Algorithms & Datastructures
Laboratory Exercise Sheet 5

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

April 21st, to be discussed on April 27th

Exercise 1) Complexity of sorting algorithms Consider a sorting algorithm A for some input sequence $a_1, a_2, \ldots, a_n$, which uses only comparisons between elements to gain order information about the input elements. That is, A may not inspect the values of the elements $a_1, \ldots a_n$ in any other way than performing key comparisons like $a_i < a_j$ on them.

Example: QuickSort fulfills this criterium, since it just compares $a[i], a[j]$ and the pivot element v. BucketSort does not, since it uses $a[i]$ for incrementing some counter c_{key}, choosing c_{key} based on the value of $a[i]$.

1. Prove that A has worst-case complexity $\Omega(n * log(n))$. Hints:

   - You may assume that all a_i are different.
   - You may also assume that all comparisons are of the form $a_i < a_j$ (in terms of the information you gain from such a comparison, the distinction between $<, \leq, \geq, >$ is irrelevant).
   - Have a look at Stirlings approximation, i.e. $n! \geq \sqrt{2\pi n} * (\frac{n}{e})^n$

Exercise 2) Quick-Select In the lecture, the pivot-function "median-of-medians" was explained for quick-select. Here, one had to (logically) divide the array into chunks of five, sort all these small arrays, and call quick-select recursively on the medians.

Here, the division of the large array into subarrays can be done in at least the following two ways (where a number i corresponds to the i-th sub-array that the array-position belongs to).

- sequential

0	0	0	0	0	1	1	1	1	1	2	2	2	2	2

- interleaved

0	1	2	0	1	2	0	1	2	0	1	2	0	1	2

1. Think about which alternative might be better to implement the pivot-function. Give your favorite and shortly explain your choice.
2. Implement the pivot-function and test it. (A complete implementation of quick-sort and quick-select is provided in the sources for this exercise-sheet, only the pivot-function is missing.)

Exercise 3) Heights of Binary Trees

1. Write an algorithm `public int height()` within the class `BinTree` to compute the height of a tree. (For the empty tree return -1.) What is the complexity of your algorithm?

2. Instead of computing the heights, one can also store the heights in the nodes. Then, the height of a tree can be determined in $\mathcal{O}(1)$. However, every modification of a tree also has to ensure that the correct heights are stored in each node. A corresponding datastructure is given in the classes `NodeH` and `BinTreeH`. (See sources for this exercise sheet.)

   - Write a method `public String toString()` that prints a tree where for each node the key and the height is displayed. For example, the tree of slide 135 might be displayed as follows where here, trees grow from left to right and every level of a tree is aligned to the same column, i.e. the root 10 of depth 4 is at the very left, all nodes on level 1 (4 and 15) are in one column, etc.

     ```
                 -2(2)
                                 2(0)
                         3(1)
           4(3)
                                 6(0)
                         7(1)
                 8(2)
                         9(0)
     10(4)
                 11(2)
                         12(1)
                                 13(0)
           15(3)
                 18(0)
     ```

     You can write any output where trees are displayed from left to right, or from top to bottom as it is done in the slides.

     Note that `+` can be used to concatenate strings, e.g. `"foo "+ 5 + "\nbar"` delivers the following string.

     ```
     foo 5
     bar
     ```

   - Modify the methods `put` and `remove` such that the heights are updated correctly. Does your modification change the asymptotic runtime of these two methods? Use the `toString`-method to test whether your modifications are correct.