1. part(a): we can find the area of all rectangles in one pass and in linear time. Since the coordinates are integer, the areas of all rectangles are integer values. So, we basically need to sort n positive integers, all smaller than n^2. This can be done, using radix sort, in linear time.

   part(b): the answer is yes. For circles, we just need to sort by radii, i.e., sort n positive integers, all smaller than n. This can be done in linear time using radix sort.

2. We sort the set X in ascending order keeping track of the original element indices. Since all $x_i \leq n^3$ we can use radix sort and obtain the sorted list X' in $O(n)$ time. We traverse X' and compute the differences $d_i = x'_{i+1} - x'_i$ for all i, $1 \leq i < n-1$. Since X' is sorted, for all t and all i satisfying $1 \leq t < n-1$ and $0 \leq i \leq n-t-1$, $|x'_{i+t} - x'_i| \geq |x'_{i+1} - x'_i|$. Therefore the minimum among all d_i is the difference with the smallest absolute value.

3. part (a): T_1 is not necessarily a balanced tree because the right subtree can be arbitrarily higher than the left one. For instance, consider a tree in which the left child of every internal node is a leaf; see Fig. 1.

Figure 1: Example of a non-balanced tree for 3(a).

   part (b): T_2 is a balanced tree. Let $N(h)$ be the minimal number of nodes in a tree T_2 of height h. Then $N(h) \geq N(h-1)+N(h-c-1)+1 \geq 2N(h-c-1)+1 > 2N(h-c-1)$. We can solve the last recurrence as in the lecture. Let $f = c+1$, then

$$N(h) > 2N(h-f) > 4N(h-2f) > \ldots > 2^i N(h-i \cdot f) > \ldots > 2^{\lfloor h/f \rfloor}$$

   Hence $h < f \log_2 N(h) = (c+1) \log_2 N(h)$.

   part (c): T_3 is not necessarily a balanced tree. The same counterexample as in part (a) can be used.

4. The left subtree of the root node has at most $n-1$ nodes. As shown in the analysis of an AVL tree height, the maximal height of a tree with $n-1$ nodes is $h_l = \log_\phi(\sqrt{5}(n-$

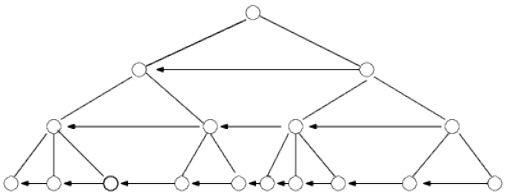

Figure 2: Pointers prev.

1)) − 3. The height of the right subtree is at most $h_r = h_l + 1$. The number of nodes in a binary tree of height h_r is at most $2^{h_r} + 2^{h_r-1} + \ldots + 2^1 + 2^0 = 2^{h_r+1} - 1$. Hence the total number of nodes does not exceed $(n-1) + 1 + 2^{h_l+2} - 1 = n - 1 + 2^{\log_\phi(\sqrt{5}(n-1))-1}$.

5. We keep all elements in a balanced search tree T implemented as an AVL tree. We also keep copies of all elements in a doubly-linked list L. We keep one additional pointer in every node u of T: if u holds an element a, then the pointer points to the node of L that contains the copy of a.

   When an element x is inserted, we insert it into T. We also append the copy of x at the end of L. Operation search(x) is implemented in the same way as in the AVL tree. When an element x is deleted, we find the node u_x in T that holds x. Then we follow the pointer from u_x to a node l_x in L and remove l_x from L. Finally we remove the key x from the AVL tree. In order to execute deleteLast(), we access the last element l_e in L; l_e is then deleted from both L and T.

6. Let T denote the B$^+$-tree with $M = 3$. In every non-root node we store a pointer to its parent node. Besides all nodes on the same tree level are connected. The list L_h contains all nodes of height h in the right-to-left order. The i-th node u in L_h contains a pointer $u.prev$ that points to the $(i+1)$-th node of height h. See Fig. 2. All key-value pairs are stored in the leaves. In internal nodes, we keep keys of selected elements. The search for the key x starts at the leaf l_f that contains e_f. During the first phase we visit ancestors of l_f until the node u is reached, such that one of the keys stored in u or in the previous node $u.prev$ is smaller than or equals to x. When u is reached, we start the second phase and move downwards. Let v be the node that holds a key $x' \leq x$, but at least one key in the subtree of v is not smaller than x. We can select either u or $u.prev$ as v. During the second phase we search for x in the subtree of v. The search is exactly the same as the search for x in a B$^+$-tree with root v. The pseudocode description is given below. We denote by $u.size$ the number of keys stored in the node u ($u.size$ is equal to either 1 or 2). The array $u.keys[]$ holds the keys stored in the node u.

   osearch(x, e_f)
   $u := l_f$
   $b := true$
   while $b == true$ **do**

```
    for i := u.size − 1 to 0 do
        if u.keys[i] ≤ x then
            b := false
            v := u
        end if
    end for
    if b == true then
        for i := u.prev.size − 1 to 0 do
            if u.prev.keys[i] ≤ x then
                b := false
                v := u.prev
            end if
        end for
    end if
    if b == true then
        u := parent(u)
    end if
end while
Search for x in the subtree of v
```

During the first phase we move up in the tree from a leaf to some node u; during the second phase the search algorithm moves from the node v to a leaf in the subtree of v. Since we spend constant time in every node on the path from the l_e to u and on the path from v to a leaf below v, the total time is $O(\text{height}(v) + \text{height}(u))$. Since each node in T has at least 2 children, there are at least 2^h leaves below every node of height h. Suppose that we visited a node u' during the upward phase and all keys stored in nodes u' and $u'.prev$ are larger than x. Then for at least one child u_1 of $u'.prev$, all keys stored in the subtree of u_l are larger than x and smaller than e_f. Hence $2^{\text{height}(u')-1} \leqslant d_f$ and $\text{height}(u') \leq \log d_f + 1$. Clearly $height(u) = height(u') + 1$ for some node u'. Hence $\text{height}(u) = O(\log d_f)$. Since $\text{height}(v) = \text{height}(u)$, the search takes $O(\log d_f)$ time.