DM22 Programming Languages

Spring 2006

Project 1

Department of Mathematics and Computer Science
University of Southern Denmark

March 20, 2006

The purpose of this project is to implement in Haskell a dictionary based on a specific type of balanced search trees, and to implement, using this dictionary, an algorithm for a problem in computational geometry. The project is to be done in groups of two or three persons.

Dictionaries

An ordered dictionary is an abstract datatype storing elements (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:

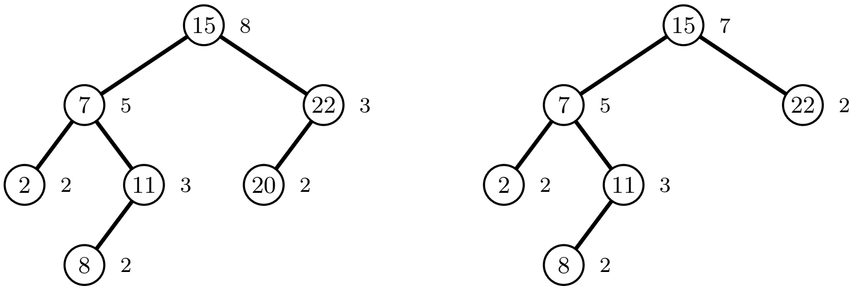
`makeDict()`	Create and return an empty dictionary.
`isEmpty(`D`)`	Return True iff the dictionary D is empty.
`search(`k, D`)`	Return the element (k, x) in dictionary D, if present.
`insert(`k, x, D`)`	Insert element (k, x) in dictionary D. returning the updated dictionary.
`delete(`k, D`)`	Delete (if 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.

Balanced Search Trees

In this project, the dictionaries should be implemented using a particular (simple) type of balanced binary search trees, which technically carries the rather long name *locally rebuilt weight-balanced trees.*

The *weight* of a node is the number of leaves in its subtree. A binary search tree is weight-balanced if for all internal nodes, the weight of any of its two subtrees is at most twice the weight of the other. It is easily proven[1] that the height of such a tree is logarithmic. Below are shown two search trees annotated with weight information. Leaves are not depicted. The tree on the left is a weight-balanced tree, 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. If the key is not found in v, the search recurses on the proper subtree. performed by a similar recursive process, but now recursing on *both* subtrees of v if the key of v is contained in the search interval (and in this case also reporting the element of v as part of the output).

Insertions and deletions in weight-balanced trees are performed as usual[2] in binary search trees. After an insertion or deletion, the balance of nodes on the path from the root to the node which was inserted or deleted[3] have changed, and could now be violated. Rebalancing is now performed by rebuilding to perfect balance the subtree rooted at the highest unbalanced node on this path. Here, perfect balance means that for all nodes, the weights of its two subtrees differ by at most one. Clearly, this will restore balance in the search tree after an update. If the rebuilding takes time linear in the size of the subtree rebuilt, it can be proven that rebalancing in this way takes amortized logarithmic time per update.

To discover unbalance (and the node on the search path whose subtree should be rebuilt), each internal node in the weight-balanced tree keeps an integer containing its weight. It of course also must keep a search key, and any data associated with the key.

Task 1

Implement in Haskell a module `Dictionary` supporting the operations above, where the exact Haskell types of the dictionary operations are part of your design choices.

The implementation should be based on the search trees described. The operations `makeDict` and `isEmpty` should take constant time, and `search` should take logarithmic time. The update operations `insert` and `delete` should take amortized logarithmic time, and for this you must

[1] Leaves have height zero and weight one. For each step upwards on the path to the root, the weight is multiplied by at least 3/2. Hence, a tree of height h has at least $(3/2)^h$ leaves, so the number n of internal nodes is at least $(3/2)^h - 1$. Taking logarithms gives $h \leq \log_{3/2}(n+1)$.

[2] See Section 6.2 in the textbook, or (even better) the slides from the lecture Tuesday, March 21.

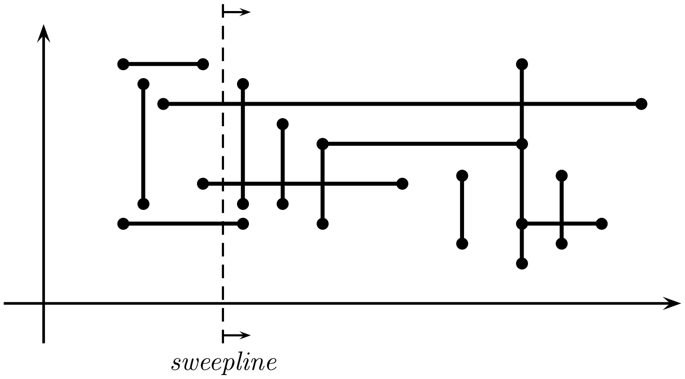
[3] In case of deletion, this should be the node physically deleted, which may be the node found using `splitBSTree` in the code on the slides (or `splitTree` in Section 6.2).

develop a linear time algorithm for rebuilding a tree to perfect balance.[4] The `rangeSearch` query should take time $O(k + \log n)$, where k is the size of the output.[5]

For testing purposes during development, your module should contain procedures (not exported) for checking the search tree invariant and the weight balance criterion in the entire tree.

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 (i.e. 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

[4] Hint: Code on slides (or from Chapter 7 in book) could be inspirational.

[5] Hint: Use the accumulator technique for collecting the output.

(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
```

The entire algorithm should run in time $O(k + n \log n)$, where k is the size of the output (the number of pairs of segments intersecting) and n is the total number of segments. To achieve this, an $O(n \log n)$ time sorting algorithm must be used.[6]

Formalities

A printed report of around seven pages should be handed in. The Haskell code (which must be sufficiently commented) and reasonable 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. Central Haskell code snippets should be included in the text of the report.

A copy of the Haskell code should be handed in using the `aflever` command on the Imada system: Move to the directory containing your code and issue the command `aflever DM22`. This will copy the contents of the directory to a place accessible by the lecturer. Repeated use of the command is possible (later uses overwrite the contents from earlier uses). In the directory, you must for identification purposes have an ASCII file named `names.txt` containing the names of the group members, with one name per line.

You must hand in the report and the code by

Tuesday, April 25, 2006

[6] Hint: Mergesort is one simple solution. Caveat: The one page 148 in the book is erroneous, and the errata should be consulted. Consider making a datatype for endpoints/points in time (for use during the sweepline algorithm), and making it an instance of the class Ord. Then the sorting can be based on the built-in comparison operators such as `<=` (i.e. no need for `f` on page 148, so mergesort just gets simpler).