Assignment 6 – Selected Model Answers

Exercise 1.

- **(i)** Implement a variant of linear probing without using __Placeholder objects. Instead use the alternative *delete* operation handled in the lecture to move another element into the freed space.
- **(ii)** Implement a variant of linear probing, where the size of the hash table is not m, but $m+m'$. In the search procedure in *insert*, *find* and *delete* increment the search index not considering the residue modulo m.

Exercise 2. The *delete* operation on binary search trees presented in the lectures merges the two successor trees, when an element labelling a non-leaf vertex is deleted. One might as well re-insert all the elements.

- **(i)** Implement this alternative for the implementation of *delete*.
- **(ii)** Discuss, why this idea should not be not recommended.

Solution. We only look at part (ii).

When re-inserting one-by-one all elements in the two subtrees of a deleted node, we perform an *insert* operation m-times, where m is the total number of nodes in the two subtrees. In the worst case we may even have $m = n - 1$, so the complexity will be in $O(n \log n)$, which is unacceptable. Instead swapping the deleted element with its direct predecessor, i.e. the largest element in the left successor tree, or with its immediate successor, i.e. the smallest element in the right successor tree, requires only a search for this predecessor or successor, which in the worst case may take time in $O(n)$, plus the actual swapping, which is done in constant time.

Furthermore, if the elements in the successor trees are taken in the usual depth-first order, they will be inserted in an ordered way, which leads to a generated BST, i.e. the worst possible case.

Exercise 3. Implement recursive *insert* and *delete* operations on AVL trees and compare the performance with the iterative implementations.

Solution. In the iterative algorithm for *insert* we need to search, whether there exists already a node with the value x that is to be inserted. Using a *while*-loop we iterate until we encounter a null pointer, always updating the current node by the left successor in case x is smaller than the stored value or by the right successor in case x is larger than the stored value. In case of equality we stop leaving the AVL tree unaltered.

We need to keep track of the path from the root followed by the search. For this we can keep a stack, i.e. we always push a visited tree node onto the stack. Alternatively, we can change the definition of a node by adding an additional *parent* pointer, which is null for the root. With parent pointers we can reconstruct the followed path.

Once we reach a null pointer, a new leaf node with the value x and null pointers for the left and right successors is created. Then we need to iterate backwards using another *while*-loop, each time popping the top element from the stack until this becomes empty. With the node taken from the stack, the previously examined node, i.e. the bad child, and the previously examined grandchild (if not null) we proceed in each step in exactly the same way as in the recursive version. That is, we update the balance, if the new balance remains between -1 and 1, or we perform single or double rotations. If the new balance without or after the rotations is 0, we can stop, otherwise we continue with the next element from the stack.

For the iterative algorithm for *delete* we proceed analogously. First we iterate until we encounter a node, in which the element x to be deleted is stored. If no such node exists, we can stop leaving the AVL tree unaltered. Same as for *insert* we use a stack to keep track of the path from the root followed by the search, i.e. we always push a visited tree node onto the stack.

Once a node with x is found, we search for the direct predecessor, i.e. the largest element in the left successor tree, unless this tree is empty. We only need to follow right pointers starting from the root of the left sucessor tree. The search ends, when the right successor becomes empty. We keep pushing nodes onto the stack. When the direct predecessor is found, it is stored in the node, the search for it started, and its left successor replaces the node with the predecessor.

Then the backward procedure is executed using another *while*-loop, each time popping the top element from the stack until this becomes empty. With the node taken from the stack, the bad child and the bad grandchild are determined in exactly the same way as in the recursive version. If necessary a single or a double rotation is executed; otherwise only the balance is updated. This is done in exactly the same way as in the recursive version. If the new balance without or after the rotations is different from 0, we can stop, otherwise we continue with the next element from the stack.

The main difference to the recursive version is the use of an additional stack (or the use of parent pointers). The overhead created by this compares for the overhead arising from recursive calls, but differences should be marginal. The core of the algorithms, downward search followed by upward propagation of balances including rotations, if necessary, is the same.

EXERCISE 4. The *median* M of a set $\{a_0, \dots, a_n\} \subseteq T$, where T is a totally ordered set, is the smallest $M = a_i$ such that less than $(n+1)/2$ elements a_j are smaller than M. If $n = 2k$ is even or $n = 2k+1$ is odd and $a_i \leq a_j$ holds for $i < j$, we have $M = a_k$. So the median is the k'th smallest element among the a_i for $k = \lfloor n/2 \rfloor$.

In the literature you sometimes find the notion *low median* for this element, denoted as M^-, whereas the ℓ'th smallest element M^+ for $\ell = \lceil n/2 \rceil$ is called the *high median*. The median is then defined as the arithmetic mean $(M^- + M^+)/2$. Note that for even n the low and high median coincide.

Define that an AVL tree satisfies the *median property* iff for all vertices v the label $\ell(v)$, i.e.

the element stored in node v, is either the median M of the set of all elements in the AVL subtree rooted at v or the high median M^+ of this set.

- **(i)** Show that if an AVL tree satisfies the median property, then it is perfectly balanced.
- **(ii)** Define and implement a modified *insert* operation that guarantees that the resulting AVL tree satisfies the median property.
- **(iii)** Define and implement a modified *delete* operation that guarantees that the resulting AVL tree satisfies the median property.
- **(iv)** Determine the worst case complexity for these modified *insert* and *delete* operations.

SOLUTION.

(i) Use induction over the number n of elements stored in the tree. For $n = 1$ we only have the root, which must contain the only element, which is the median. We have the height $0 = \log_2 1$.

Next assume that all trees with median property with at most $n \geq 1$ elements are balanced. If we have $n + 1$ elements, then the median property implies that the two successor trees have $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ elements, respectively. Using the induction hypothesis the height of the tree is

$$\lfloor \log_2 \lceil n/2 \rceil \rfloor + 1 = \lfloor \log_2 (2 \cdot \lceil n/2 \rceil) \rfloor = \lfloor \log_2 (n+1) \rfloor \ .$$

(ii) In the following we use $m = \frac{M^- + M^+}{2}$.

The idea for an *insert* operation is straightforward as follows. Instead of the balance store the median m of the elements in the subtree rooted at x with the element x. If we insert a new element y, we have three cases:

$x < m$. In this case we have $m = \frac{x+x'}{2}$ for some x' stored in the tree, so we have $x' = 2x - m$. In case $x < y < x'$ the new median is $m' = y$, so we store (y, m') in the node and insert x into the left successor tree. In case $y < x < x'$ the new median is $m' = x$, so we store (x, m') in the node and insert y into the left successor tree. In case $x < x' < y$ the new median is $m' = x'$, so we store (x', m') in the node, insert x into the left successor tree, and insert y into the right successor tree.

$x > m$. In this case we have $m = \frac{x+x'}{2}$ for some x' stored in the tree, so we have again $x' = 2x - m$. In case $x > y > x'$ the new median is $m' = y$, so we store (y, m') in the node and insert x into the right successor tree. In case $y > x > x'$ the new median is $m' = x$, so we store (x, m') in the node and insert y into the right successor tree. In case $x > x' > y$ the new median is $m' = x'$, so we store (x', m') in the node, insert x into the right successor tree, and insert y into the left successor tree.

$x = m$. In case $y < x$ we determine the largest element z in the left successor tree, if this is $> y$ or otherwise $z = y$. The new median is $m' = \frac{x+z}{2}$, so we store (x, m') in the node and insert y into the left successor tree. In case $y > x$ we determine the smallest element z in the right successor tree, if this is $< y$ or otherwise $z = y$. The new median is $m' = \frac{x+z}{2}$, so we store (x, m') in the node and insert y into the right successor tree.

The implementation is straightforward.

(iii) If we are to *delete* an element y, we first search for y in the usual way until we find a node with label (y, m). Once the search has found the node with label (y, m) we have again three cases:

$y < m$. In this case $m = \frac{x+y}{2}$ with $x = 2m - y$. Search for the smallest element z in the right successor tree. Then the new median is $m' = z$, so we store (z, m') in the node and delete z in the right successor tree.

$y > m$. In this case $m = \frac{x+y}{2}$ with $x = 2m - y$. Search for the largest element z in the left successor tree. Then the new median is $m' = z$, so we store (z, m') in the node and delete z in the left successor tree.

$y = m$. In this case search for the largest element x in the left successor tree and the smallest element z in the right successor tree. Then the new median is $m' = \frac{x+z}{2}$, so we either store (x, m') in the node and delete x in the left successor tree, or we store (z, m') in the node and delete z in the right successor tree.

Once the delete has been done we proceed backwards to the root and adjust all node labels. We have three cases:

$x < m$. If the delete was done in the right successor tree, i.e. we had $y > x$, then the new median is x, and we change the label to (x, x). If the delete was done in the left successor tree, i.e. we had $y < x$, then the new median is $x' = 2m - x$, and we update the label to (x', x').

$x > m$. If the delete was done in the left successor tree, i.e. we had $y < x$, then the new median is x, and we change the label to (x, x). If the delete was done in the right successor tree, i.e. we had $y > x$, then the new median is $x' = 2m - x$, and we update the label to (x', x').

$x = m$. If the delete was done in the left successor tree, then determine the smallest element x' in the right successor tree. Determine the new median $m' = \frac{x+x'}{2}$ and store (x, m') in the node. If the delete was done in the right successor tree, then determine the largest element x' in the left successor tree. Determine the new median $m' = \frac{x+x'}{2}$ and store (x, m') in the node.

Again the implementation is straightforward.

(iv) For the *insert* operation the worst case occurs, if for every insertion into a subtree we have to first search for the largest or smallest element in the left or right successor tree, respectively. The complexity of this search is linear in the height of the subtree, i.e. it is in $O(\log_2 i)$. Hence the time complexity of *insert* is in

$$O(\sum_{i=1}^{\log_2 n} \log_2 i) = O(\log_2(\prod_{i=1}^{\log_2 n} i)) = O(\log_2(\log_2 n)!) \subseteq O((\log n)^2) \ .$$

For the *delete* operation also the worst case occurs, if for every node on a path to a leaf we have to search for the largest or smallest element in the left or right successor tree, respectively. Using the same argument the time complexity of *delete* is also in $O((\log n)^2)$.