Foundations of Algorithms

Amortized Analysis

Ira A. Fulton Schools of Engineering
Arizona State University

Amortized Analysis

Objectives

- Explain Amortized Analysis
- Explain different methods of Amortized Analysis

Incrementing a Binary Counter

- What is the running time of INCREMENT?

- The running time depends on the array of bits passed as input. If the first k bits are all 1s, then INCREMENT takes Θ(k) time.

- If B is an integer between 0 and n, then INCREMENT takes Θ(log n) time in the worst case, since the binary representation for n is exactly $\lfloor \lg n \rfloor$ + 1 bits long.

```
INCREMENT(B):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
```

Counting from 0 to n: The Aggregate Method

- Use INCREMENT algorithm to count from 0 to n.
- Using the worst-case running time for each call, we get O(nlog n) total running time: not the best we can do!
- The total number of bit-flips for the entire sequence is:

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

- On average, each call to INCREMENT flips only two bits, and so runs in constant time.

Amortization

- This averaging idea is called amortization.
- The amortized cost of each INCREMENT is O(1).
- Amortization is used by accountants to average large one-time costs over long periods of time.
- An example of amortization is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time.

Amortized Analysis: Aggregate Method

- Find the worst case running time, $T(n)$, for a sequence of n operations.

- The amortized cost of each operation is $T(n)/n$.

Amortized Analysis: Accounting Method

- Suppose it costs us a dollar to toggle a bit, so we can measure the running time of our algorithm in dollars.

- Instead of paying for each bit flip when it happens, we charge two dollars when we want to set a bit from 0 to 1.
  - One of those dollars is spent changing the bit from 0 to 1.
  - The other is stored as credit until we need to reset the same bit to 0.
  - We always have enough credit to pay for the next INCREMENT.

- The amortized cost of an INCREMENT is the total charge it incurs, which is exactly two dollars, since each INCREMENT changes just one bit from 0 to 1.

Amortized Analysis: Potential Method

- The most powerful method (and the hardest to use) builds on a physics metaphor of ‘potential energy’.

- Instead of associating costs or charges with particular operations, we consider prepaid work as potential that can be spent on later operations. The potential is a function of the entire data structure.

Amortized Analysis: Potential Method

- Let D_i denote our data structure after i operations, and let Φi denote its potential.
- Let c_i denote the actual cost of the ith operation, which changes D_{i-1} into D_i.
- The amortized cost of the ith operation, denoted a_i, is defined as the actual cost plus the change in potential:

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

Amortized Analysis: Potential Method

- The total amortized cost of n operations is the actual total cost plus the total change in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

- Define a potential function so that $\Phi_0 = 0$ and $\Phi_i >= 0$ for all i.

- The total actual cost of any sequence of operations will be less than the total amortized cost.

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

Potential Method on Binary Counter

- We can apply the potential method to Binary Counter.

- The potential Φ_i after the ith INCREMENT is equal to the number of bits with value 1.

- Initially, all bits are zero, so $\Phi_0 = 0$ and $\Phi_i > 0$ for all i.

Potential Method on Binary Counter

- The actual cost of an INCREMENT and the change in potential then become

c_i = #bits changed from 0 to 1 + #bits changed from 1 to 0

$\Phi_i - \Phi_{i-1}$ = #bits changed from 0 to 1 - #bits changed from 1 to 0

- Thus, the amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 * (\text{\#bits changed from 0 to 1})$$

- The INCREMENT changes only one bit from 0 to 1. Hence, the amortized cost of INCREMENT is 2.

Summary

Foundations of Algorithms

Splay Trees

Ira A. Fulton Schools of Engineering
Arizona State University

Splay Trees

Objectives

- Review Dynamic Binary Search Trees and Balanced Search Trees
- Explain Splay Trees
- Amortized analysis of Splay Trees

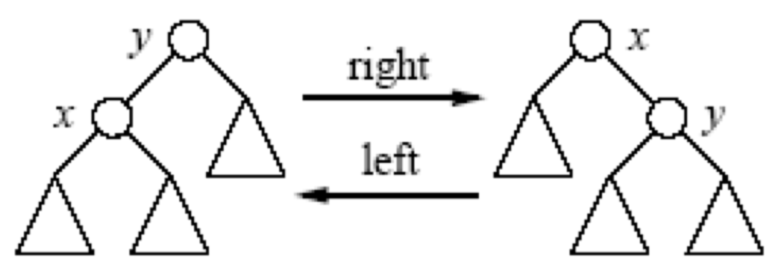
Dynamic Binary Search Trees

Definitions

- Every internal node has exactly two children.
- *depth d(v)* = distance from the root to node v.
  height h(v) = distance from v to the farthest leaf in the subtree rooted at v.
- The height (or depth) of a tree is just the height of the root
- The *size* |*v*| *of v* is the number of nodes in the subtree rooted at v.
- n = size of the whole tree (total number of nodes)
- Minimum height of any binary tree is $\lceil \log n \rceil$
- Balanced search tree: tree of height O(log *n*). Balanced search trees support search, insertion and deletion in O(log n) worst-case time.

Rotations

- Balanced Binary Search Trees use rotations to maintain the tree balanced.
- A (single) rotation adjusts the shape of the tree locally. A rotation at a node x decreases its depth by one and increases its parent's depth by one.
- Each rotation can be performed in constant time.



A right rotation at x and a left rotation at y are inverses.

Double Rotations

- There are two types of double rotations: roller-coaster and zig-zag.

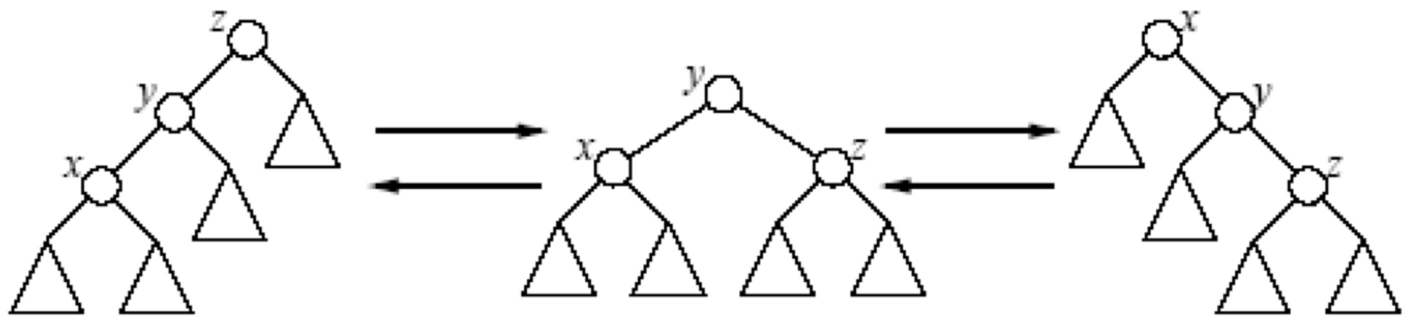
- A roller-coaster at a node x consists of a rotation at x's parent followed by a rotation at x, both in the same direction.

- A zig-zag at x consists of two rotations at x, in opposite directions.

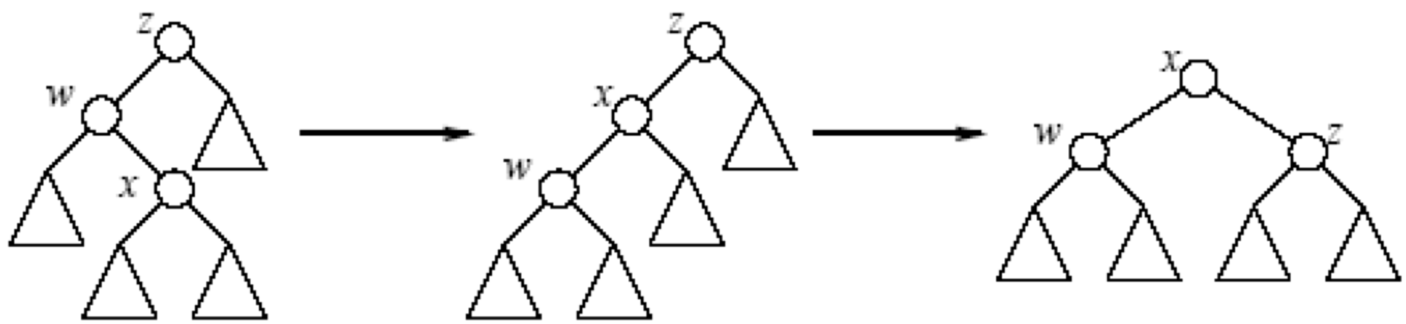
- Each double rotation decreases the depth of x by two, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation.

- Either type of double rotation can be performed in constant time.

Double Rotations



A right roller-coaster at x and a left roller-coaster at z.



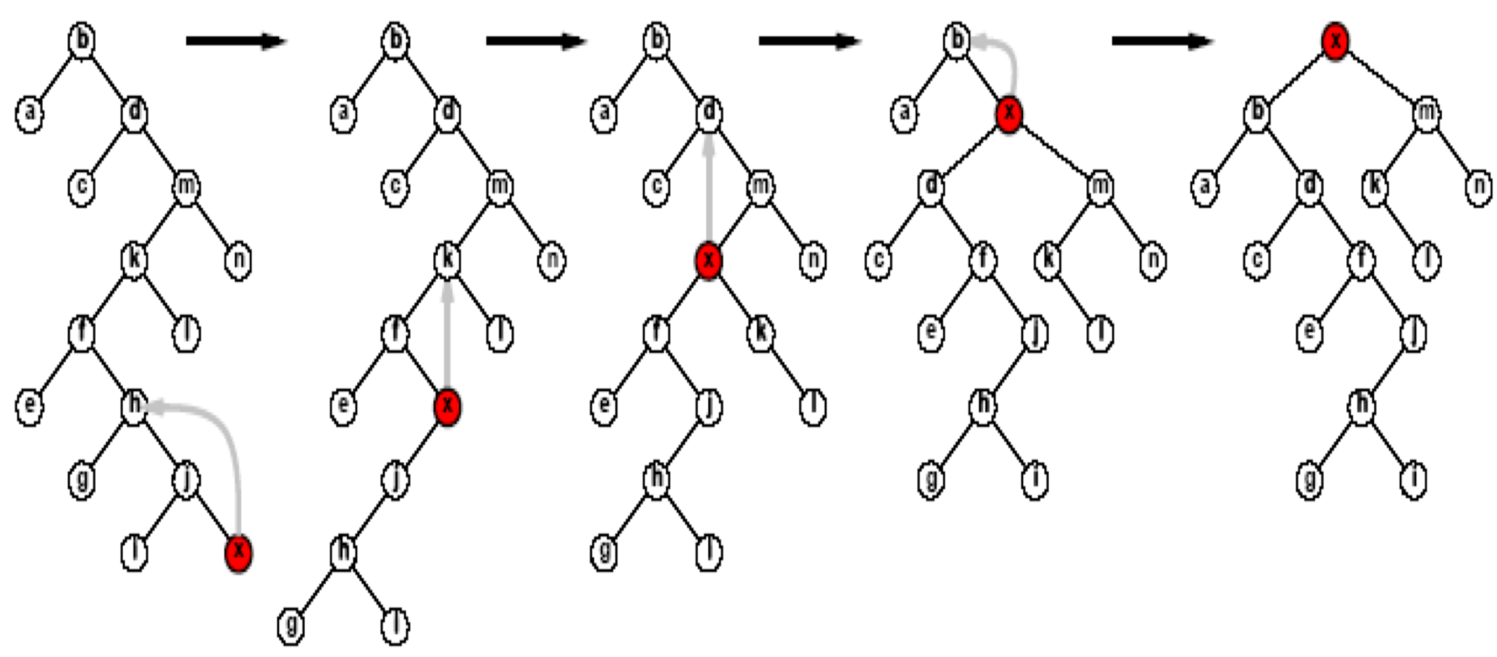
A zig-zag at x. The symmetric case is not shown.

Splaying

- A splay operation moves an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end.

- Splaying a node v requires time proportional to d(v), which is the depth before splaying.

Splaying Example



Splaying a node. Irrelevant subtrees are omitted for clarity.

Splay Trees

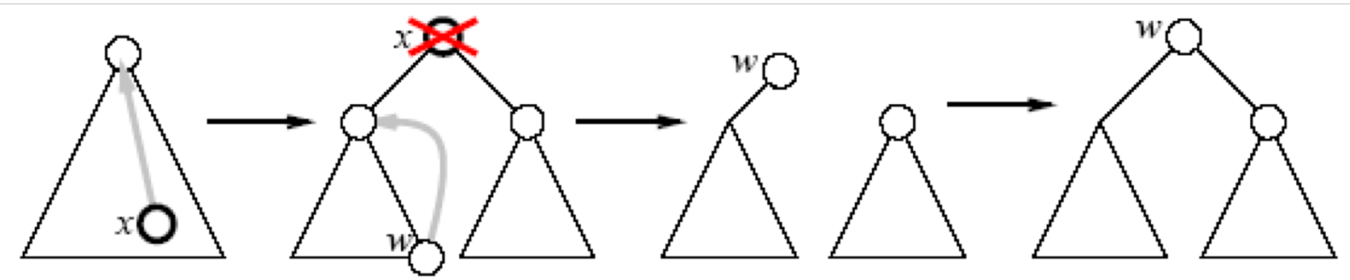
- A Splay Tree is a binary search tree that is kept more or less balanced by splaying. Intuitively, after we access any node, we move it to the root with a splay operation as follows:

- **Search:**
  - Use standard binary search tree search.
  - Find the node containing the key, or its predecessor or successor if the key is not present.
  - Splay whichever node was found.

- **Insert:**
  - Insert a new node using the standard binary search tree insert algorithm, then splay that node.

Splay Trees

- **Delete:**
  - Find the node x to be deleted, splay it, and then delete it. This splits the tree into two subtrees, one with keys <= x, the other with keys >= x.
  - Find the node w in the left subtree with the largest key (i.e., the predecessor of x in the original tree), splay it, and finally join it to the right subtree



Deleting a node in a splay tree.

Splay Trees

- Each search, insertion, or deletion consists of constant number of operations of the form: “walk down to a node, and then splay it up to the root.”

- Since the walk down is clearly cheaper than the splay, all we need to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

- We will use the potential method. The rank of any node v is defined as r(v) = $\lfloor \log |v| \rfloor$. We define the potential of a splay tree *T* at time *t* to be the sum of the ranks of all its nodes:

$$\Phi_t(T) = \sum_v r(v) = \sum_v \lfloor \log |v| \rfloor$$

Splay Trees

- $r(v)$ = rank of v before a (single or double) rotation. $r'(v)$ = rank of v after the rotation is performed.

- **Lemma.** The amortized cost of a single rotation at v is at most $1+3r'(v) - 3r(v)$, and the amortized cost of a double rotation at v is at most $3r'(v) - 3r(v)$.

Splay Trees

- By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node v is at most

$$1 + 3r_{final}(v) - 3r_{start}(v)$$

- After the splay, v becomes the root, hence

$$r_{final}(v) = \lfloor \log n \rfloor$$

- Amortized cost of a splay is at most

$$3 \log n + 1 = O(\log n)$$

- Thus, every insertion, deletion, or search in a splay tree takes amortized time, $O(\log n)$, which is optimal.

Summary