Using SPLAY Tree’s for state-full packet classification

1- What is a Splay Tree?

These notes discuss the splay tree, a form of self-adjusting search tree in which the amortized time for an access, insertion, or deletion $O(\log n)$. Splay trees are binary search trees that have implemented a self-adjusting mechanism. This mechanism performs in the following way: every time we access a node of the tree, whether for insertion or retrieval, we perform rotations (like in AVL trees), lifting the newly inserted/accessed node all the way up, so that it becomes the root of the modified tree.

The nodes on the way are rotated such that the tree becomes more balanced. The splay tree is NOT a height balanced tree since there are situations when a node may have the balance factor different from -1, 0 or +1. Nodes that are frequently accessed will frequently be lifted up to become the root, and they will never drift too far from the top position. Inactive nodes, on the other hand, will slowly be pushed farther and farther from the root. It is possible that splay trees can become highly unbalanced, so that a single access to a node of the tree can be quite expensive. However, we can prove that, over a long sequence of accesses, splay trees are not at all expensive and are guaranteed to require not many more operations even than AVL amortized analysis trees.

The analytical tool used is called amortized algorithm analysis, since, like insurance calculations, the few expensive cases are averaged in with many less expensive cases to obtain excellent performance over a long sequence of operations. The operations that are performed are rotations of a similar form to those used for AVL trees, but now with many rotations done for every insertion or retrieval in the tree. In fact, rotations are done all along the path from the root to the target node that is being accessed. Let us now discuss precisely how these rotations proceed.

The structure of a splay node is similar with the one that was used for binary search trees. Note the appearance of the parent pointer.

```c
struct NodeSplay {
    int key;
    nod *left, *right, *parent;
};
```
Where:

- key represents the tag of the node (integer number),
- left, right and parent represent pointers to the left and right children and to parent node.

Splay trees are typically used in the implementation of caches, memory allocators, routers, garbage collectors, data compression, ropes (replacement of string used for long text strings), in Windows NT (in the virtual memory, networking, and file system code) etc.

A splay tree is an efficient implementation of a balanced binary search tree that takes advantage of locality in the keys used in incoming lookup requests. For many applications, there is excellent key locality.

A good example is a network router. A network router receives network packets at a high rate from incoming connections and must quickly decide on which outgoing wire to send each packet, based on the IP address in the packet. The router needs a big table (a map) that can be used to look up an IP address and find out which outgoing connection to use. If an IP address has been used once, it is likely to be used again, perhaps many times. Splay trees can provide good performance in this situation.

### 2- Operations on Splay Trees

These are binary search trees which are self-adjusting in the following way:

Every time we access a node of the tree, whether for retrieval or insertion or deletion, we perform radical surgery on the tree, resulting in the newly accessed node becoming the root of the modified tree. This surgery will ensure that nodes that are frequently accessed will never drift too far away from the root whereas inactive nodes will get pushed away farther from the root.

- Amortized complexity
  - Splay trees can become highly unbalanced so that a single access to a node of the tree can be quite expensive.
  - However, over a long sequence of accesses, the few expensive cases are averaged in with many inexpensive cases to obtain good performance.

- Does not need heights or balance factors as in AVL trees and colours as in Red-Black trees.

- The surgery on the tree is done using rotations, also called as splaying steps. There are six different splaying steps.
1. Zig Rotation (Right Rotation)

2. Zag Rotation (Left Rotation)

3. Zig-Zag (Zig followed by Zag)

4. Zag-Zig (Zag followed by Zig)

5. Zig-Zig

6. Zag-Zag

Here we briefly explain them:

- Consider the path going from the root down to the accessed node.
  - Each time we move left going down this path, we say we `zig' and each time we move right, we say we `zag.'

- Zig Rotation and Zag Rotation
  - Note that a zig rotation is the same as a right rotation whereas the zag step is the left rotation.
  - See Figure 1

- Zig-Zag
  - This is the same as a double rotation in an AVL tree. Note that the target element is lifted up by two levels.
  - See Figure 2

- Zag-Zig
  - This is also the same as a double rotation in an AVL tree.
  - Here again, the target element is lifted up by two levels.
  - See Figure 3

- Zig-Zig and Zag-Zag
  - The target element is lifted up by two levels in each case.
  - Zig-Zig is different from two successive right rotations; zag-zag is different from two successive left rotations. For example, see Figures 4 and 5.
• See Figure 6 for an example

**Figure 1:** Zig rotation and zag rotation

**Figure 2:** Zig-zag rotation

**Figure 3:** Zag-zig rotation

**Figure 4:** Zig-zig and zag-zag rotations
Figure 5: Two successive right rotations

Figure 6: An example of splaying
3- Search, Insert, Delete in Bottom-up Splaying

Search \((i, t)\)

If item \(i\) is in tree \(t\), return a pointer to the node containing \(i\); otherwise return a pointer to the null node.

- Search down the root of \(t\), looking for \(i\)
- If the search is successful and we reach a node \(x\) containing \(i\), we complete the search by splaying at \(x\) and returning a pointer to \(x\)
- If the search is unsuccessful, i.e., we reach the null node, we splay at the last non-null node reached during the search and return a pointer to null.
- If the tree is empty, we omit any splaying operation.

Example of an unsuccessful search: See Figure 7
In the Name of Allah

Network Processors

Instructor: Dr. Mahdi Abbasi

Department of Computer Engineering

**figure 7:** An example of searching in splay trees

![Splay Tree Illustration]

Insert \((i, t)\)

- Search for \(i\). If the search is successful then splay at the node containing \(i\).
- If the search is unsuccessful, replace the pointer to null reached during the search by a pointer to a new node \(x\) to contain \(i\) and splay the tree at \(x\).

For an example, see Figure 8.

**Figure 8:** An example of an insert in a splay tree

![Splay Tree Illustration]

Delete \((i, t)\)

- Search for \(i\). If the search is unsuccessful, splay at the last non-null node encountered during search.
- If the search is successful, let \(x\) be the node containing \(i\). Assume \(x\) is not the root and let \(y\) be the parent of \(x\). Replace \(x\) by an appropriate descendent of \(y\) in the usual fashion and then splay at \(y\).

For an example, see Figure 9.
**Figure 9:** An example of a delete in a splay tree

A sample code of splay tree may be found at :

http://www.sanfoundry.com/cpp-program-implement-splay-tree/

### 4- Splay Trees to do state-full packet classification

The process of classifying packets into “flows” in routers, firewalls, packet filters etc., is called packet classification. Packet classification is used in a variety of applications such as security, monitoring, multimedia applications etc. These applications operate on packet flows or set of flows. Therefore these nodes must classify packets traversing through it in order to assign a flow identifier, called as Flow.

Packet Classification starts by building a classifier of rules or filter table, then searching that table for a particular filter or a set of filters that match the incoming packets. Each filter consists of a number of filed values. The field values may be an address filed such as source, destination addresses or a port field namely source, destination ports or protocol type.

The main research issues in the design of optimal packet classification techniques are: to increase the packet classification speed, to increase the update performance speeds for new rules, to decrease the storage requirements for caching these rules.
Splay trees are self-balancing (or) self-adjusting binary search trees [2]. It has special update and access rules. Every time we access a node of the tree, whether for retrieval or insertion or deletion, we perform radical surgery on the tree, resulting in the newly accessed node becoming the root of the modified tree. This surgery will ensure that nodes that are frequently accessed will never drift too far away from the root whereas inactive nodes will get pushed away farther from the root.

When we access a node, we apply either a single rotation or a series of rotations to move the node to the root. The biggest advantage of using Splay trees is that it does not require height or balance factors as in AVL trees and colors as in Red-Black trees. Informally, one can think of the splay trees as implementing a sort of LRU policy on tree accesses i.e. the most recently accessed elements are pulled closer to the root; and indeed, one can show that the tree structure adapts dynamically to the elements accessed, so that the least frequently used elements will be those furthest from the root. But remarkably, although no explicit balance conditions are imposed on the tree, each of these operations can be shown to use time O(log n) on an n-element tree, in an amortized sense . as it is mentioned before, There are six rotations possible in a splay tree:

1. Zig Rotation
2. Zag Rotation
3. Zig-Zig Rotation
4. Zag-Zag Rotation
5. Zig-Zag Rotation
6. Zag-Zig Rotation

All operations are illustrated in figure 10.
Description of SP-Classifier

In this project, we examine a novel representation of the input set and build a tree from this input set. The basic idea is to convert the set of prefixes into integers. Firstly, we find out the lower and upper bounds for each prefix in the source address. Then we convert these values to integers and store them in a database. The same procedure is carried out for each of the prefixes of the destination and stored in the database. While classifying incoming packets, we reject packets if they do not match the constraints. Finally we find out the best matching filter (rule) among the various filters for the valid packets.

The proposed work is a basic extension of the Hierarchical Tries. Here, we convert each of the source and destination prefixes into integer ranges. Then the corresponding tree is constructed. The greatest advantage of this approach is that the prefix specification can be extended to any number of bits. Let follow a sample filter-set representation by this method.

Table 1 and Table 2 represent a sample filter set. We first compute the lower and upper bounds for each of the prefixes in the source address as shown in Table 1. Then a source splay tree is constructed with the bounds converted to integers, which is shown in Figure 11. Similarly, using the destination addresses, a destination splay tree is constructed as shown in Table 2 and Figure 12.

Table 1. Source prefixes conversion
**Table 2. Destination Prefix Conversion**

<table>
<thead>
<tr>
<th>Filter</th>
<th>Source Prefix</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Start</th>
<th>Finish</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>01*</td>
<td>010000</td>
<td>011111</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>F2</td>
<td>1*</td>
<td>100000</td>
<td>111111</td>
<td>32</td>
<td>63</td>
</tr>
<tr>
<td>F3</td>
<td>10*</td>
<td>100000</td>
<td>101111</td>
<td>32</td>
<td>47</td>
</tr>
<tr>
<td>F4</td>
<td>01*</td>
<td>010000</td>
<td>011111</td>
<td>16</td>
<td>31</td>
</tr>
<tr>
<td>F5</td>
<td>00*</td>
<td>000000</td>
<td>001111</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>F6</td>
<td>*</td>
<td>000000</td>
<td>111111</td>
<td>0</td>
<td>65</td>
</tr>
</tbody>
</table>

**Figure 11. Source Splay Tree**

**Figure 11. Destination Splay Tree**
Now the source and destination splay trees are to be linked. For this, we connect each leaf of the source splay tree to the root of the destination splay tree. This connection pointer is similar to a Next-Trie pointer used in a hierarchical trie data structure.

In order to find out the best matching filter, we convert the source and destination prefixes of the search packet to integer values and then begin searching. We first see the integer source value of the packet and find out the filters whose lower bound is less than the packet’s source integer value and whose upper bound is greater than the packet’s destination integer value. In other words, we pick out all those filters within whose range the search packet’s value lies. Similarly, we find out the matching filters for the destination’s integer value of the search packet. Then, we perform a simple comparison between the filters that have matched the source and destination of the packet separately.

To construct a source search table and a destination search table, we get the set of distinct integer values in both the source and destination trees and arrange them in the ascending order. We now find the set of filters that match all points between the first and second integer values, between the second and third value and so on until the entire table is constructed as shown in Tables 3 and 4.

An example search packet is also shown below these tables. We firstly convert the prefixes of the search packet to integers and then find the separate source and destination matching filters and finally find the final set of matching filters for that particular search packet.

<table>
<thead>
<tr>
<th>Src Point</th>
<th>Filter &gt; than point</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F5 , F6</td>
</tr>
<tr>
<td>15</td>
<td>F6</td>
</tr>
<tr>
<td>16</td>
<td>F1 , F4 , F6</td>
</tr>
<tr>
<td>31</td>
<td>F6</td>
</tr>
<tr>
<td>32</td>
<td>F2 , F3 , F6</td>
</tr>
<tr>
<td>47</td>
<td>F2 , F6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dest Point</th>
<th>Filter &gt; than point</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F3 , F4</td>
</tr>
<tr>
<td>15</td>
<td>F3</td>
</tr>
<tr>
<td>16</td>
<td>F1 , F2 , F3</td>
</tr>
<tr>
<td>31</td>
<td>--</td>
</tr>
<tr>
<td>32</td>
<td>F5 , F6</td>
</tr>
</tbody>
</table>
Example Search

(000110, 101100) => (06, 44)
  => \{(F_5, F_6) ; (F_5, F_6) \}
  => \{F_5, F_6 \}

Objectives:

1- Use ClassBench tool to produce some synthetic rule-set and headers.

2- Implement SP-Classifier according to above explanation (in C++ Only).

3- Store your rule-set in SP-Classifier

4- Classify synthetic packets of step1, and compute following parameters:
   
   1- Number of memory accesses for classifying each packet
   
   2- Max/ Min and average number of memory accesses for classifying all synthetic packets.

5- Prepare a technical report explain steps of implementation, simulation and results!

Project due date is **18 Feb 2015**

Good Luck!