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 1) Solution

1. Consider a decision tree for the problem. Each node represents one particular sort order, i.e. one permutation of $a_1, \ldots, a_n$. Each edge represents one comparison between exactly two elements a_i and a_j. A leaf represents the particular sort order which results, if all comparisons along the edges between the root and the leaf returned true. Thus, for one particular sequence $a_1, \ldots, a_n$, the algorithm returns the permutation (sort order) of the particular leaf with the property that all comparisons along the path between the root and the leaf returned true.

   As we have no additional knowledge about $a_1, \ldots, a_n$, each permutation of $1, \ldots, n$ may be returned by the algorithm, depending on the input $a_1, \ldots, a_n$. Thus, our decision tree contains one leaf for each permutation of $1, \ldots, n$, in total $n!$ leaves.

   A binary tree of height h has no more than 2^h leaves. Thus $n! \leq 2^h$. By taking logarithms, we get $h \geq log_2(n!)$, since the log_2 function is monotonically increasing. Stirlings approximation gives us $n! > (\frac{n}{e})^n$.

   Thus, $h \geq log_2((\frac{n}{e})^n) = n * log_2(n) - n * log_2(e)$. Finally, we get $h \in \Omega(n * log_2(n))$.

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 2) Solution The interleaved splitting is preferable, since after the sorting the medians are directly beside each other. (In contrast, for the sequential division the medians are not directly beside each other.) Of course, the sorting becomes more difficult, as for the interleaved division, the 5 values of each sub-array are not directly beside each other. Here, there are two solutions.

- copy the 5 values in an auxiliary array, sort it, and copy the sorted values back
- directly adapt the sorting algorithm such that it can deal with non-consecutive values

In the following code we used the latter approach.

Pivot function for median of medians

```
static int pivot(int[] a, int l, int r) {
   int n = (r - l + 1) / 5;
   if (n == 0) {
      return (l+1)/2 + r/2;
   } else {
      for (int k=0; k < n; k++) {
         // sort a[l + k + 0*n], a[l + k + 1*n], ... , a[l + k + 4*n]
         // with insertion sort
         for (int i=1; i<5; i++) {
            int j = l + k + i*n;
            int ai = a[j];
            while (j >= l + n) {
               if (a[j-n] > ai) {
                  a[j] = a[j-n];
               } else {
                  break;
               }
               j -= n;
            }
            a[j] = ai;
         }
      }
      int middle = l + 2*n + n / 2;
      quickSelect(a, l + 2*n, l + 3*n - 1, middle);
      return middle;
   }
}
```

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.

Exercise 3) Solution

1.
```
public class BinTree<D> {
   Node<D> root;
   public int height() {
      return height(this.root);
   }

   static <D> int height(Node<D> n) {
      if (n == null) {
         return -1;
      } else {
         int hl = height(n.left);
         int hr = height(n.right);
         return Math.max(hl, hr) + 1;
      }
   }
}
```

The costs of computing the height are $\mathcal{O}(n)$ where n is the number of nodes in the tree.

2.
```
1  public class BinTreeH<D> {
2     NodeH<D> root;
3
4     public void put(int key, D data) {
5        NodeH<D> node = find(key);
6        if (node == null) { // root is null
7           this.root = new NodeH<D>(key,data,null,null,null,0);
8        } else if (node.key == key) { // overwrite
9           node.data = data;
10       } else { // new node below node
11          NodeH<D> newNode = new NodeH<D>(key,data,null,null,node,0);
12          if (key < node.key) {
13             node.left = newNode;
14          } else {
15             node.right = newNode;
16          }
17          updateHeights(node);
18       }
19    }
20    static <D> NodeH<D> maximum(NodeH<D> node) {
21       while (node.right != null) {
22          node = node.right;
23       }
24       return node;
25    }
26    void removeNode(NodeH<D> node) {
27       if (node.left == null && node.right == null) {
28          if (node.parent == null) {
29             this.root = null;
30          } else {
31             if (node.key < node.parent.key) {
32                node.parent.left = null;
33             } else {
34                node.parent.right = null;
35             }
36             updateHeights(node.parent);
37          }
38       } else { // non-leaf
39          if (node.left == null || node.right == null) {
40             NodeH<D> child = node.left == null ? node.right : node.
                 left;
```

```
41              if (node.parent == null) { // delete root
42                 this.root = child;
43              } else if (node.key < node.parent.key) {
44                 node.parent.left = child;
45                 updateHeights(node.parent);
46              } else {
47                 node.parent.right = child;
48                 updateHeights(node.parent);
49              }
50              child.parent = node.parent;
51           } else { // two children
52              NodeH<D> maxLeft = maximum(node.left);
53              removeNode(maxLeft);
54              node.data = maxLeft.data;
55              node.key = maxLeft.key;
56           }
57        }
58     }
59
60     public void remove(int key) {
61        NodeH<D> node = find(key);
62        if (node != null && node.key == key) {
63           removeNode(node);
64        }
65     }
66
67     static <D> void updateHeights(NodeH<D> n) {
68        int hl = height(n.left);
69        int hr = height(n.right);
70        int h = Math.max(hl, hr) + 1;
71        if (h != n.height) {
72           n.height = h;
73           if (n.parent != null) {
74              updateHeights(n.parent);
75           }
76        }
77     }
78     public int height() {
79        return height(this.root);
80     }
81
82     static <D> int height(NodeH<D> n) {
83        return n == null ? -1 : n.height;
84     }
85
86     public String toString() {
87        return toString(this.root, "");
88     }
89
90     static <D> String toString(NodeH<D> n, String indent) {
91        if (n != null) {
92           String moreIndent = indent + "      ";
93           String s = toString(n.left, moreIndent);
94           s += "\n" + indent + n.key + "(" + n.height + ")";
95           s += toString(n.right, moreIndent);
96           return s;
97        } else {
98           return "";
99        }
100    }
101 }
```

The only changes to `put` and `remove` are the calls to `updateHeights` in lines 17, 36, 45, and 48. As `updateHeights` has costs $\mathcal{O}(h)$, the asymptotic runtime of both `put` and `remove` are still $\mathcal{O}(h)$ where h is the height of the tree.