Part I

A self-adjusting data structure – Splay Trees

In Datalogi 1, a number of different data structures for dictionaries are presented. Among these are balanced search trees which support execution of the operations *insert*, *member* and *delete* in worst case time $O(\log n)$, where n is the number of elements stored in the structure. To obtain worst case time $O(\log n)$, different measures were used to measure to what extent a tree is balanced. If an operation created an unbalanced tree, a rebalancing operation (typically a rotation (Figure 1)) was performed.

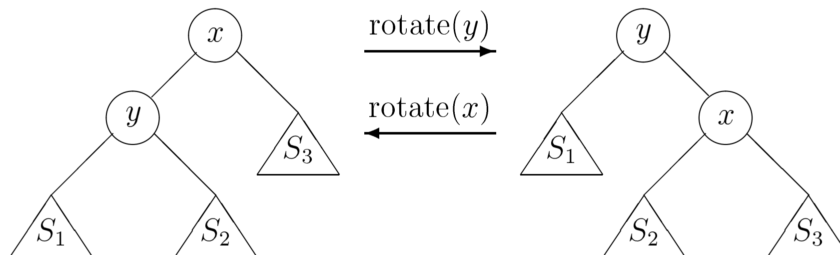

Figure 1: Rotations

Unlike these standard tree structures, *splay trees* use no explicit rebalancing. The rebalancing happens automatically as an integrated part of the operations.

1 Splay Trees

Splay trees do not support execution of the operations in worst case time $O(\log n)$ but in *amortised* time $O(\log n)$. That is, a particular operation

might take more than $O(\log n)$ time, but if a sequence of operations are performed on a set of initially empty trees, then the average time per operation is at most $O(\log n)$.

Splay trees are a competitive alternative to balanced trees if we are to perform a whole sequence of operations and do not mind that a single operation might be slow. Furthermore, although balanced trees are conceptually simple, splay trees are simpler to implement in practice.

A splay tree (over an ordered set of elements E) is an ordinary binary search tree. In a node x there are three fields: *item*(x) (the element stored in x), *left*(x) and *right*(x) (pointers to left and right subtrees of x). All elements (stored) in the left subtree of x are smaller than *item*(x) and all elements of the right subtree are greater than *item*(x). The tree is accessed and identified by a pointer to the root.

Table 1 contains a list of some of the operation which are easy to implement using splay trees.

Each operation from Table 1 can be implemented using a constant number of *splay* operations in addition to a constant number of simple operations, such as pointer manipulations and comparisons. The following operation is the core operation for splay trees.

- *splay*(S, e) Reorganises the tree S and returns the resulting tree. If S is nonempty, the element stored in the root of the resulting tree is either $\max\{t \text{ in } S \mid t \leq e\}$ or $\min\{t \text{ in } S \mid t \geq e\}$.

Having the *splay* operation at hand, the other operations are easy to implement. We sketch the implementations of *insert*, *join* and *deletemin*. Implementations of the remaining operations is left for Exercise 1.1.

insert(S, e): If S is empty, then create a tree with one node containing e. If S is non-empty then call *splay*(S, e) obtaining a tree with root r. If *item*(r) $= e$ then we are done. Otherwise, if $e <$ *item*(r) then create a new root x containing e and rearrange as shown in Figure 2. The case where $e >$ *item*(r)

• *init*(S)	Returns an empty tree S.
• *insert*(S, e)	Inserts element e into the tree S, and returns the resulting tree.
• *delete*(S, e)	Deletes element e from tree S if it is there, and returns the resulting tree.
• *access*(S, e)	Returns a pointer to the node in S storing e if it exists and returns **nil** otherwise.
• *join*(S_1, S_2)	Returns a tree storing the elements in S_1 followed by the elements in S_2. S_1 and S_2 are destroyed. (It is assumed that all elements in S_1 are smaller than all elements in S_2.))
• *split*(S, e)	Returns two trees S_1 and S_2, where S_1 contains all elements in S less than or equal to e, and S_2 contains all elements in S greater than e.
• *min*(S)	Returns the minimal element in S if S is nonempty, otherwise returns **nil**.
• *deletemin*(S)	Deletes the minimal element in S if S is nonempty and returns the resulting tree.

Table 1: Operations supported by splay trees

is done analogously.

join(S_1, S_2): If S_1 is empty then return S_2. Otherwise call *splay*(S_1, ∞). This reorganises S_1 into a tree where the root r contains the largest element stored in S_1 and has no right subtree. Now make S_2 the right subtree of r (Figure 3).

deletemin(S): If S is empty, we are done. Otherwise call *splay*($S, -\infty$). Return the right subtree of the root. Note that *splay*($S, -\infty$) reorganises the tree such that the minimal element is moved to the root (Figure 4).

Exercise 1.1 *Show how to implement the operations listed in Section 1 using splay trees.*

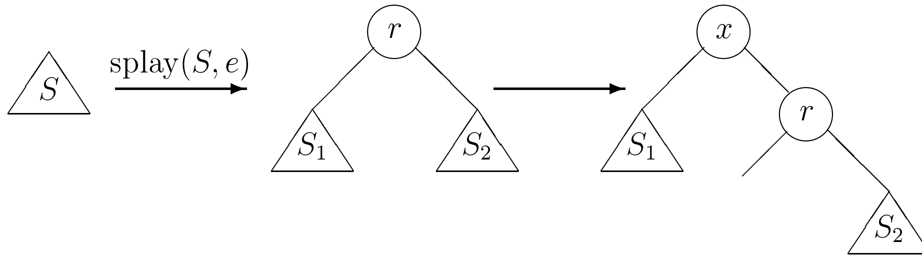

Figure 2: *insert*(S, e)

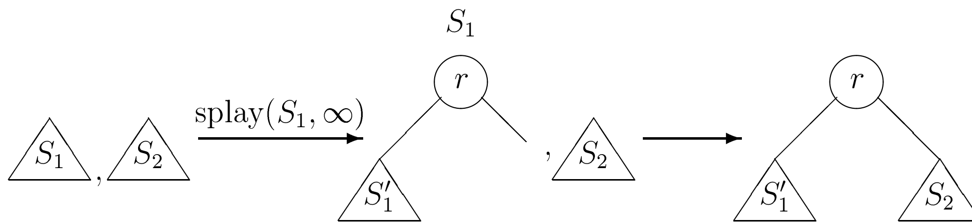

Figure 3: *join*(S_1, S_2)

1.1 Implementation of *splay*

We use rotations (Figure 1) to move nodes towards the root.

If x is a son of y then *rotate*(x) moves x up and y down and changes a few pointers. To move a specific node x to the root of the tree, we could repeatedly call *rotate*(x) until x becomes the root. However, to achieve good time bounds we implement *splay*(S, e) in a more perfected manner:

First search for e in S. Three cases may occur.

- e is in S and we find the node x where *item*$(x) = e$.
- e is not in S and we end up in a node x with no left subtree and $e <$ *item*(x). Then *item*$(x) = \min\{t \text{ in } S \mid t \geq e\}$.
- e is not in S and we end up in a node x with no right subtree and

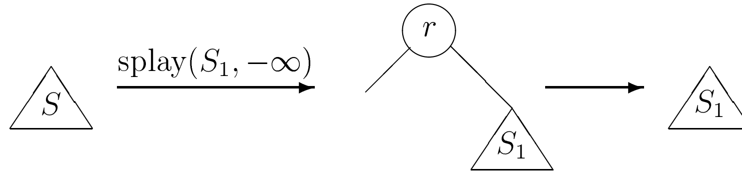

Figure 4: *deletemin*

$e > item(x)$. Then $item(x) = \max\{t \text{ in } S \mid t \leq e\}$.

In all three cases we have determined the node x, which is to be moved to the root.

x is moved to the root by applying a number of *macro steps* until x has become the root of the tree. If x is a child of the root, one macro step moves x up one level making it the root of the tree. Otherwise, it moves x up two levels in the tree by performing two rotations.

There are three types of macro steps:

1. $rotate(x)$, if x is a child of the root.
2. $rotate(y)$ followed by $rotate(x)$, if x has a parent y and a grandparent z, and x and y are both right children or both left children.
3. $rotate(x)$ followed by $rotate(x)$ again, if x has a parent y and a grandparent z, and one of x and y is a left child and the other a right child.

See Figures 5 and 6 for examples.

Exercise 1.2 *a) Perform* $\text{insert}(S_1, 5)$ *and show the resulting tree.*

b) Perform $\text{join}(S_1, S_2)$ *and show the resulting tree.*

c) Perform $(S_4, S_5) := \text{split}(S_3, 4)$ *followed by* $\text{join}(S_4, S_5)$ *and show the resulting tree.*

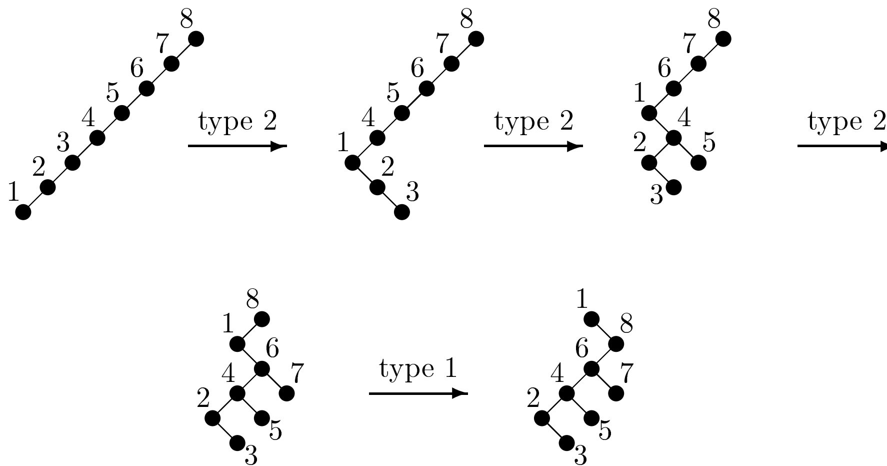

Figure 5: Example. $splay(S, 1)$

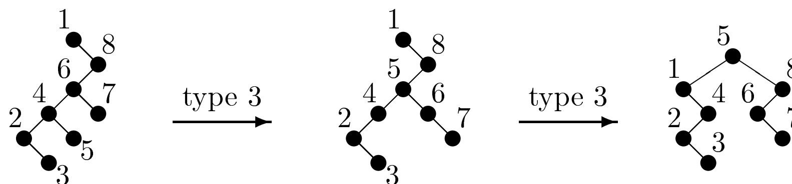

Figure 6: Example. $splay(S, 5)$

1.2 Analysis

We will show that a sequence of m operations takes time $\mathrm{O}(m \log n)$, where n is the the maximal number of elements ever stored in the set of trees.

We are to show that on average $\mathrm{O}(\log n)$ time suffices for the operations in the the sequence.

This is done by using a credit accounting scheme like the one you have seen for a binary counter in Datalogi 1.

We charge a certain amount of time for an operation. If it is more than

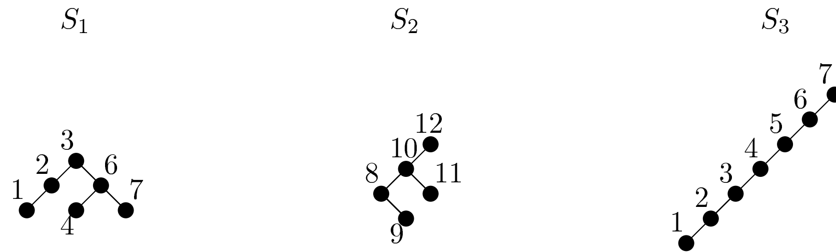

Figure 7: Trees used in Exercise 1.2

needed to perform the operation we deposit the extra time as credits on various accounts associated to the nodes in the trees. If the amount charged is insufficient we spend some of the credits saved on accounts to take care of the difference.

The unit we will use is **ECU** (Enough Credit Unit). The value of one **ECU** will be set by demand later.

We do not restrict ourselves to integer deposits but use the real values for convenience.

For the node x in a tree, let $S(x)$ be the subtree rooted at x. Let $|S|$ denote the number of nodes in tree S. For a tree S of size at least 1, let $\mu(S) = \log(|S|)$. Finally let $\mu(x) = \mu(S(x))$.

We show that we can deposit the surplus at all times such that the following invariant is maintained:

Any node x in any tree has at least $\mu(x)$ credits on the account associated to it.

Lemma 1.1 *For a node x in a splay tree S it suffices to charge $3(\mu(S) - \mu(x)) + 1$ **ECU** to the operation splay(S,item(x)) to perform the operation and to maintain the invariant.*

Proof Let $r = x_1, x_2, \ldots, x_k = x$ be the search path for *item*(x) in S. To perform the splay operation we do k comparisons and some bookkeeping

$k-1$ times to do the search and finally $\lfloor k/2 \rfloor$ macro steps. We set the exchange rate for **ECU** such that one **ECU** can pay for two rotations, two comparisons and the (constant) amount of bookkeeping for two steps in the search.

We show by induction on the depth of x that $3(\mu(S)-\mu(x))+1$ **ECU** suffice to pay for the search, the $\lfloor k/2 \rfloor$ macro steps and to maintain the invariant.

Basis. $k \leq 2$. Assume first that $k=1$. Then x is the root of S and no reorganisation takes place. Therefore the invariant remains true. The one comparison involved requires less than one **ECU** to be paid. We are willing to pay $3(\mu(S)-\mu(x))+1=1$, so we are done.

If $k=2$ we do a macro step of type 1, namely rotate(x). One **ECU** suffices for the operation. To maintain the invariant we must pay

$$\mu'(x)+\mu'(r)-\mu(x)-\mu(r)=\mu'(r)-\mu(x)$$

where μ and μ' refer to the values of μ before and after doing *rotate*(x). Note that $\mu'(r)-\mu(x)$ might be negative.

Since

$$3(\mu(S)-\mu(x))+1=3(\mu(r)-\mu(x))+1>\mu'(r)-\mu(x)+1,$$

we are done again.

Induction step. Assume that the the search path is $r=x_1,x_2,\ldots,x_k(=z),y,x$ and the claim holds for k. The first macro step moves x to the location of z. Observe, that the size of the subtree with root z before this step is the same as the size of the subtree with root x after the step is taken. By the induction hypothesis $3(\mu(S)-\mu(z))+1$ **ECU** suffice to pay for the remaining macro steps and to maintain the invariant during those steps. This means that the amount of **ECU** we can allow to pay for the first macro step and to maintain the invariant during this step is

$$3(\mu(S)-\mu(x))+1-(3(\mu(S)-\mu(z))+1)=3(\mu(z)-\mu(x)).$$

In the following we prove that this is indeed sufficient.

The first macro step is either of type 2 or 3 (see page 8). Consider type 2 first (Figure 8). We do a *rotate*(y) followed by a *rotate*(x).

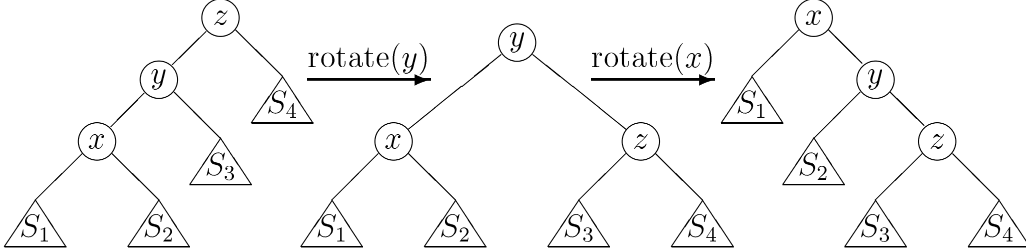

Figure 8: Type 2 macro step

Let μ and μ' refer to the values of μ before and after performing *rotate*(y) and *rotate*(x). We must pay

$$\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) = \mu'(y) + \mu'(z) - \mu(x) - \mu(y)$$

ECU to maintain the invariant and one **ECU** to do the rotations, comparisons, and bookkeeping.

We can allow to pay $3(\mu(z) - \mu(x)) = 3(\mu'(x) - \mu(x))$ so we must verify that

$$3(\mu'(x) - \mu(x)) \geq \mu'(y) + \mu'(z) - \mu(x) - \mu(y) + 1$$

or

$$3\mu'(x) - \mu'(y) - \mu'(z) - 2\mu(x) + \mu(y) \geq 1$$

From the facts $\mu(y) > \mu(x)$, $\mu'(x) > \mu'(y)$ and Fact 1.2 (from the Appendix) we get

$$\begin{aligned} & 3\mu'(x) - \mu'(y) - \mu'(z) - 2\mu(x) + \mu(y) > 2\mu'(x) - \mu'(z) - \mu(x) \\ = \; & 2\log(|S_1| + |S_2| + |S_3| + |S_4| + 3) - \log(|S_3| + |S_4| + 1) \\ & - \log(|S_1| + |S_2| + 1) > 1 \end{aligned}$$

which was what we wanted.

The proof is similar if the first macro step is of type 3 (Figure 9). This is the subject of Exercise 1.3.

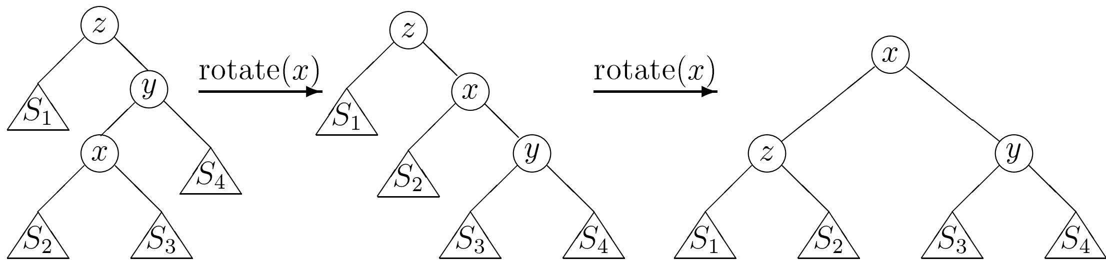

Figure 9: Type 3 macro step

□

Exercise 1.3 *Finish the proof of Lemma 1.1.*

Theorem 1.1 *Starting with a set of empty trees, a sequence of m operations from Table 1 can be executed in worst case time* $O(m \log n)$ *where n is the maximal number of elements stored in the trees at any time.*

Proof It suffices to check for each operation *init*, *insert*, ... in turn and verify that $O(\log n)$ **ECU** 's are sufficient to pay for the operation and to maintain the invariant at page 10. For *insert* we argue as follows. See Figure 2 for notation.
$O(\log n)$ is sufficient to pay for *splay*(S, e) and maintaining the invariant during the splay operation. If e is not in S we add a new node x. All nodes but the new node x are associated adequate credit (the demand on r might even decrease), but node x has no credit initially. Therefore the invariant is violated. $\mu(x) = O(\log n)$ is needed for node x. Altogether $O(\log n)$ **ECU** is sufficient. So the amortised time for *insert* is $O(\log n)$. We leave the details for the remaining operations to Exercise 1.4.

Exercise 1.4 *Finish the proof of Theorem 1.1.*

Exercise 1.5 *Let r be the root of a splay tree S. Find the maximal depth of r after performing one of the operations in Section 1. It is assumed that r also exists in the resulting tree(S).*

References

[1] Sleator, D.D. & Tarjan, R.E.: Self-Adjusting Binary Trees. *ACM STOC 1983 pp 235-245*.

[2] Kozen, D.C.: *The Design and Analysis of Algorithms*, Lecture 12. Springer-Verlag 1992.

1.3 Problems

Problem 1.1 *Let* $t_{fixed}(S_f, \sigma)$ *be the time for performing a sequence* σ *of* member*-operations on a fixed binary search tree* S_f *storing* n *elements. Let* S *be an arbitrary binary search/splay tree storing the same elements and let* $t_{splay}(S, \sigma)$ *be the time for performing the sequence* σ *when splay operations are used to the implementation.*

Prove that, $t_{splay}(S, \sigma) = \mathrm{O}(t_{fixed}(S_f, \sigma) + n^2)$.

Note that given σ, S_f *can be chosen to minimise* $t_{fixed}(S_f, \sigma)$*. So the statement express that up to a constant splay trees adjust themselves to the optimal with respect to a sequence of* member*-operations. The learning time is* $\mathrm{O}(n^2)$.

1.4 Appendix

Fact 1.2 *For* $a, b > 0: \ 2\log(a+b) - \log(a) - \log(b) > 1.$

Proof

$$
\begin{aligned}
& 2\log(a+b) - \log(a) - \log(b) \\
= \ & \log\left(\frac{(a+b)^2}{ab}\right) \\
= \ & \log\left(\frac{a^2+b^2+2ab}{ab}\right) > \log 2 \\
= \ & 1
\end{aligned}
$$

□