DM22 Programming Languages

Spring 2005

Project 1

Department of Mathematics and Computer Science
University of Southern Denmark

March 7, 2005

The purpose of this project is to implement in Haskell a dictionary structure based on AVL-trees, and to implement, using this dictionary, an algorithm for a problem in computational geometry. The project is to be done in groups of up to two persons.

Dictionaries

An ordered dictionary is an abstract datatype storing elements of type (k, x), where the key k is from some ordered set (e.g. the real numbers $\mathbb{R}$) and the data x is from some arbitrary set.

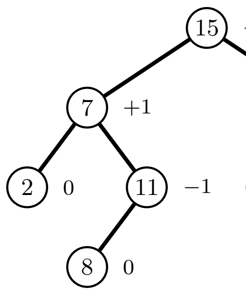
It offers the following operations:

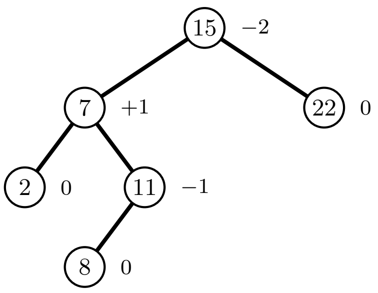
`makeDict()`	Create and return an empty dictionary.
`isEmpty(`D`)`	Return True iff the dictionary D is empty.
`search(`k, D`)`	Return the element (k, x) in dictionary D, if present.
`insert(`k, x, D`)`	Insert element (k, x) in dictionary D. returning the updated dictionary.
`delete(`k, D`)`	Delete (if present) the element (k, x), returning (k, x) and the updated dictionary.
`rangeSearch(`k_1, k_2, D`)`	Return the set of (k, x)'s in D where $k_1 \leq k \leq k_2$.

For simplicity, we assume that each key k is present only once in the dictionary.

AVL-trees

An AVL-tree is a type of balanced binary search tree (in fact, the first type of balanced search tree, invented in 1962 by Adel'son-Vel'skiĭ and Landis). The balance criterion is that for each node v in the tree, the heights of the two subtrees of v differ by at most one. It can be proven that the height of such a tree is logarithmic—more precisely, that the height is bounded by $\log_\varphi(n+1) = c \cdot \log_2(n+1)$, where n is the number of internal nodes, $\varphi = (\sqrt{5}+1)/2 \approx 1.618$ is the golden ratio, and c is $1/\log_2 \varphi \approx 1.44$. In the following, we let $\text{bal}(v)$ denote the height of the right subtree of v minus the height of the left subtree of v. The balance criterion for AVL-trees can be expressed as $|\text{bal}(v)| \leq 1$ for all nodes v in the tree. Below are shown two search trees annotated with balance information. External nodes, i.e. subtrees which are empty, are not shown. The tree on the left is an AVL-tree, whereas the tree on the right is not.

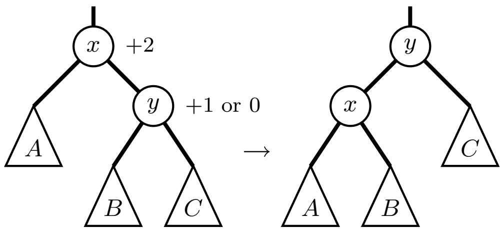


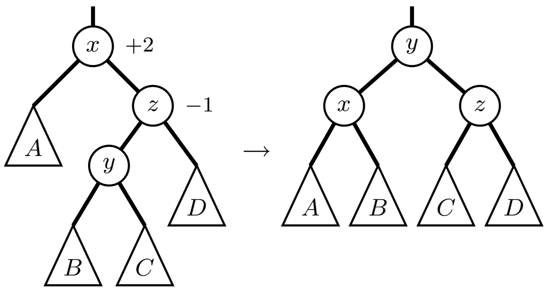


Searching in any search tree is a well-known recursive process based on the search tree invariant that all keys in the left (right) subtree of a node v are smaller (larger) than the key in v. Range searches are performed by a similar recursive process, but now recursing on both children of v if the key of v is contained in the search interval.

Insertions and deletions in AVL-trees are performed as usual in search trees (see Section 16.7 in the textbook). After an insertion or deletion, the heights of subtrees rooted at nodes on the path from the root to the node which was deleted (which may be the node containing the key `mini` in the code on page 318 in textbook) have possibly changed by one. Hence, nodes v in this path may now have $|\text{bal}(v)| = 2$, which will violate the balance criterion. Nodes outside this path clearly change neither height nor balance.

The balance criterion is restored in a bottom-up fashion along this path. For an unbalanced node v on this path, one of the two transformations (denoted a single rotation and a double rotation) shown below is performed.





The choice between the two transformations depends on the balance of the tallest child of the unbalanced node v, as indicated on the figure. Symmetric cases (reflected in a vertical line) of the transformations exist, in which all balance values shown have changed sign.

It can be shown that these transformations will restore the balance at v, given that all unbalance below it has been handled. After v, the process continues with the parent of v, etc., until the root is reached.

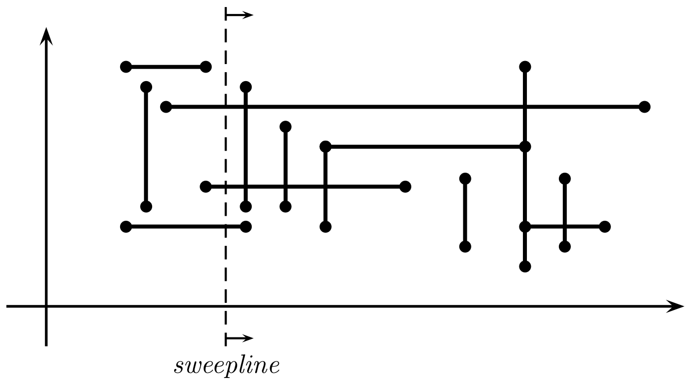
Each node in the AVL-tree will keep an integer containing its height. Note that the height of a node can be found from the heights of its two children. From this, the new heights of nodes on the path can be found during the bottom-up rebalancing process, from which the rebalancing cases (if any) can be recognized.

Task 1

Implement in Haskell a module `Dictionary` supporting the operations above. The implementation should be based on AVL-trees. For testing purposes, your module should contain procedures (not exported) for checking the search tree invariant and for checking the AVL balance criterion. The first is implemented by doing an in-order traversal of the tree and verifying that the keys are met in sorted order. The second is implemented by doing a post-order traversal, verifying that for all nodes v, the height stored is correct and $|\mathrm{bal}(v)|$ is at most one.

Orthogonal Line Segment Intersection

The problem of orthogonal line segment intersection finding is for two sets A and B of line segments, where those in A are horizontal, and those in B are vertical, to find all pairs $(a, b) \in A \times B$ for which a and b intersect. For simplicity, we assume that all line segments are finite and closed (the endpoints are included in segments), and that no two segments in A touch each other. The figure below shows an instance where the result has size 9.



One algorithm for the problem is based on the sweep-line technique: Think of the x-axis as a time line and consider a vertical line sweeping from left to right. When a left endpoint of a line segment in A is met, the line segment is inserted into a dictionary D with the y-value of the line segment as key. When a right endpoint of a line segment in A is met, the corresponding entry is deleted from D. When a line segment l from B is met, a range search in D is performed with the y-values of the endpoints of l giving the search interval. The result of this range search is exactly the line segments of A that intersects l.

The "points in time" (i.e. the x-values) of interest are therefore the x-values of the left and right endpoints of line segments in A, and the x-values of the line segments in B. The algorithm starts by making a list of these objects, sorted on their x-values. Sweeping the line from left to right can now be implemented as a traversal of this list.

Task 2

Implement in Haskell a module `Intersection` which implements the algorithm above. The module should support a single function which takes as input two lists of line segments (the first representing A, the second representing B) and returns a list of pairs of segment identifiers. The following self-explanatory type definitions should be used for A and B:

```
data HSegment = HSeg SegID MinX MaxX Y
data VSegment = VSeg SegID MinY MaxY X
type SegID = String
type MinX = Float
type MaxX = Float
type MinY = Float
type MaxY = Float
type Y = Float
type X = Float
```

Formalities

A printed report of around five to seven pages should be handed in. Haskell code and any test data should be given as appendices. The main aim of the report should be to describe the modeling and program design choices made during development, the reasoning behind these choices, and the structure of the final solution. You should state the running time of all operations exported by your modules. A copy of the Haskell code should be mailed to the lecturer at `rolf@imada.sdu.dk` as a `.tgz` or `.zip` file. Remember to state the name of the group members.

You must hand in the report and the code by

Monday, April 11, 2005