Splay Trees

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.

Importantly, splay trees offer amortized $O(\lg n)$ performance; a sequence of M operations on an n-node splay tree takes $O(M \lg n)$ time.

A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

We have already seen a way to move an element upward in a binary search tree: tree rotation. When an element is accessed in a splay tree, tree rotations are used to move it to the top of the tree. This simple algorithm can result in extremely good performance in practice. Notice that the algorithm requires that we be able to update the tree in place, but the abstract view of the set of elements represented by the tree does not change and the rep invariant is maintained. This is an example of a benign side effect, because it does not change the value represented by the data structure.

There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects: they move the node being splayed upward in the tree, and they also shorten the path to any nodes along the path to the splayed node. This latter effect means that splaying operations tend to make the tree more balanced.

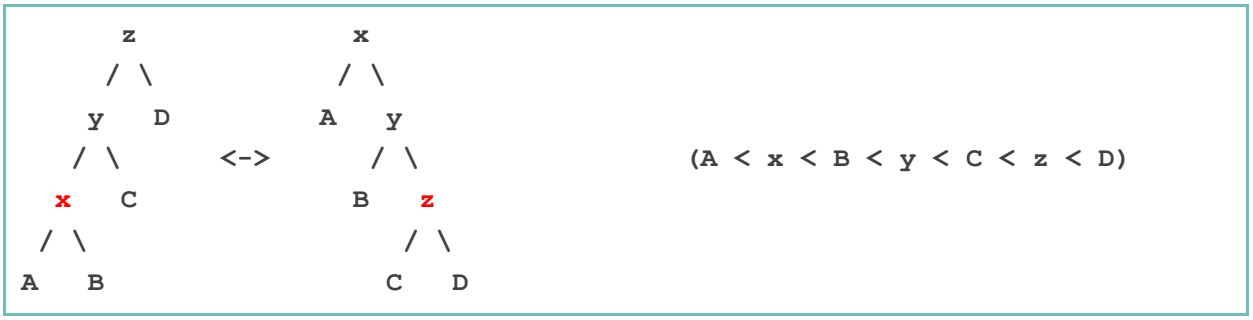
Rotation 1: Simple rotation

The simple tree rotation used in **AVL trees** and **treaps** is also applied at the root of the splay tree, moving the splayed node `x` up to become the new tree root. Here we have `A < x < B < y < C`, and the splayed node is either `x` or `y` depending on which direction the rotation is. It is highlighted in red.

Rotation 2: Zig-Zig and Zag-Zag

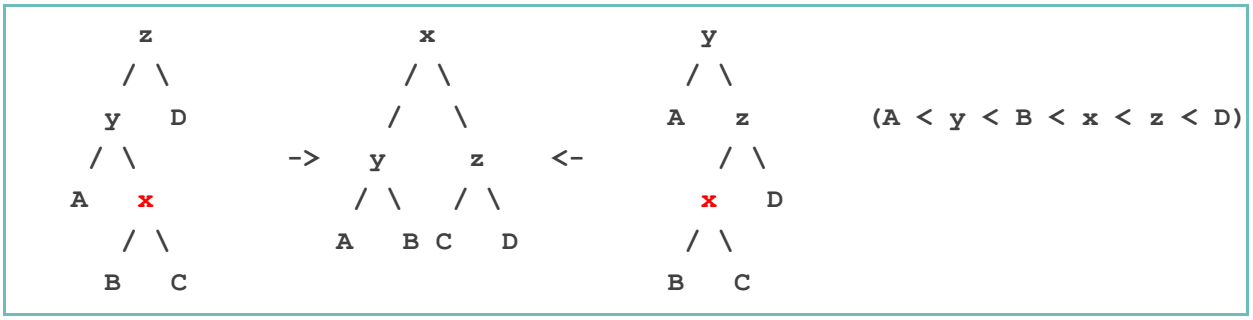
Lower down in the tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average. In the "zig-zig" case, the

splayed node is the left child of a left child or the right child of a right child ("zag-zag").



Rotation 3: Zig-Zag

In the "zig-zag" case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree. In each of the two cases shown, `y` is the splayed node:



See **this page** for a nice visualization of splay tree rotations and a demonstration that these rotations tend to make the tree more balanced while also moving frequently accessed elements to the top of the tree.

The classic version of splay trees is an imperative data structure in which tree rotations are done by imperatively updating pointers in the nodes. This data structure can only implement a mutable set/map abstraction because it modifies the tree destructively.

The **SML code below** shows that it is possible to implement an immutable set abstraction using a version of splay trees. The key function is splay, which takes a non-leaf node and a key `k` to look for, and returns a node that is the new top of the tree. The element whose key is `k`, if it was present in the tree, is the value of the returned node. If it was not present in the tree, a nearby value is in the node.

Unlike the classic imperative splay tree, this code builds a new tree as it returns from the recursive splay call. And the case of a simple rotation occurs at the bottom of the tree rather than the top.

```
signature ORDERED_FUNCTIONAL_SET = sig
  (* Overview: a "set" is a set of distinct
   * elements of type "elem". Each element is
   * identified by a unique key, which may be the
   * same as the element itself. Two elements are
   * considered distinct if they have different
   * keys.  Keys are a totally ordered set.
   *
   * A set can be used to represent an ordinary
   * set if key = elem.  It can be used to
```

```
   * represent a mapping if elem = key * value.
   *
   * For example, if key and elem are int,
   * then a set might be {1,-11,0}, {}, or
   * {1001}. If key is string and elem is int,
   * a set could be {("elephant", 2), ("rhino",
   * 25), ("zebra", 2)} *)
  type key
  type elem
  type set

  (* compare(k1,k2) reports the ordering of k1 and k2. *)
  val compare: key * key -> order
  (* keyOf(e) is the key of e. *)
  val keyOf: elem -> key
  (* empty() is the empty set. *)
  val empty : unit -> set
  (* Effects: add(s,e) is s union {e}. Returns true
   * if e already in s, false otherwise. *)
  val add: set * elem -> set * bool
  (* remove(s,k) is (s',eo) where s' = s - {k} (set difference)
   * and eo is either SOME e if there is an e in s
   * where k is e's key, or NONE otherwise. *)
  val remove: set * key -> set * elem option
  (* lookup(s,k) is SOME e where k = keyOf(e), or NONE if
   * the set contains no such e. *)
  val lookup: set * key -> elem option
  (* size(s) is the number of elements in s. *)
  val size: set -> int

  (* Ordered set operations *)

  (* first(s) is SOME of the element of s with the smallest key,
   * or NONE if s is empty. *)
  val first: set -> elem option
  (* last(s) is SOME of the element of s with the largest key,
   * or NONE if s is empty. *)
  val last: set -> elem option
  (* A fold operation on ordered sets takes a key argument
   * that defines the element where the fold starts. *)
  type 'b folder = ((elem*'b)->'b) -> 'b -> key -> set -> 'b
  (* fold over the elements in key order. *)
  val fold_forward: 'b folder
  (* fold over the elements in reverse key order. *)
  val fold_backward: 'b folder

  val print: set -> unit
end

signature ORDERED_SET_PARAMS = sig
```

```
  type key
  type elem
  val keyOf: elem -> key
  val compare: key * key -> order
  val toString: elem -> string
end

functor SplayTree(structure Params : ORDERED_SET_PARAMS)
  :> ORDERED_FUNCTIONAL_SET where type key = Params.key and
                                  type elem = Params.elem =
struct
  type key = Params.key
  type elem = Params.elem
  val compare = Params.compare
  val keyOf = Params.keyOf
  datatype tree =
    Empty
  | Node of tree * elem * tree
  type node = tree * elem * tree
  (* Representation invariant: given a node (L, V, R),
   * All values in the left subtree L are less than V, and
   * all values in the right subtree R are greater than V, and
   * both L and R also satisfy the RI.
   *)
  type set = int * (tree ref)
  (* Representation invariant: size is the number of elements in
   * the referenced tree. *)

  fun empty() = (0, ref Empty)

  (* splay(n,k) is a BST node n' where n' contains
   * all the elements that n does, and if an
   * element keyed by k is in under n, #value n'
   * is that element.  Requires: n satisfies the
   * BST invariant.
   *)
  fun splay((L, V, R), k: key): node =
    case compare(k, keyOf(V))
      of EQUAL => (L, V, R)
       | LESS =>
        (case L
           of Empty => (L, V, R) (* not found *)
            | Node (LL, LV, LR) =>
             case compare(k, keyOf(LV))
               of EQUAL => (LL, LV, Node(LR, V, R)) (* 1: zig *)
                | LESS =>
                 (case LL
                    of Empty => (Empty, LV, Node(LR, V, R))
                                (* not found *)
                     | Node n => (* 2: zig-zig *)
```

```
                    let val (LLL, LLV, LLR) = splay(n,k) in
                      (LLL,LLV,Node(LLR,LV,Node(LR,V,R)))
                    end)
                | GREATER =>
                    (case LR
                       of Empty => (LL, LV, Node(Empty, V, R))
                        | Node n =>  (* 3: zig-zag *)
                         let val (LRL, LRV, LRR) = splay(n,k) in
                           (Node(LL,LV,LRL),LRV,Node(LRR,V,R))
                         end))
      | GREATER =>
       (case R
           of Empty => (L, V, R) (* not found *)
           | Node (RL, RV, RR) =>
           case compare(k, keyOf(RV))
               of EQUAL => (Node(L,V,RL),RV,RR) (* 1: zag *)
               | GREATER =>
               (case RR
                   of Empty => (Node(L,V,RL),RV,RR) (* not found *)
                   | Node n => (* 3: zag-zag *)
                       let val (RRL, RRV, RRR) = splay(n,k) in
                       (Node(Node(L,V,RL),RV,RRL),RRV,RRR)
                       end)
               | LESS =>
               (case RL
                   of Empty => (Node(L,V,RL),RV,RR) (* not found *)
                   | Node n => (* 2: zag-zig *)
                       let val (RLL, RLV, RLR) = splay(n,k) in
                       (Node(L,V,RLL),RLV,Node(RLR,RV,RR))
                       end))

fun lookup((size,tr),k) =
  case !tr of
    Empty => NONE
  | Node n =>
      let val n' as (L,V,R) = splay(n,k) in
        tr := Node n';
        if compare(k, keyOf(V)) = EQUAL then SOME(V)
        else NONE
      end

fun add((size,tr):set, e:elem) = let
  val (t', b) = add_tree(!tr, e)
  val t'': node = splay(t', keyOf(e))
  val size' = if b then size else size+1
in
  ((size', ref (Node(t''))),b)
end
and add_tree(t: tree, e: elem): node * bool =
  case t
```

```
      of Empty => ((Empty, e, Empty), false)
       | Node (L,V,R) =>
        (case compare (keyOf(V),keyOf(e))
           of EQUAL => ((L,e,R),true)
            | GREATER => let val (n',b) = add_tree(L, e) in
                           ((Node(n'),V,R),b)
                         end
            | LESS =>    let val (n',b) = add_tree(R, e) in
                           ((L,V,Node(n')),b)
                         end)


  fun size(s,tr) = s

  type 'b folder = ((elem*'b)->'b) -> 'b -> key -> set -> 'b

  fun fold_forward f b k (size,tr) = fold_forward_tree f b k (!tr)
  and fold_forward_tree (f: elem*'b->'b) (b:'b) (k:key) (t:tree) =
    case t
      of Empty => b
       | Node (L,V,R) =>
        (case compare(keyOf(V), k) of
           EQUAL => fold_forward_tree f (f(V,b)) k R
         | LESS => fold_forward_tree f b k R
         | GREATER => let val lv = fold_forward_tree f b k L in
             fold_forward_tree f (f(V,lv)) k R
           end)

  fun first((size,tr)): elem option = raise Fail "first: not implemented"
  fun remove(s,e) = raise Fail "remove: not implemented"
  fun last(s,tr) = raise Fail "last: not implemented"
  fun fold_backward f b k s = raise Fail "fold_backward: not implemented"

(* The remainder of this code is for pretty-printing binary trees *)
  fun spaces(n) =
    if n > 0 then " " ^ spaces(n-1) else
      if n <= 0 then "" else
        raise Fail "negative space!"
  fun rdiag(n): string list =
    if n = 0 then []
    else rdiag(n-1) @ [spaces(n-1) ^ "\\"]
  fun ldiag(n) =
    if n = 0 then []
    else  [spaces(n-1) ^ "/"] @ ldiag(n-1)

  fun indent(sl,n) = let val ws = spaces(n) in
    map (fn(s) => ws^s) sl
  end
  fun pad(sl, n) = map (fn(s) => s ^ spaces(n - String.size(s))) sl
  fun hsplice(h1::t1,h2::t2,w1,w2) =
```

```
    (h1^h2) :: hsplice(t1,t2,w1,w2)
    | hsplice(sl1: string list, nil,w1,w2) = pad(sl1,w1+w2)
    | hsplice(nil, sl2,w1,w2) = indent(sl2, w1)
  (* toStrings(t) is (sl,w,h,r) where "sl" is a list of h strings of
   * length w representing a drawing of "t", where the root of the tree
   * is positioned in the first string at offset "r" *)
  fun toStrings(t: tree): (string list)*int*int*int =
    case t of
      Empty => ([], 0, 0, 0)
    | Node (L,V,R) => let
        val vs = Params.toString(V)
        val vl = String.size(vs)
        val (sl1,w1,h1,r1) = toStrings(L)
        val (sl2,w2,h2,r2) = toStrings(R)
        val padding = case r2 + w1 - r1
          of 0 => 2
           | 1 => 1
           | 2 => 0
           | diff => if diff mod 2 = 0 then 0 else 1
        val w = Int.max(w1 + w2 + padding,vl)
        val diagsize = (r2 + w1 - r1 + padding) div 2
        val leftarc = case L of
          Empty => []
          | _ => ldiag(diagsize)
        val rightarc = case R of
          Empty => []
        | _ => rdiag(diagsize)
        val sl = pad(indent([vs], r1 + diagsize - (vl div 2)), w) @
          pad(indent(hsplice(pad(leftarc, diagsize+1), pad(rightarc, diagsize),
                             diagsize+1, diagsize),
                     r1), w) @
          hsplice(sl1, indent(sl2, padding), w1, w2+padding)
      in
        (sl, w, Int.max(h1,h2)+diagsize+1, diagsize+r1)
      end
  fun print((size,tr)) =
    let val (sl,w,h,r) = toStrings(!tr) in
      List.app (fn(s:string) => TextIO.print (s^"\n")) sl
    end
end

structure I_Params = struct
  type key = int
  type elem = int
  fun keyOf x = x
  val compare = Int.compare
  val toString = Int.toString
end

structure IST = SplayTree(structure Params = I_Params)
```

```
open IST
fun ins_n(n) =
  if n = 0 then empty() else #1(add(ins_n(n-1), n))
```

Amortized analysis

To show that splay trees deliver the promised amortized performance, we define a potential function $\Phi(T)$ to keep track of the extra time that can be consumed by later operations on the tree T. As before, we define the **amortized time** taken by a single tree operation that changes the tree from T to T' as the **actual time** t, plus the **change in potential** $\Phi(T')$-$\Phi(T)$. Now consider a sequence of M operations on a tree, taking actual times $t_1, t_2, t_3, ..., t_M$ and producing trees $T_1, T_2, ... T_M$. The **amortized time** taken by the operations is the sum of the actual times for each operation plus the sum of the changes in potential: $t_1 + t_2 + ... t_M + (\Phi(T_2)-\Phi(T_1)) + (\Phi(T_3) - \Phi(T_2)) + ... + (\Phi(T_M) - \Phi(T_{M-1}))$ $= t_1 + t_2 + ... t_M + \Phi(T_M) - \Phi(T_1)$. Therefore the amortized time for a sequence of operations underestimates the actual time by at most the maximum drop in potential $\Phi(T_M)$ $- \Phi(T_1)$ seen over the whole sequence of operations.

The key to amortized analysis is to define the right potential function. Given a node x in a binary tree, let $size(x)$ be the number of nodes below x (including x). Let $rank(x)$ be the log base 2 of $size(x)$. Then the potential $\Phi(T)$ of a tree T is the sum of the ranks of all of the nodes in the tree. Note that if a tree has n nodes in it, the maximum rank of any node is $\lg n$, and therefore the maximum potential of a tree is $n \lg n$. This means that over a sequence of operations on the tree, its potential can decrease by at most $n \lg n$. So the correction factor to amortized time is at most $\lg n$, which is good.

Now, let us consider the amortized time of an operation. The basic operation of splay trees is splaying; it turns out that for a tree t, any splaying operation on a node x takes at most amortized time $3*rank(t) + 1$. Since the rank of the tree is at most $\lg(n)$, the splaying operation takes $O(\lg n)$ amortized time. Therefore, the actual time taken by a sequence of n operations on a tree of size n is at most $O(\lg n)$ per operation.

To obtain the amortized time bound for splaying, we consider each of the possible rotation operations, which take a node x and move it to a new location. We consider that the rotation operation itself takes time $t=1$. Let $r(x)$ be the rank of x before the rotation, and $r'(x)$ the rank of node x after the rotation. We will show that simple rotation takes amortized time at most $3(r'(x) - r(x))) + 1$, and that the other two rotations take amortized time $3(r'(x) - r(x))$. There can be only one simple rotation (at the top of the tree), so when the amortized time of all the rotations performed during one splaying is added, all the intermediate terms $r(x)$ and $r'(x)$ cancel out and we are left with $3(r(t) - r(x)) + 1$. In the worst case where x is a leaf and has rank 0, this is equal to $3*r(t) + 1$.

Simple rotation

The only two nodes that change rank are x and y. So the cost is $1 + r'(x) - r(x) + r'(y) -$ $r(y)$. Since y decreases in rank, this is at most $1 + r'(x) - r(x)$. Since x increases in rank, $r'(x) - r(x)$ is positive and this is bounded by $1 + 3(r'(x) - r(x))$.

Zig-Zig rotation

Only the nodes x, y, and z change in rank. Since this is a double rotation, we assume it

has actual cost 2 and the amortized time is

$$2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$$

Since the new rank of x is the same as the old rank of z, this is equal to

$$2 - r(x) + r'(y) - r(y) + r'(z)$$

The new rank of x is greater than the new rank of y, and the old rank of x is less than the old rank y, so this is at most

$$2 - r(x) + r'(x) - r(x) + r'(z) \quad = \quad 2 + r'(x) - 2r(x) + r'(z)$$

Now, let $s(x)$ be the old size of x and let $s'(x)$ be the new size of x. Consider the term $2r'(x) - r(x) - r'(z)$. This must be at least 2 because it is equal to $\lg(s'(x)/s(x)) + \lg(s'(x)/s'(z))$. Notice that this is the sum of two ratios where $s'(x)$ is on top. Now, because $s'(x) \geq s(x) + s'(z)$, the way to make the sum of the two logarithms as small as possible is to choose $s(x) = s(z) = s'(x)/2$. But in this case the sum of the logs is $1 + 1 = 2$. Therefore the term $2r'(x) - r(x) - r'(z)$ must be at least 2. Substituting it for the red 2 above, we see that the amortized time is at most

$$(2r'(x) - r(x) - r'(z)) + r'(x) - 2r(x) + r'(z) \quad = \quad 3(r'(x) - r(x))$$

as required.

Zig-Zag rotation

Again, the amortized time is

$$2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$$

Because the new rank of x is the same as the old rank of z, and the old rank of x is less than the old rank of y, this is

$$\begin{aligned} & 2 - r(x) + r'(y) - r(y) + r'(z) \\ \leq \quad & 2 - 2\, r(x) + r'(y) + \mathrm{r}'(z) \end{aligned}$$

Now consider the term $2r'(x) - r'(y) - r'(z)$. By the same argument as before, this must be at least 2, so we can replace the constant 2 above while maintaining a bound on amortized time:

$$\leq \quad (2r'(x) - r'(y) - r'(z)) - 2\, r(x) + r'(y) + r'(z) \quad = \quad 2(r'(x) - r(x))$$

Therefore amortized run time in this case too is bounded by $3(r'(x) - r(x))$, and this completes the proof of the amortized complexity of splay tree operations.