1 Overview

In this lecture we cover Splay Trees, first introduced and analyzed by Sleator and Tarjan in [6]. Splay tree is a comparison based data structure (in particular — Binary Search Tree), which — in contrast to BST data structures such as AVL tree and Red-Black tree — does not maintain a property of being balanced in any way. Thus, in the worst case, complexity of a single operation could be linear in the size of structure. Nevertheless, thanks to its self-adjusting it has a bunch of interesting properties in terms of amortized guarantees — in particular, the amortized cost of a single operation is at most $O(\log n)$, matching worst-case bounds obtained by balanced binary tree. Moreover, often when sequence of inputs is structured in additional way (for example: the sequence is in some sense local, or frequencies of accessing items are far from uniform), the upper bounds for a sequence of operations are significantly better then those given by balanced binary search trees. More about this in Section 4.

It should be noted, that this data structure has great practical application. In particular, in 1999 Sleator and Tarjan were granted a Paris Kanellakis Theory and Practice Award, which is given to honor specific theoretical accomplishments that have had a significant and demonstrable effect on the practice of computing in 1999, with a note: For creating the Splay-Tree Data Structure [...] one of the most widely used data structures invented in the last 20 years.

2 Problem formulation and model

2.1 Problem formulation

Our data structure will support following operations:

- **Insert**$(k, v)$ — inserts new item with key $k$ and value $v$.
- **Find**$(k)$ — finds value associated with item $v$.
- **Delete**$(k)$ — deletes from data structure value associated with key $k$

Those are exactly operations that are defining dictionary problem in data structures — one which is traditionally approached by hash tables. Here we insist that the data structure we provide is comparison based (the only operation available on keys, apart from copying and storing them, is comparison). Moreover, the data structure we provide, will actually fit in much more strict model: namely BST model. Such data structures usually can be enriched to support more complex operations such as various forms of range queries and range updates, or predecessor problem.
2.2 BST model

At the beginning of each operation our program is allowed to have only a single pointer to the root of a tree — which is required to be a Binary Search Tree, i.e. every node has at most two children and exactly a single key $k$, all the nodes in left subtree should have keys smaller then $k$, all the nodes in right subtree should have keys larger than $k$. The operations we are allowed to use, is to follow a pointer to any child or a parent, or do a rotation. Rotation of node $x$, with a parent $y$ basically makes $x$ a parent of $y$ preserving a BST property — there is only one way of doing it locally, and it is presented on Figure 1.

3 Splay trees

The most important internal operation in the Splay tree data structure is Splay($x$). This operation in a sequence of rotations lift the element $x$ to the root of the tree, in such a way, so that it significantly improves a structure of the tree if the path to $x$ were long in the first place. Before we discuss how this operation is realized, and analyze its complexity, lets see how to define all other operations of splay tree in terms of splay operation. For a time being we only need to know that in effect of a splay operation on a node $x$, node $x$ is a root of BST containing the same elements as the one before splay operation.

- **Insert($k,v$)** — follows child pointers as usually in BST, to find appropriate place to insert key $k$, creates new node there, then splay on this node.

- **Find($k$)** — finds a value in BST, then splay on the node containing it.

- **Delete($k$)** — finds a node with key $k$, splay it. One obtains two BSTs $L$ and $R$ out of the left and right subtree of $k$. Splay the minimum element in the $R$, then set $L$ as a left child of the root of $R$. Note that after splaying minimum element in $R$ root does not have a left child.

It is easy to see, that we can charge the cost of finding an element in BST (thus: descending a path to it, based on results of comparisons), to subsequent splay operations. Therefore it is enough to understand the amortized complexity of all splay operations in the data structure. This analysis
will be presented in Section 5. Before we do it, let’s review some interesting properties of splay tree in terms of amortized complexity.

4 Properties

We will state a set of properties of a data structure desirable by us. Some of them are proven to hold for splay trees, some are conjectured. For a time being let’s assume that we are interested in total complexity of a sequence which consists only of \texttt{find()} operations. We can assume also, that keys are just natural numbers \{1, \ldots, n\}, as only the ordering of keys matters for a comparison based data structure.

- **Static Optimality** — proven for Splay Trees in [6].
  
  Given a sequence of find operations, for any fixed binary tree \( T \) we have:
  
  \[
  \text{time to perform all operations with splay tree} = \mathcal{O}(\text{time to perform all operations with } T)
  \]

  Note, that in particular it means that average cost of an operation is upper bounded by \( \mathcal{O}(\log n) \), thus splay trees are on average no worse than balanced tree. Moreover, if the frequencies of items are far from equal (i.e. probability distribution on the items is far from uniform), they perform significantly better. Thus, they can perform only by a constant factor worse then optimal BST constructed for this set of frequencies, even though when using splay trees one does not have to know in advance what the set of frequencies is.

- **Working Set** — proven for Splay Trees in [6].
  
  Let \( t_j \) to be the number of items accessed since the last time we saw the \( j \)-th item in a sequence of operations. Splay trees achieve:
  
  \[
  \mathcal{O}(m + n \log n + \sum_{i=0}^{m} \log(t_j))
  \]

  This should be interpreted as follows: splay trees perform significantly better than balanced BSTs when the sequence of operations is time local. For example queries a small set of items over and over again, then switches to another small set of items and focus on it for a longer time. This kind of local query pattern is actually sometimes reasonable assumption in real life applications.

- **Static finger property** — proven for Splay Trees in [6].
  
  Let \( f \) be any fixed item. Then amortized cost of \texttt{find(i)} in splay tree is:
  
  \[
  \mathcal{O}(\log(|f - i| + 1) + 1)
  \]

  A reasonable interpretation of property is following: splay trees performs significantly better than balanced BST if a sequence of queries is very space local, for example there is a large number of queries near a given item \( i_1 \), succeeded by a large number of queries near a given item \( i_2 \), and so on. Again this locality is reasonable in real life applications.
Dynamic finger property — proven for Splay Trees by Cole, Mishra, Schmidt, Siegel in [2] [1].

If the access sequence is $\sigma_1, \sigma_2, \ldots, \sigma_m$, then the amortized cost of the $i$-th operation is:

$$O(\log(|\sigma_i - \sigma_{i+1}| + 1) + 1)$$

This again guarantees good behaviour of the splay tree with respect to space local query sequences: in particular, if elements are accessed in consecutive order, we will expect average cost of an operation to be constant.

Unified Property — Splay trees are not known yet to have this property. This property was introduced by Iacomo in [5].

If the access sequence is $\sigma_1, \sigma_2, \ldots, \sigma_m$, then the amortized cost of the $i$-th operation is:

$$O(\min_{i< j} \log(|\sigma_i - \sigma_j| + j - i + 1) + 1)$$ (1)

This property unifies both dynamic finger property and working set property, and it is conjectured in the same paper by Iacomo that splay trees indeed do have this property. Iacomo in [5] gives other comparison based data structure, which does have the Unified Property, unfortunately it is not a BST data structure, and it is significantly more complicated than splay trees (requires 21 pointers per node).

Later Derryberry and Sleator [4] provided a BST data structure obtaining time cost for a sequence of form

$$O(m \log \log n + UB(\sigma))$$

where UB($\sigma$) is unified bound for the sequence, i.e. sum of costs as in equation (1) over all $j$.

Dynamic Optimality Property — Splay Trees are still not known to have this property. They were conjectured to have it in the original paper [6].

Let OPT($\sigma$) be the optimal cost of servicing access sequence $\sigma$ in the BST model (i.e. if one knows the entire sequence in the advance). Then dynamic optimality property is that total time of serving a sequence $\sigma$ is upper bounded by $c \text{OPT}(\sigma)$ for some universal constant $c$.

Splay trees are conjectured to have this property, and it is the most important unsolved conjecture about splay trees at the time: it remains unsolved for almost 30 years.

Note that, trivially a cost of serving sequence $\sigma$ with splay trees is upper bounded by $O(\log n \text{OPT}(\sigma))$. Demaine, Harmon, Iacono and Patrascu in [3] gave a data structure which achieves cost($\sigma$) $\leq O(\text{OPT}(\sigma) \log \log n)$.

5 Splay operation

5.1 Operation

In this section we will tell the details of realizing splay operation. We would like to remind at this point, that as a result of a splay operation on a given node, this node should be lifted up to a root of the tree. The naive way of doing this would be to perform a sequence of rotate operations on a given node, until the node becomes a root of the tree. As we see on Figure 2 if the tree was in fact a long list (i.e. of height $n$), such a sequence of rotate operations would result in a tree again of
height $n$. Thus, if one performed a sequence of $n$ splay operations on the lowest leaf of the tree, the total time of such a sequence would be of order $\Omega(n^2)$.

We will do something slightly more complicated: let us consider node $x$, its parent $y$, and $z$ — parent of node $y$. There are three cases:

- **Case 1 (zig):** Parent of $x$ is a root of the tree — we just do rotate($x$).
- **Case 2 (zig-zig):** $x$ is left child of $y$ and $y$ is left child of $z$ or symmetric: in this case we do rotate($y$) and rotate($x$) as on Figure 3.
- **Case 3 (zig-zag):** $x$ is left child of $y$ and $y$ is right child of $z$, or symmetric: in this case we do rotate($x$) followed by rotate($x$).

Figure 2: Repeatedly rotating element in a list-like tree

Figure 3: Zig-zig step in splay operation (Case 2)
We perform a sequence of above defined operations until \( x \) becomes a new root of the tree. Note that if we do this on a leaf of a tree which is originally just a long list of height \( n \), the resulting tree will have height roughly \( \frac{n}{2} \) — as presented on Figure 5.

5.2 Analysis

To analyze a amortized complexity of a splay operation we will use the potential method. Let’s consider an arbitrary function \( w \) mapping nodes of a tree to positive real numbers. We will define potential function in terms of weight function \( w \), prove a bound for amortized complexity of a splay operation in terms of \( w \). Then, taking different functions as \( w \) various properties (as static optimality, or working set property) will follow.

In deed, for a weight function \( w \), and a node \( x \) of a tree let us define 
\[
s(x) := \sum_{y \in \text{subtree}(x)} w(y)
\]
then 
\[
r(x) := \log(s(x))
\]
and finally:
\[
\Phi(S) := \sum_{x \text{ node}} r(x)
\]
Where \( \Phi \) is potential function of interests. The main lemma is presented below. Note that by taking constant weight function, simple consequence of this lemma is that amortized cost of a single operation is upper bounded by \( O(\log n) \).

**Lemma 1.** Amortized cost of a single splay(\( x \)) operation is \( O(1 + \log \frac{W}{s(x)}) \), where \( W \) is sum of weights of all items.

**Proof.** We want to bound amortized cost of the three basic operations: zig, zig-zig, and zig-zag. Let us say, that \( r'(x) \) is defined as \( r \) on node \( x \) after performing an operation, and \( r(x) \) would be the value before performing an operation. If we had that amortized cost of zig-zig, and zig-zag on node \( x \) were at most \( 3(r'(x) - r(x)) \), and amortized cost of zig were at most \( 3(r'(x) - r(x)) + 3 \), then by summing over all basic operations in a splay operation, and telescoping the sum of we would obtain that cost of a splay operation is bounded by \( 3 + r'(\text{root}) - r(x) = O(1 + \log \frac{W}{s(x)}) \).

Thus all we have to prove is that the amortized cost of each operation is as claimed. We will prove it only for zig-zag case, as it is the most interesting — the remaining cases are similar.
Figure 5: Result of a splay operation on the leaf of linear height tree. The height of the tree is roughly two times smaller after the operation.
We want to show that amortized cost of zig-zig operation is not greater than $3(r'(x) - r(x))$. As the amortized cost is by definition equal to the real cost plus difference in potential, whereas the real cost is equal to 2, what we really need is to upper bound the difference in potential by $3(r'(x) - r(x)) - 2$. Once again, let $y$ be a parent of $x$, and $z$ be a parent of $y$ as on Figure 3. Now it is easy to see, that $r(p) = r'(p)$ when $p \notin \{x, y, z\}$. Therefore the difference in potential is equal:

$$
\Delta \Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)
$$

Where the first equality is justified by $r'(x) = r(z)$, and two last inequalities are just of form $r(y) \geq r(x)$ and $r'(y) \leq r'(x)$. Now we want to bound $r'(z)$:

$$\frac{r(x) + r'(z)}{2} = \log(\sqrt{s(x)s'(z)})$$

$$\leq \log(\frac{s(x) + s'(z)}{2})$$

$$\leq \log(\frac{s'(x)}{2})$$

$$= r'(x) - 1$$

Where $s(x) + s'(z) \leq s'(x)$ follows from the fact, that subtree rooted at $x$ in tree before the operation is completely disjoint (in terms of node set) with the subtree rooted at $z$ after the operation. Moreover, both those subtrees are contained in the subtree rooted in $x$ of a tree after the operation.

Rearranging terms we have:

$$r'(z) \leq 2r'(x) - r(x) - 2$$

Now we plug this into the bound for a difference of potential:

$$\Delta \Phi \leq r'(x) + r'(z) - 2r(x)$$

$$\leq 3(r'(x) - r(x)) - 2$$

We obtained desired bound on the difference of potential in single zig-zig step. The amortized cost of this step is hence upper bounded by $3(r'(x) - r(x))$. One can obtain a same bound for a zig-zag step, and a similar (up to constant) for a zig step. In every splay operation, a zig step is performed at most once, the total cost of the splay operation is upper bounded by a sum of expressions of form $3(r'(x) - r(x))$, and this sum telescopes to a $O(1 + r'(\text{root}) - r(x))$, which is $O(1 + \log \frac{W}{r(x)})$.

6 Concrete amortized complexity bounds

As noted before stating the main lemma, by setting a constant weight function $w \equiv 1$, we get a $O(\log n)$ upper bound on amortized complexity of single operation — thus splay trees are, in
average, at least as good as balanced BSTs. We will see how to set a weight function to prove a slightly weaker version of static optimality. The argument below is due to a LiveJournal post of David Eppstein, whose goal was to present static optimality of splay trees in a way that avoids mention of Shannon entropy as was done in the original paper [6].

Consider a tree $T$, and let $l_i$ be the level on which element $i$ is in tree $T$, and let $m_i$ to be the number of times that $i$ were accessed in the access sequence under consideration. Note that cost of processing this sequence with a tree $T$ is $O(m + \sum_{i=1}^{n} m_i l_i)$. We will prove that for splay trees this cost is at most $O(m + n^2 + \sum_{i=1}^{n} m_i l_i)$. Let's take the weight function $w(i) = 3^{-l_i}$. Note that with non integral weights potential function might take negative values — this is not a problem, we still have the important property:

$$\text{amortized\_time} = \text{actual\_time} + \Delta \Phi$$

hence

$$\text{actual\_time} = \text{amortized\_time} - \Delta \Phi$$

First note that sum of all weights of a tree is at most constant:

$$W = \sum_i w_i = \sum_l \frac{1}{3^l} (\#\text{nodes at height } l) \leq \sum_l \left(\frac{2}{3}\right)^l \leq C$$

We observe that $\Phi$ takes values in the interval $[-cn^2, C'n]$ — indeed: $w(i) \geq 3^{-n}$, so $s(x) \geq 3^{-n}$, then $r(x) \geq -cn$, and $\sum_x r(x) \geq -cn^2$. On the other hand sum of all weights is at most constant, so $s(x) \leq C$, and $r(x) \leq C'$ — sum of all $r(x)$ is at most $C'n$. Now, as $\Phi$ takes values in an interval of length $O(n^2)$, we conclude that $-\Delta \Phi = O(n^2)$.

We need to prove that amortized time is at most $O(m + \sum_{i=1}^{n} m_i l_i)$. By the main lemma, amortized cost of a single operation is at most $O(1 + \log(\frac{W}{s(i)}))$, and as $W$ is constant, and $s(i) = 3^{-l_i}$ this cost is upper bounded by $O(1 + \log(3^{l_i})) = O(l_i)$, and amortized cost of a whole sequence is indeed $\sum_{i=1}^{n} l_i m_i$.

References


