Splay Trees

02-713
Dictionary Abstract Data Type (ADT)

- Most basic and most useful ADT:
  - `insert(key, value)`
  - `delete(key)`
  - `value = find(key)`

- Many languages have it built in:
  - awk:
    ```
    D["AAPL"] = 130
    # associative array
    ```
  - perl:
    ```
    my %D; $D["AAPL"] = 130;
    # hash
    ```
  - python:
    ```
    D = {}; D["AAPL"] = 130
    # dictionary
    ```
  - C++:
    ```
    map<string,string> D = new map<string, string>();
    D["AAPL"] = 130;
    // map
    ```

- **Insert, delete, find** each either $O(\log n)$ [C++] or expected constant [perl, python]

- How can such dictionaries are implemented?
Binary Search Trees

- **BST Property:** If a node has key $k$ then keys in the left subtree are $< k$ and keys in the right subtree are $> k$.

- For convenience, we disallow duplicate keys.

- Good for implementing the dictionary ADT we’ve already seen: insert, delete, find.
Find $k = 6$: 

```
    5
   / \
  3   8
 / \ / \  
2  4 6  11  
   /       / 
2         9  
```
BST Find

Find $k = 6$:

Is $k < 5$?
BST Find

Find $k = 6$:

Is $k < 5$?  No, go right
BST Find

Find $k = 6$:

Is $k < 5$?  No, go right

Is $k < 8$?
Find $k = 6$:

Is $k < 5$?  No, go right

Is $k < 8$?  Yes, go left
BST Find

Find $k = 9$: 

```
      5
     / \
   3    8
  / \   / \n 2   4  6   11
   \   \   \   
    9
```
BST Find

Find $k = 9$:

Is $k < 5$?
BST Find

Find $k = 9$:

Is $k < 5$?  No, go right
Find $k = 9$:

Is $k < 5$?  No, go right

Is $k < 8$?
BST Find

Find $k = 9$:

Is $k < 5$?  No, go right

Is $k < 8$?  No, go right
BST Find

Find $k = 9$:

Is $k < 5$? No, go right

Is $k < 8$? No, go right

Is $k < 11$?
Find $k = 9$:

- Is $k < 5$? No, go right
- Is $k < 8$? No, go right
- Is $k < 11$? Yes, go left
BST Find

Find $k = 13$:
BST Find

Find $k = 13$:
Is $k < 5$?
BST Find

Find $k = 13$:

Is $k < 5$?  *No, go right*
BST Find

Find $k = 13$:

Is $k < 5$?  No, go right

Is $k < 8$?
Find $k = 13$:

Is $k < 5$?  No, go right

Is $k < 8$?  No, go right
BST Find

Find $k = 13$:

Is $k < 5$?  No, go right
Is $k < 8$?  No, go right
Is $k < 11$?
Find $k = 13$:

Is $k < 5$? No, go right
Is $k < 8$? No, go right
Is $k < 11$? No, go right
**BST Insert**

\[
\text{insert}(T, K): \\
\text{\quad} q = \text{NULL} \\
\text{\quad} p = T \\
\text{\quad} \text{while} \ p \neq \text{NULL} \text{ and} \ p\.\text{key} \neq K: \\
\text{\quad\quad} q = p \\
\text{\quad\quad} \text{if} \ p\.\text{key} < K: \\
\text{\quad\quad\quad} p = p\.\text{right} \\
\text{\quad\quad} \text{else if} \ p\.\text{key} > K: \\
\text{\quad\quad\quad} p = p\.\text{left} \\
\text{if} \ p \neq \text{NULL}: \text{error DUNPLICATE} \\
\]

\[
N = \text{new Node}(K) \\
\text{if} \ q\.\text{key} > K: \\
\text{\quad} q\.\text{left} = N \\
\text{else:} \\
\text{\quad} q\.\text{right} = N \\
\]
BST Insert

\[
\text{insert}(T, K):
q = \text{NULL}
p = T
\text{while } p \neq \text{NULL and } p.\text{key} \neq K:
    q = p
    \text{if } p.\text{key} < K:
        p = p.\text{right}
    \text{else if } p.\text{key} > K:
        p = p.\text{left}

\text{if } p \neq \text{NULL: error DUPLICATE}
\]

\[
N = \text{new Node}(K)
\text{if } q.\text{key} > K:
    q.\text{left} = N
\text{else:}
    q.\text{right} = N
\]
**BST Insert**

\[
\text{insert}(T, K):
q = \text{NULL} \\
p = T \\
\text{while } p \neq \text{NULL and } p.\text{key} \neq K:
\quad q = p \\
\quad \text{if } p.\text{key} < K:
\quad \quad p = p.\text{right} \\
\quad \text{else if } p.\text{key} > K:
\quad \quad p = p.\text{left} \\
\text{if } p \neq \text{NULL: error DUPLICATE} \\
\]

\[
N = \text{new Node}(K) \\
\text{if } q.\text{key} > K:
\quad q.\text{left} = N \\
\text{else:}
\quad q.\text{right} = N
\]

*Same idea as BST Find*
BST Insert

insert(T, K):
  q = NULL
  p = T
  while p != NULL and p.key != K:
    q = p
    if p.key < K:
      p = p.right
    else if p.key > K:
      p = p.left

  if p != NULL: error DUPLICATE

N = new Node(K)
  if q.key > K:
    q.left = N
  else:
    q.right = N

Same idea as BST Find
BST FindMin

```
BST: 5
  /  \
 3  8
/   /  \
2  4  6 11
/     \
1
```

```
BST: 5
  /  \
 4  8
 /   /
1  3  6 11
```
BST FindMin

Walk left until you can’t go left any more
BST FindMin

Walk left until you can't go left any more
Can you express inorder_successor using find_min?

Walk left until you can’t go left any more
**BST Delete**

- **Node is leaf:**
  - Node: 5
  - Children: 3, 2

- **Node has 1 child:**
  - Node: 5
  - Children: 3

- **Node has 2 children:**
  - Node: 5
  - Children: 3, 2
  - Sub-children: 6, 4
BST Delete

Node is leaf:

```
3
/   \
2    4
   /\   /
  5  3 2
```

Node has 1 child:

```
3
/   \
2    2
   /\   /
  5  3  5
```

Node has 2 children:

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

```
3
/   \
2    5
   /\   /
  2  4 5
```
BST Delete

Node is leaf:

Node has 1 child:

Node has 2 children:
Partitioning

- Ordering implicitly gives a partitioning based on the "<" relation.
- Partitioning usually combined with linking to point to the two halves.

Find 18

All keys in the left subtree are < the root
All keys in the right subtree are ≥ the root
Where’s the FBI?
J. Edgar Hoover Building
935 Pennsylvania Avenue, NW
Washington, DC, 20535-0001
Why is the DC partitioning bad?

- Everything interesting is in the northwest quadrant.
- Want a **balanced** partition!
- Another example: an unbalanced binary search tree: (becomes sequential search)

- How can we force a BST tree to be balanced if items are constantly being inserted and deleted?
Splay Trees (Sleator & Tarjan, 1985)

- no extra storage requirement
- simple to implement
- Main idea: move frequently accessed items up in tree
- amortized $O(\log n)$ performance
- worst case single operation is $\Omega(n)$
Splay Trees

**splay**\(T, k\): if \(k \in T\), then move \(k\) to the root using a particular set of transformations of the tree. Otherwise, move either the inorder successor or predecessor of \(k\) to the root.

Without knowing how \(splay\) is implemented, we can implement the dictionary ADT as follows:

- **find**\((T, k)\): \(splay(T, k)\). If \(\text{root}(T) = k\), return \(k\), otherwise return **not found**.

- **insert**\((T, k)\): \(splay(T, k)\). If \(\text{root}(T) = k\), return **duplicate!**; otherwise, make \(k\) the root and add children as in figure:

- **concat**\((T_1, T_2)\): Assumes all keys in \(T_1\) are < all keys in \(T_2\). \(Splay(T_1, \infty)\). Now root \(T_1\) contains the largest item, and has no right child. Make \(T_2\) right child of \(T_1\).

- **delete**\((T, k)\): \(splay(T, k)\). If root \(r\) contains \(k\), \(concat(\text{LEFT}(r), \text{RIGHT}(r))\).
Dictionary Operations, in pictures

- **find** ($T, k$): splay & check root
- **insert** ($T, k$): splay and insert just below root
- **delete** ($T, k$): splay & concat left & right subtrees
Implementing the Splay operation
Right rotation (at n)

Too tall!

Right rotation
(aka clockwise rotation)
Left Rotation (at $n$)

Left rotation (aka counterclockwise rotation)

Only a constant # of pointers need to be updated for a rotation: $O(1)$ time
Right & Left Rotations are Inverses

Right rotation:
- $i$ moves toward the root

Left rotation:
- $n$ moves toward the root

Diagram:
- Original tree configuration
- Right rotation
- Left rotation
Double Rotation

$k$ moves toward the root
**Splay Operation**

- $\text{Splay}(T, k)$: find $k$, walk back up root. Let $x$ be the current node.

- Cases:
  1. $x$ has no grandparent
  2. $x$ is left child of $\text{parent}(x)$, which is the right child of $\text{parent}^2(x)$.
  3. $x$ is left child of $\text{parent}(x)$, which is the left child of $\text{parent}(\text{parent}(x)) = \text{parent}^2(x)$

**Rotations with goal:**
move $x$ toward the root
Case 1: no grandparent:

Single rotation around p(x)
Case 2: zigzag (right, left):

Rotation around $p(x)$
Case 3: zigzag (left, left):

Rotation around $p^2(x)$

Rotation around $p(x)$
Splay Idea

• The find / insert / delete operations can be written in terms of the “splay” operation.

• Splay is implemented by doing a standard BST “find” and then applying particular rotations walking back up toward the root.

• This is somewhat like the idea of “path compression” for the tree-based union-find data structure: during a find, you flatten out the tree.

• Here: splay may actually make the tree worse, but over a series of operations the tree always gets better (e.g. a slow find results in a long splay, but this long splay tends to flatten the tree a lot).
Splay Notes

- Might make tree less balanced
- Might make tree taller
- So, how can they be good?
Amortized Analysis – Concept

- Some operations will be costly, some will be cheap
- Total area of \( m \) bars bounded by some function \( f(m,n) \).
  - \( m = \) number of operations, \( n = \) number of elements
- E.g. if area = \( O(m \log n) \), each operation takes \( O(\log n) \) amortized time
Analysis of the Splay operation
Node Ranks & Money Invariant

\[ w(u) := \text{weight of } u \]
\[ := \# \text{ nodes in the subtree rooted at } u \]
\[ \text{rank}(u) := \lfloor \log w(u) \rfloor \]

[x] means floor(x)

**Money Invariant:** we will always keep \( \text{rank}(u) \) dollars stored at every node.

Each rotation/double rotation costs $1.  
O(1) amount of work

Also have to spend $ to maintain invariant.
Idea:

**Thm.** It costs at most $3[\log n] + 1$ new dollars to splay, keeping the money invariant

- So, for every splay, we’re going to spend $O(\log n)$ new dollars; we might do more *work* than that if we use some of the $\$$ already in the tree.

- If we start with an empty tree, after $m$ splay operations, we’ll have spent $\leq m(3[\log n] + 1)$ dollars.

- The dollars pay for both:
  - the money invariant
  - cost of all the rotations (time)

- So, total time for $m$ splay operations is $O(m \log n)$. 
Thm. It costs $3[\log n] + 1$ new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at $x$ costs the following:

- zig: $3(rank^1(x) - rank(x)) + 1$
- zigzag: $3(rank^1(x) - rank(x))$
- zigzig: $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

$$
3(rank^1(x) - rank(x)) + 3(rank^2(x) - rank^1(x)) + 3(rank^3(x) - rank^2(x)) + \ldots + 3(rank^k(x) - rank^{(k-1)}(x)) + 1
$$

Then cost of a whole splay

$$
= 3(rank^k(x) - rank(x)) + 1 \\
\leq 3(rank^k(x)) + 1 \\
\leq 3[\log n] + 1
$$
**Thm.** It costs $3[\log n] + 1$ new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at $x$ costs the following:

- **zig:** $3(rank^1(x) - rank(x)) + 1$
- **zigzag:** $3(rank^1(x) - rank(x))$
- **zigzig:** $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

$$3(rank^1(x) - rank(x)) + 3(rank^2(x) - rank^1(x)) + 3(rank^3(x) - rank^2(x)) + 3(rank^k(x) - rank^{k-1}(x)) + 1$$

Then cost of a whole splay

$$= 3(rank^k(x) - rank(x)) + 1$$

$$\leq 3(rank^k(x)) + 1$$

$$\leq 3[\log n] + 1$$
**Thm.** It costs $3[\log n] + 1$ new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at $x$ costs the following:

- **zig:** $3(rank^1(x) - rank(x)) + 1$
- **zigzag:** $3(rank^1(x) - rank(x))$
- **zigzig:** $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

$$3(rank^1(x) - rank(x)) + 3(rank^2(x) - rank^1(x)) + 3(rank^3(x) - rank^2(x)) + 3(rank^k(x) - rank^{(k-1)}(x)) + 1$$

Then cost of a whole splay

$$= 3(rank^k(x) - rank(x)) + 1$$

$$\leq 3(rank^k(x)) + 1$$

$$\leq 3[\log n] + 1$$
**Thm.** It costs $3[\log n] + 1$ new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at $x$ costs the following:

- **zig:** $3(rank^1(x) - rank(x)) + 1$
- **zigzag:** $3(rank^1(x) - rank(x))$
- **zigzig:** $3(rank^1(x) - rank(x))$

Then cost of a whole splay $= 3(rank^1(x) - rank(x))$

Then cost of a whole splay $= 3(rank^1(x) - rank(x)) + 3(rank^2(x) - rank^1(x))$

Then cost of a whole splay $= 3(rank^1(x) - rank(x)) + 3(rank^2(x) - rank^1(x)) + 3(rank^3(x) - rank^2(x))$

Then cost of a whole splay $= 3(rank^k(x) - rank^{k-1}(x)) + 1$

Then cost of a whole splay $= 3(rank^k(x) - rank(x)) + 1$

Then cost of a whole splay $\leq 3(rank^k(x)) + 1$

Then cost of a whole splay $\leq 3[\log n] + 1$
**Thm.** It costs $3 \log n + 1$ new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at $x$ costs the following:

- zig: $3(\text{rank}^1(x) - \text{rank}(x)) + 1$
- zigzag: $3(\text{rank}^1(x) - \text{rank}(x))$
- zigzig: $3(\text{rank}^1(x) - \text{rank}(x))$

Then cost of a whole splay =

$$3(\text{rank}^1(x) - \text{rank}(x)) + 3(\text{rank}^2(x) - \text{rank}^1(x)) + 3(\text{rank}^3(x) - \text{rank}^2(x)) + \ldots + 3(\text{rank}^k(x) - \text{rank}^{k-1}(x)) + 1$$

Telescoping sum

Then cost of a whole splay

$$= 3(\text{rank}^k(x) - \text{rank}(x)) + 1$$

$$\leq 3(\text{rank}^k(x)) + 1$$

$$\leq 3 \log n + 1$$
**Thm.** It costs $3[\log n] + 1$ new dollars to splay, keeping the money invariant.

Suppose zig/zigzig/zigzag at $x$ costs the following:

- **zig:** $3(rank^1(x) - rank(x)) + 1$
- **zigzag:** $3(rank^1(x) - rank(x))$
- **zigzig:** $3(rank^1(x) - rank(x))$

Then cost of a whole splay =

$$3(rank^1(x) - rank(x)) + 3(rank^2(x) - rank^1(x)) + 3(rank^3(x) - rank^2(x)) + \cdots + 3(rank^k(x) - rank^{k-1}(x)) + 1$$  

Then cost of a whole splay

$$= 3(rank^k(x) - rank(x)) + 1$$

$$\leq 3(rank^k(x)) + 1 \quad \text{rank}^k(x) = \text{after } k \text{ steps, } x \text{ is at the root}$$

$$\leq 3[\log n] + 1$$
case 1: \(3(rank^1(x) - rank(x)) + 1\)

\[3(\text{rank}^1(x) - \text{rank}(x)) + 1\]

\(\text{+1 pays for the rotation}\)

\(\text{Extra } \$\text{ to keep the invariant is:}\)

\[\text{rank}^1(x) = \text{rank}(p(x))\]

Extra $ to keep the invariant is:

\[\text{rank}^1(x) + \text{rank}^1(p(x)) - (\text{rank}(x) + \text{rank}(p(x)))\]
case 1: \[3(rank^1(x) - rank(x)) + 1\]

Extra $ to keep the invariant is:

\[
rank^1(x) + rank^1(p(x)) - (rank(x) + rank(p(x))
\]

$ needed for x and p(x)
case 1: \(3(rank^1(x) - rank(x)) + 1\)

Extra $ to keep the invariant is:

\[rank^1(x) + rank^1(p(x)) - (rank(x) + rank(p(x)))\]

$ needed for x and p(x) $ already on x and p(x)

+1 pays for the rotation \(rank^1(x) = rank(p(x))\)
case 1: \(3(rank^1(x) - rank(x)) + 1\)

+1 pays for the rotation

Extra $ to keep the invariant is:

\[
(rank^1(x) + rank^1(p(x))) - (rank(x) + rank(p(x)))
\]

- $ needed for \(x\) and \(p(x)\)
- $ already on \(x\) and \(p(x)\)

\(r(x)\) after = \(r(p(x))\) before.
case 1: \[ 3(rank^1(x) - rank(x)) + 1 \]

+1 pays for the rotation

Extra $ to keep the invariant is:

\[
\text{rank}^1(x) + \text{rank}^1(p(x)) - (\text{rank}(x) + \text{rank}(p(x))
\]

\[= \text{rank}^1(p(x)) - \text{rank}(x) \]

\[\leq \text{rank}^1(x) - \text{rank}(x) \leq 3(\text{rank}^1(x) - \text{rank}(x))\]
case 2: $3(\text{rank}^1(x) - \text{rank}(x))$

If $\text{rank}^1(x) - \text{rank}(x) > 0$, then we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$ budgeted, so we have at least $\$1$ to pay for the rotations.

Otherwise see next slide.

$\$ needed to add $\leq \text{rank}^1(R) - \text{rank}(x)$
$\leq \text{rank}^1(x) - \text{rank}(x)$

But how do we pay for the rotations?
case 2: $3(rank^1(x) - rank(x))$

\begin{align*}
$ \text{needed to add} & \leq \ rank^1(R) - rank(x) \\
& \leq \ rank^1(x) - rank(x)
\end{align*}

But how do we pay for the rotations? 

If $rank^1(x) - rank(x) > 0$, then we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$ budgeted, so we have at least $\$1$ to pay for the rotations. 

Otherwise see next slide.
case 2: \(3(rank^1(x) - rank(x))\)

\[\text{\$ needed to add} \leq rank^1(R) - rank(x) \leq rank^1(x) - rank(x)\]

But how do we pay for the rotations?

If \(rank^1(x) - rank(x) > 0\), then we need to add \(\leq r^1(x) - r(x)\) but we have \(3(r^1(x) - r(x)) > 1\) budgeted, so we have at least \$1\ to pay for the rotations. Otherwise see next slide.
case 2: $3(rank^1(x) - rank(x))$

If $rank^1(x) - rank(x) > 0$, then we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$ budgeted, so we have at least $1$ to pay for the rotations. Otherwise see next slide.
case 2: $3(rank^1(x) - rank(x))$

\[ rank^1(R) - rank(x) \leq rank^1(R) - rank^1(x) - rank(x) \]

But how do we pay for the rotations?

If $rank^1(x) - rank(x) > 0$, then we need to add $\leq r^1(x) - r(x)$ but we have $3(r^1(x) - r(x)) > 1$ budgeted, so we have at least $1$ to pay for the rotations. Otherwise see next slide.
case 2: What if \((\text{rank}^1(x) - \text{rank}(x))\) is 0?

Since \(r^1(x) = r(x)\), and \(r^1(x) = r(R)\), we must have \(r(x) = r(R) = r(S)\):

Also, \(r^1(R) < r^1(x)\) or \(r^1(S) < r^1(x)\)

Suppose both \(r(R) = r(S) = q\).
Then \(R\) would have at least \(2^q\) nodes under it and \(S\) would have at least \(2^q\) nodes under it.
So \(x\) would have at least \(2(2^q)\) nodes under it and \(x\) would then have rank \(q+1\) instead of \(q\).

Had \(3q\) $ on tree before, now need < \(3q\), so have 1 $ left to pay for rotations.
case 3: $3(rank^1(x) - rank(x))$

$\text{needed to add:}$

$$r^1(x) + r^1(S) + r^1(R) - (r(x) + r(S) + r(R))$$

$\text{needed for moved nodes}$ $\quad$ $\text{already on moved nodes}$
case 3: 3(rank\(^1\)(x) - rank(x))

\[\text{x} \quad S \quad R\]

$ needed to add:

\[
r^1(x) + r^1(S) + r^1(R) - (r(x) + r(S) + r(R))
\]

$ needed for moved nodes $ already on moved nodes

\[
= r^1(S) + r^1(R) - r(x) - r(S)
\]

\[r^1(x) = r(R)\]
case 3: \(3(rank^1(x) - rank(x))\)

\[
\begin{align*}
\text{$ needed to add:} & \quad r^1(x) + r^1(S) + r^1(R) - (r(x) + r(S) + r(R)) \\
\text{\$ needed for moved nodes} & \quad $ already on moved nodes \\
\text{=} & \quad r^1(S) + r^1(R) - r(x) - r(S) \\
\leq & \quad r^1(x) + r^1(x) - r(x) - r(x) \\
\leq & \quad 2(r^1(x) - r(x)) \\
\end{align*}
\]

\[r^1(x) = r(R)\]

\[r^1(R) \leq r^1(S) \leq r^1(x)\]

\[r(x) \leq r(S)\]
case 3: What if $\text{rank}^1(x) - \text{rank}(x) = 0$?

Why must $R$ have rank $< q$?

Suppose not. Then $x$ has at least $2^q$ nodes under it, and $R$ has at least $2^q$ nodes under it.

So $S$ has at least $2(2^q) = 2^{q+1}$ nodes under it, so it should have rank $Q+1$, but it has rank $Q$, which is a contradiction.

So before we had $3q \cdot $ on the nodes, now we need only $\leq 2q + q - 1$
Additional Cost of Insert & Concat

- Cost of *insert* & *concat* more than the cost of a splay because may have to add $s to root to maintain invariant:

\[ \text{insert}(T, k): k \text{ has } n \text{ descendants, so need to put } [\log n] \text{ $ on } k \]

\[ \text{concat}(T_1, T_2): \text{ root gets at most } n \text{ new descendants from } T_2, \text{ so need to put } [\log n] \text{ dollars on root.} \]