Splay Trees

February 22, 1996

CMU 15-211, D. Sleator

1. Introduction

Search trees are an efficient method for representing a set of elements (each with a key from a totally ordered universe) such that many operations dealing with the order of the keys in the set can be carried out efficiently. These are operations such as: insert, delete, predecessor, successor, rank, findmax, findmin, deletemax, deletemin, searchrange, and many others. This note describes a very simple and efficient way to do these things.

A *binary tree* is a collection of *internal nodes*, each of which has pointers to two other nodes, its left child and right child. Each internal node (except the *root* is pointed to by exactly one other node. The root is neither the left nor the right child of any other node, and it is where all searching in the tree begins.

Here is a typical declaration of a binary search tree node in C.

```
typedef struct node {
    struct node *left, *right;
    int key;
} Tree;
```

A *symmetric order* traversal of a binary tree is a recursive traversal of the tree the works as follows:

```
symmetric_order(Tree *t) {
    if (t == NULL) return;
    symmetric_order(t->left);
    printf("%d",t->key);
    symmetric_order(t->right);
}
```

We'll store the elements of our set in the internal nodes of a binary tree in symmetric order, that is, in such a way that a symmetric order traversal of the tree prints out all the elements in lexicographic order. For example, suppose the set of elements is $\{0, 1, 2, 3, 4, 5, 6\}$, one binary tree to represent this set might be:

```
          3
         / \
        /   \
       /     \
      1       6
     / \     /
    0   2   4
             \
              5
```

Of course there are many binary trees that represent the same set.

Below is an implementation of search.

```
Tree * search(Tree *t, int x){
    for (;;) {
        if (t==NULL) return NULL;
        else if (t->key < x) t = t->left;
        else if (t->key > x) t = t->right;
        else return t;
    }
}
```

Here is an elegant way to implement insertion. Note that it's recursive. This means that if the tree has a very long path in it, there is a risk that the stack might overflow.

```
Tree * insert(Tree * t, int lab) {
    if (t == NULL) {
        t = malloc(sizeof(Tree));
        t->left = t->right = NULL;
        t->key = lab;
        return t;
    } else {
        if (lab < t->key) {
            t->left = insert(t->left, lab);
        } else {
            t->right = insert(t->right, lab);
        }
        return t;
    }
}
```

I won't write out deletion in detail. It works as follows. To delete x, first find the node v with x as its key. This is similar to searching. If v has a NULL left child, replace v by its right child. Use an analogous method if v has no right child. If v has a non-NULL left child, then swap the key field of v with its symmetric order successor s. Now delete s using the method above (s's right child is NULL).

2. Balanced trees

The scheme described above works very well (it has expected logarithmic behavior) if the elements are inserted in a random order. We would like to be able to guarantee efficient performance under more general conditions; the assumption of randomness is rarely reasonable in practice. For example, if the elements happen to be inserted in sorted order the performance of this algorithm is $\Omega(n)$. Our goal is to get the performance to be $O(\log n)$ in the worst case, for any sequence of requests, even those planned by an adversary attempting to foil our algorithm. This is the purpose of *balanced binary trees*.

A variety of different balanced binary trees schemes have been devised. Some examples are: 2-3 trees, 2-3-4 trees, AVL trees, Red-Black trees, trees of bounded balance. These all share the following properties.

1. They all satisfy a "balance condition". This ensures that that at any node, the size of the left subtree is roughly the same as the size of the right subtree. This implies that the distance of any node from the root is bounded by a constant times $\log n$ in a tree of n nodes.
2. They store balance information in the nodes of the tree.
3. They make use of *rotations* to maintain balance. (Splitting in 2-3 and 2-3-4 trees play the same roll as rotations do in the other types of balanced search trees.)

A rotation is a local restructuring operation. There are left rotations and right rotations. The C code for doing a right rotation is the following:

```
Tree * rotate_right(Tree * y) {
    /* Do a right rotation at y.  Assumes the left child of y is not
       NULL.  Returns the new root of the tree rooted here. */

    Tree * x = y->left;
    y->left = x->right;
    x->right = y;
    return x;
}
```

The picture below shows a left rotation starting from node 1.

```
    1                                  2
   / \      left rotation at 1        / \
  0   2     ------------------->     1   4
       \                            /   / \
        4                          0   3   5
       / \
      3   5
```

3. Definition of Splaying

Balanced binary search trees have the property that every operation has a cost that is bounded by $O(\log n)$. Suppose that we relax the requirement that every operation is efficient individually, and instead consider algorithms that are efficient over a sequence of operations but not necessarily efficient on any individual operation. Such a data structure is said to possess *amortized* efficiency.

An ideal such solution would not limit the structure of the tree in any way whatsoever (relaxing constraint 1 above). It would also not require any balance information to be stored in the nodes (relaxing property 2 above). Bob Tarjan and I devised a method for doing this, which we called splay trees [1]

Suppose that the structure of the tree happens to be very unbalanced. Then there will be a node in the tree that is very deep, and thus searching for it will be very expensive. To obtain amortized efficiency it is mandatory that the tree be restructured after the operation in such a way that another search for the same element will be more efficient. We call such a method *self-adjusting*.

Splaying is the primitive operation with which we build all the other operations. The operation splay(k, t) restructures the tree in such a way that the node containing the key k (or a node which would be a neighbor of k if k is not in the tree) ends up being the root of the tree. It works by making a series of steps, each of which moves the node containing k one or steps closer to the root of the tree.

Before explaining the splay operation, I'll give a slightly simpler way to restructure the tree so that the accessed node moves to the root. Consider the following C program.

[1] A more complete treatment can be found in "Self-Adjusting Binary Search Trees", JACM, Volume 32, No. 3, July 1985, pages 652–686.

```
Tree * move_to_root (int i, Tree * t) {
    Tree N, *l, *r;
    if (t == NULL) return t;
    N.left = N.right = NULL;
    l = r = &N;

    for (;;) {
        if (i < t->key) {
            if (t->left == NULL) break;
            r->left = t;                              /* link right */
            r = t;
            t = t->left;
        } else if (i > t->key) {
            if (t->right == NULL) break;
            l->right = t;                             /* link left */
            l = t;
            t = t->right;
        } else {
            break;
        }
    }
    l->right = t->left;                               /* assemble */
    r->left = t->right;
    t->left = N.right;
    t->right = N.left;
    return t;
}
```

Here's how this works. We walk down a particular path by doing the usual comparisons until we find the node with the desired key, or we walk off the bottom of the tree. Let x be the node on which this process terminates. As we're walking down, we build up a pair of trees. One of them (whose root is potinted to by `N.right`) contains all the nodes to the left of x (not descending from x) and the other one (whose root is potinted to by `N.left`) contains those to the right of x (not descending from x).

This is accomplished one step at a time. One step puts the current node at the rightmost spot in the left tree (a place pointed to by the variable `l`) or it puts the current node at the leftmost spot in the right tree (a place pointed to by `r`). Each of these is called a link step.

After the node x is found, all the parts are assembled. The left and right subtrees of x are put into the left and right trees, and the node x becomes the root of the tree. Below is an example of this move-to-root operation.

```
                8                                  4
               /                                  / \
              7                                  3   8
             /                                  /   /
            6                                  2   7
           /        t = move-to-root(4,t)     /   /
          5         --------------------->   1   6
         /                                  /   /
        4                                  0   5
       /
      3
     /
    2
   /
  1
 /
0
```

If the node being searched for is not in the tree, then the one that ends up at the root is one of the two nodes in the tree whose key is a neighbor of the key being searched for. In other words, it's either the next larger key in the tree or the next smaller one. Which one it is depends on the structure of the tree. (The splaying algorithm described below also has this property.)

It is easy to see that this operation is not efficient in the amortized sense. For example, if we start out with a long left path of n nodes like that shown above, and apply the move-to-root algorithm to each node in tree (in left to right order) the cost will be $\Omega(n^2)$. And at the end of all this, we're back right where we started.

Only one small change is required to give an amortized cost of $O(\log n)$ per operation. The change is simply this. If we are required to take two steps in a row in the same direction while going down the tree, then when we attach those two nodes into the left or right tree, we apply an appropriate rotation to them. Here is the program to implement this.

```
Tree * splay (int i, Tree * t) {
    Tree N, *l, *r, *y;
    if (t == NULL) return t;
    N.left = N.right = NULL;
    l = r = &N;

    for (;;) {
        if (i < t->key) {
            if (t->left == NULL) break;
            if (i < t->left->key) {
                y = t->left;                          /* rotate right */
                t->left = y->right;
                y->right = t;
                t = y;
                if (t->left == NULL) break;
            }
            r->left = t;                              /* link right */
            r = t;
            t = t->left;
        } else if (i > t->key) {
            if (t->right == NULL) break;
            if (i > t->right->key) {
                y = t->right;                         /* rotate left */
                t->right = y->left;
                y->left = t;
                t = y;
                if (t->right == NULL) break;
            }
            l->right = t;                             /* link left */
            l = t;
            t = t->right;
        } else {
            break;
        }
    }
    l->right = t->left;                               /* assemble */
    r->left = t->right;
    t->left = N.right;
    t->right = N.left;
    return t;
}
```

Here is an example of a sequence of splay operations.

```
Initial tree:

                  9
                 /
                8
               /
              7
             /
            6
           /
          5
         /
        4
       /
      3
     /
    2
   /
  1
 /
0

t = splay(0,t)

      0
       \
        8
       / \
      6   9
     / \
    4   7
   / \
  2   5
 / \
1   3

t = splay(1,t)

  1
 / \
0   4
   / \
  /   \
 /     \
2       8
 \     / \
  4   7   9
 / \
3   5
```

```
t = splay(7,t)

      7
     / \
    6   8
   /     \
  1       9
 / \
0   2
     \
      4
     / \
    3   5
```

4. Implementing Other Operations Using Splaying

```
Tree * insert(int i, Tree * t) {
    Tree * new;

    new = malloc (sizeof (Tree));
    new->key = i;
    if (t == NULL) {
        new->left = new->right = NULL;
    } else {
        t = splay(i,t);
        if (i < t->key) {
            new->left = t->left;
            new->right = t;
            t->left = NULL;
        } else {
            new->right = t->right;
            new->left = t;
            t->right = NULL;
        }
    }
    return new;
}


Tree * delete(int i, Tree * t) {
/* Deletes i from the tree if it's there.              */
/* Return a pointer to the resulting tree.             */
    Tree * x;
    if (t==NULL) return NULL;
    t = splay(i,t);
    if (i == t->key) {               /* found it */
        if (t->left == NULL) {
            x = t->right;
        } else {
            x = splay(i, t->left);
            x->right = t->right;
        }
        free(t);
        return x;
    }
    return t;                         /* It wasn't there */
}
```

It is also easy to modify these programs to maintain parent pointers, or a size field which contains the number of nodes in the subtree rooted there. The size field allows efficient implementation of the rank operation. (The rank of a node is its position in the symmetric order. So I can find the 17th element in the list.)

Splaying can also be used to split a tree into two trees at an element x, or to join two trees together, find predecessors, successors, etc. Recent experiments seem to indicate that splaying is faster than any known balanced tree implementation. They're also the simplest and use the least space.

A sequence of these operations on a collection of splay trees is efficient, as shown by the following theorem.

Theorem: Let $\sigma = \sigma_1 \sigma_2 \cdots \sigma_m$ be a sequence of these operations on a collection of splay trees. Let s_i be the size of the largest tree occurring in operation σ_i. Then the cost of σ is $O(\sum_i \log s_i)$.

5. Intuition

In these notes I have not proven any theorems about the performance of splaying. However, it is possible to gain some intuition for why the method works. The basic principle on which the analysis rests is the idea that an expensive operation (such as splaying a long left path) causes the tree to become much "better". The example above illustrates this. Note that all the nodes along the access path move to about half their original distance from the root. Other nodes could get farther from the root, but not much farther.

6. Theorems and Conjectures

The following theorem bounds the cost of a sequence of splay operations.

Balance Theorem: A sequence of m splays in a tree of n nodes does at most $3m \log_2 n + m + n \log n$ rotations. If the initial tree is balanced (every node has depth bounded by $\log_2 n$) then the total cost of splaying is $3m \log n + m$.

There are a variety of other theorems about the performance of splaying. I just mention a few of these here, and a few of the many open problems.

Static Optimality Theorem: The cost of a sequence of splay operations is within a constant factor of the cost of the same sequence of accesses into any static tree, even one that is built explicitly to perform well on the sequence.

Working Set Theorem: The cost of a sequence of splay operations that only accesses a subset of the elements will be as though the elements that are not accessed are not present at all.

Sequential Access Theorem: The cost of splaying each of the n nodes in a tree in symmetric order is $O(n)$.

A doubly ended queue (or dequeue) can be implemented using splaying (of course there are other efficient methods for doing this). To insert at the left, merely make a new root, with a null left child and the old tree as its right child. To delete from the left, splay the leftmost element, and delete it. Operations on the right are analogous. This leads to a very interesting conjecture

Dequeue Conjecture: Using this implementation, starting from an empty queue, a sequence of m doubly ended queue operations take time $O(m)$.