop (and the whole program) is $O(n) * O(\log n) = O(n \log n)$

For a VG only:

c) A simple example is if the `keys` array is sorted (in any order). Then the BST will be completely unbalanced (to the left or the right), and adding new elements will take $O(n)$ time. How the `values` array looks doesn't matter at all.

d) The keys in a hash table are not stored in order, so when we iterate through the map entries we will get them in an unknown order. So `sort_values` will not sort the values according to the keys.

e) If we have a perfect hash function, adding elements to a hash table is constant time, $O(1)$. So the first loop becomes $O(n) * O(1)$, and therefore the complexity of `sort_values` will be $O(n)$.

Exercise 2 (sorting)

a) The easiest way is to split in half, so you get the two lists: `[42, 12, 54, 72, 36]` and `[82, 99, 66, 6, 27]`.

b) Calling merge sort on each half gives us two sorted lists: `[12, 36, 42, 54, 72]` and `[6, 27, 66, 82, 99]`.

c) Merging compares the heads of each list and puts the smallest in a new list:

```
1.  [12,36,42,54,72] + [6,27,66,82,99] -> [6]
2.  [12,36,42,54,72] + [27,66,82,99] -> [6,12]
3.  [36,42,54,72] + [27,66,82,99] -> [6,12,27]
4.  [36,42,54,72] + [66,82,99] -> [6,12,27,36]
5.  [42,54,72] + [66,82,99] -> [6,12,27,36,42]
6.  [54,72] + [66,82,99] -> [6,12,27,36,42,54]
7.  [72] + [66,82,99] -> [6,12,27,36,42,54,66]
8.  [72] + [82,99] -> [6,12,27,36,42,54,66,72]
9.  [] + [82,99] -> [6,12,27,36,42,54,66,72,82]
10. [] + [99] -> [6,12,27,36,42,54,66,72,82,99]
```

For a VG only:

d) The `merge` function can be defined as follows:

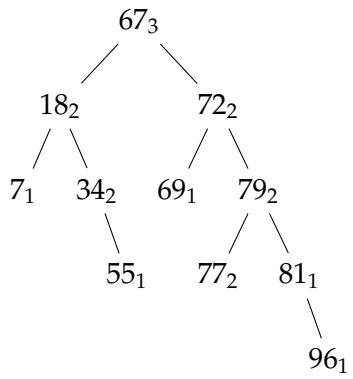
```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

e) This version of merge sort is in essence the same as insertion sort. The input list does not get divived into sublists of (nearly) equal size. In the worst case the to be merged singleton list contains an element that is greater than all elements in the other list in which case merge takes $O(n)$, which needs to be done n times, so the complexity is quadratic $O(n^2)$.

f) The best case is a reverse sorted list and the worst case a sorted list.

Exercise 3 (search trees)

The given tree in the exercise contained a small typo, which has quite a big impact on the questions, unfortunately. The root of the tree was supposed to be 67 and not 37:



This makes the questions c, d, e, and f more difficult to answer, so all attempts were accepted (that is, these questions are ignored).

a) The tree in the exam (with the incorrect root) is not a valid AA tree because does not satisfy the BST property (55 is larger than 37).

Answers that considered 67 as the root should point out that the level of 34 is wrong, it should be 1 (making it a proper three-node). The level of 77 is incorrect as well, and should be 1.

b) The tree given on the exam cannot be made into a AA tree because of the BST violation.

Otherwise one should use the suggestions from the previous answer.

c) The node 72 is imbalanced since its right child has height 3 and its left child has height 1.

d) Perform a single left rotation at node 72.

For a VG only:

e) Delete the root by replacing it by the largest element from the left subtree, which is 55.

f) Delete the root by replacing it by the smallest element from the right subtree, which is 69.

g)
```
delete :: Ord a => a -> Tree a -> Tree a
delete x Empty = Empty
delete x (Node l y r)
  | x < y     = Node (delete x l) y r
  | x > y     = Node l y (delete x r)
  | otherwise = maybe r (\(l', m) -> Node l' m r) (deleteMax l)
 where
  deleteMax Empty = Nothing
  deleteMax (Node l x r) = case r of
    Empty -> Just (l, x)
    _     -> deleteMax r
```

Exercise 4 (heaps)

a) This is a binary heap and the array looks like: [3, 6, 11, 12, 15, 32, 12, 19, 23, 17].

b) This is not a binary heap: the node 10 is smaller than its parent 12.

c) This is not a binary heap: it is not a complete tree.

d) This is a binary heap and the array looks like: [3, 11, 6, 19, 15, 12, 10, 32, 21].

For a VG only:

a) With the 0-based array indexing convention, an array `x` represents a heap exactly if for every valid array index *i*, we have `x[(i - 1) / 2 (rounded down)] <= x[i]`.

b) Call the dynamic array `x` and the element to be inserted `a`. Then we do:

```
i = size(x)
x.append(a)
loop:
        j = (i - 1) / 2 (rounded down)
        if (not j >= 0) or x[j] <= x[i]:
                break out of loop
        swap positions j and i of x
        i = j
```

Exercise 5 (basic data structures: hash table and queue)

a) 3, because the hash function could be very bad

b) 1

c) 3

d) 2, this is the idea with lazy deletion

For a VG only:

a) All methods have constant time worst case complexity $O(1)$. The usual implementation of a queue with a singly-linked list has both a pointer to the start as well as the end of the queue.

b) See source code below.

c) See source code below.

```java
import java.util.NoSuchElementException;

class QueueSingle<E> implements Queue<E> {
  private class Node {
    E data;
    Node next;
  }

  private Node first, last;
  private int size;

  public QueueSingle() {
    first = new Node();  // sentinel node
    last = first;
    size = 0;
  }

  public void enqueue(E elem) {
    Node n = new Node();
    n.data = elem;
```

```
21      last.next = n;
22      last = n;
23      size++;
24    }
25
26    public E dequeue() {
27      if (isEmpty()) throw new NoSuchElementException();
28
29      Node n = first.next;
30      E d = n.data;
31      first.next = n.next;
32      size--;
33
34      return d;
35    }
36
37    public boolean isEmpty() {
38      return size == 0;
39    }
40
41    public int size() {
42      return size;
43    }
44  }
```

Exercise 6 (graphs)

Node	Distance	Predecessor	Queue
A	0	-	{B-3, C-1, D-6}
C	1	A	{B-3, D-5, E-4, G-9}
B	3	A/C	{D-5, E-4, G-9}
E	4	C	{D-5, F-6, G-9}
D	5	C	{F-6, G-9}
F	6	E	{G-9}
G	9	C	{}

The priority queue may have extra duplicate nodes (with different distances) depending on the implementation.

For a VG only:

Here's an efficient solution. We use a variant of depth-first search that processes every vertex only the first time it is visited. With every vertex `v`, we store two Booleans flags (both initially false):

- ancestor: does `v` appear as an ancestor with respect to the current position in the

graph?

- visited: have we already processed v?

```
function has_cycle() -> bool:
        unmark all vertices (as both visited and ancestor)
        for each vertex v:
                if vertex_has_cycle(v):
                        return true
        return false

function vertex_has_cycle(v : Vertex) -> bool:
        if v is marked as ancestor:
                return true
        if v is not marked as visited:
                mark v as visited
                mark v as ancestor
                for each edge e from v:
                        if vertex_has_cycle(target of e):
                                return true
                unmark v as ancestor
        return false
```

We call `vertex_has_cycle(v)` for every vertex `v`. If this does not report a cycle, then no cycle exists.

We now check (not required) that this algorithm is $O(V + E)$. For every vertex `v`, the `for`-loop in the function `vertex_has_cycle` is executed only once. Thus, the loop body is executed only once for every edge. This also means that the visit function is called at most $V + E$ many times (E from this loop body and V from the outer loop). The remaining statements in an invocation of `vertex_has_cycle` are constant time.