CSC 226

Algorithms and Data Structures: II
Midterm Review

Tianming Wei
twei@uvic.ca
ECS 466

Midterm

- Friday, June 21 in HSD A240
- 9:30 a.m. to 10:20 a.m. (50 minutes)
- 4 Questions – 10 marks each
  - Question 1 – Miscellaneous (5 parts)
  - Question 2 – Miscellaneous (3 parts)
  - Question 3 – Search Trees (2 parts)
  - Question 4 – MSTs (2 parts)

Midterm

- Closed book
- Bring a calculator
- No washroom break
- Don’t get stuck on one question, skip to the next one.

Course Division – CSC 226

The course will have four modules:

1. Topics from Sorting and Discrete Math

2. Advanced Graph Algorithms

3. Text-Processing Algorithms

4. Algorithms for *Hard* Problems

Course Topics – CSC 226

- Introduction and asymptotic review (1.4)
- Sorting revisited (2.2)
- Discrete and Combinatorial Math
- Balanced Binary Search Trees (3.3)
- Undirected graphs (4.1)
- Directed graphs (4.2)
- Minimum spanning trees (4.3)
- Union-find (1.5)
- Shortest path algorithms (4.4)
- Network flow (6.4)
- Longest Common Subsequence
- Tries (5.2)
- Substring search (5.3)
- Data compression (5.5)
- Planar graph algorithms (6.5)
- Coping with intractability (6.6)

Asymptotic Notation Review

- Big-Oh
- Big-Omega
- Big-Theta
- Little-oh
- Little-omega

Most Common Functions in Algorithm Analysis Ordered by Growth

$$1 \quad \log\log n \quad \log \mathrm{n} \quad \sqrt{\mathrm{n}}$$

$$n \quad n\log n \quad \mathrm{n}^{2} \quad \mathrm{n}^{2.31}$$

$$n^{3} \quad n^{k}, \text{for } k > 3$$

$$2^{n} \quad 3^{n} \quad n! \quad n^{n}$$

Little-Oh Notation

Let f: IN→IR and g: IN→IR.

f(n) is o(g(n)) if and only if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Ex: $n \log n$ is $o(n^2)$ (Hint: l'Hopital's Rule)

Notation	Name	Description	Definition	Limit
$f(n) \in O(g(n))$	Big Oh	f is bounded above by a constant factor of g	$\exists c > 0, \exists n_0 > 0$ s.t. $f(n) \leq cg(n)$, $\forall n \geq n_0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$
$f(n) \in o(g(n))$	Little Oh	f is dominated by g asymptotically	$\forall c > 0, \exists n_0 > 0$ s.t. $f(n) \leq cg(n)$, $\forall n \geq n_0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
$f(n) \in \Omega(g(n))$	Big Omega	f is bounded below by a constant factor of g	$\exists c > 0, \exists n_0 > 0$ s.t. $f(n) \geq cg(n)$, $\forall n \geq n_0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$
$f(n) \in \omega(g(n))$	Little Omega	f dominates g asymptotically	$\forall c > 0, \exists n_0 > 0$ s.t. $f(n) \geq cg(n)$, $\forall n \geq n_0$	$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$
$f(n) \in \Theta(g(n))$	Big Theta	f is bounded below and above by a constant factor of g	$\exists c_1, c_2 > 0, \exists n_0 > 0$ s.t. $c_1 g(n) \leq f(n) \leq c_2 g(n)$, $\forall n \geq n_0$	$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$

Different types of Sorting Algorithms

- Comparison Based Sorting
  - sorting algorithm that sorts based only on comparisons
  - elements to be sorted must satisfy total order properties

- Integer Sorting
  - sorting algorithm that sorts a collection of data values by numeric keys, each of which is an integer

	Type of Sorting Algorithm	Worst Case Time	Best Case Performance	Average Case Performance	Properties
Insertion Sort	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	adaptive, in place, stable, online
Bubblesort	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	in place
Selection Sort	Comparison Based Sorting	$O(n^2)$	$O(n^2)$	$O(n^2)$	in place
Binary Insertion	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	adaptive, in place
Shakersort	Comparison Based Sorting	$O(n^2)$	$O(n)$	$O(n^2)$	stable, in place
Shellsort	Comparison Based Sorting	$O(n^2)$	$O(n \log n)$		in place
Quicksort	Comparison Based Sorting	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	in place
Heapsort	Comparison Based Sorting	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	in place
Mergesort	Comparison Based Sorting	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	not in place
Bucketsort	Integer Sorting	$O(n+k)$		$O(n+k)$	can be implemented such that stable
Radixsort	Integer Sorting	$O(dn)$			stable

How fast can we sort?

Can we say that it is impossible to sort faster than $\Omega\ (n\ logn)$ in the worst case?

If we could prove it, then $\Omega\ (n\ logn)$ denotes the lower bound for comparison based sorting.

Fundamental Principles of Counting

Permutations

In general, the number of permutations of size r from n distinct objects, where $0 \leq r \leq n$, is given by

$$P(n, r) = \frac{n!}{(n-r)!}$$

- Note: $P(n, 0) = \frac{n!}{n!} = 1$ and $P(n, n) = \frac{n!}{0!} = n!$

Fundamental Principles of Counting

Combinations

In general, the number of combinations of r objects from n distinct objects, where $0 \leq r \leq n$, is given by

$$\binom{n}{r} = C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!\,(n-r)!}$$

- Note: $C(n,0) = \frac{n!}{0!n!} = 1$ and $C(n,n) = \frac{n!}{n!0!} = 1$

Fundamental Principles of Counting

The Binomial Theorem

If x and y are variables and n a positive integer, then

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

- Proof: Consider $(x+y)^n = \underbrace{(x+y)\cdots(x+y)}_{n\ times}$.

  For any $0 \le k \le n$, the number of combinations of k x's is $\binom{n}{k}$.

Fundamental Principles of Counting

Combinations with Repetition

In general, taking n distinct objects, with repetition, taken r at a time can be done in

$$\binom{n+r-1}{r} = \frac{(n+r-1)!}{r!\,(n-1)!}$$

ways.

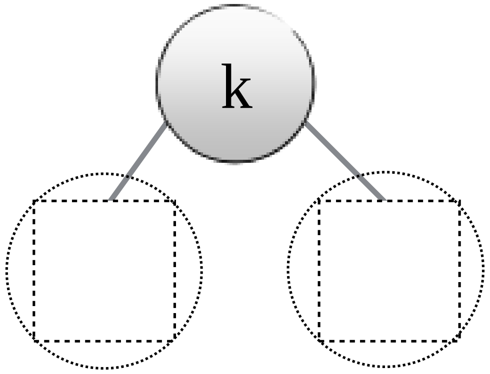
The Pigeonhole Principle

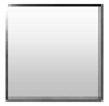
If m pigeons occupy n pigeonholes and $m > n$, then at least one pigeonhole has two or more pigeons roosting in it.

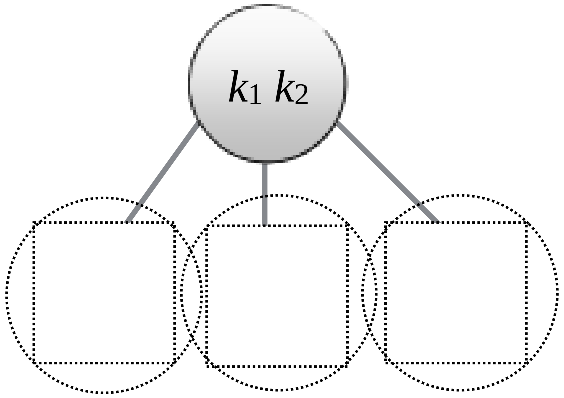
Balanced Search Trees

- Why balanced search trees?
  - unbalanced search trees are not efficient due to height $O(n)$
- Examples
  - AVL trees
  - 2-3 trees & red-black trees

Definition (2-3 tree)



- A *2-3 search tree* is a tree that is
  - either empty 
  - or a *2-node*, with one key k (and associated value) and two links: a left link to a 2-3 search tree with keys smaller than k, and a right link to a 2-3 search tree with keys larger than k
  - or a *3-node*, with two keys $k_1 < k_2$ (and associated values) and three links: a left link to a 2-3 search tree with keys smaller than k_1, a middle link to a 2-3 search tree with keys larger than k_1 and smaller than k_2, and a right link to a 2-3 search tree with keys larger than k_2



2-3 trees: insertion of an element with key *k*

- We only insert if key *k* is not yet in the tree. The search for key *k* returns a leaf.

- **Case 1.** If the leaf is root, then the tree is empty and the leaf (root node) is replaced by a 2-node with key *k*

- **Otherwise**, the search terminates in a leaf with parent node *v*.

- We distinguish two cases

  - **Case 2.** *v* is a 2-node

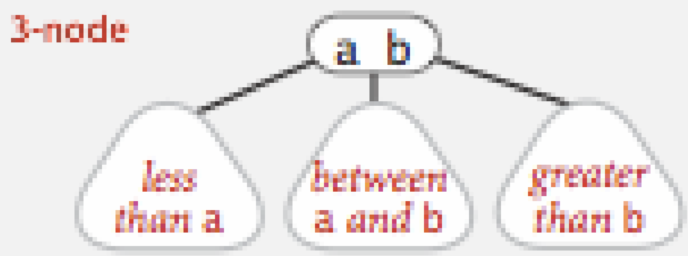
  - **Case 3.** *v* is a 3-node

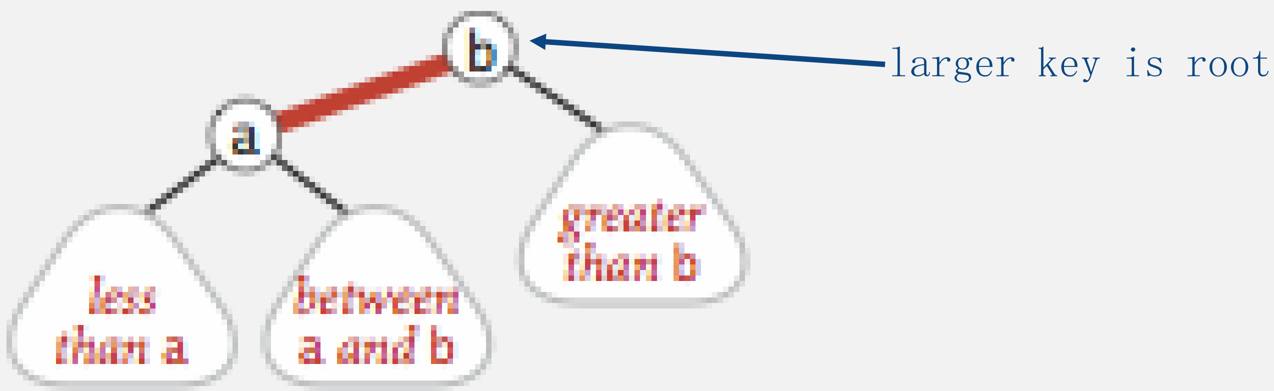
Case 3. *v* is a 3-node

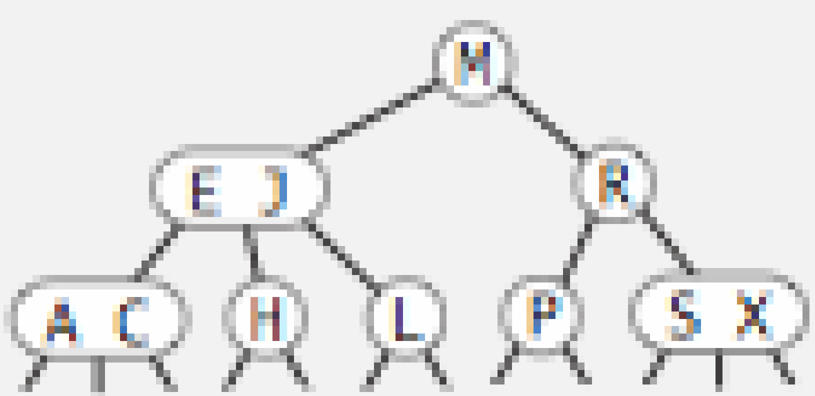
- We distinguish the following cases
  - **Case 3.1** *v* is root
  - **Case 3.2** *v*'s parent is a 2-node
  - **Case 3.3** *v*'s parent is a 3-node
  - These are all cases since the search tree is perfectly balanced.

Left-leaning red-black BSTs (Guibas-Sedgewick 1979 and Sedgewick 2007)

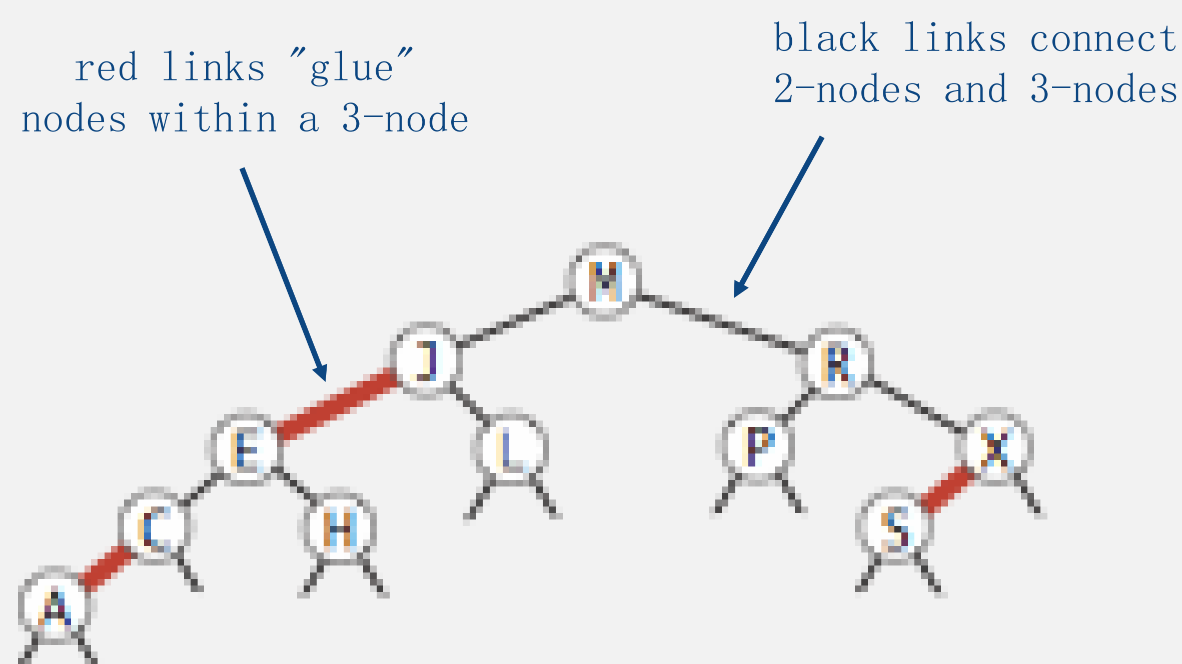
1. Represent 2-3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.







2-3 tree



corresponding red-black BST

Definition: Red-Black Tree

- A red-black tree is a binary search tree where each link/edge is either red or black. Further
  - All red links lean left
  - No node has two red links connected to it
  - The tree has a balance: every path from the root to a leaf has the same number of black links
  - Links to leaves are black

Inserting into a red-black tree

- Insert just as in BST
- but: link/edge to new node is red
- rotations and color flipping (depending on case)

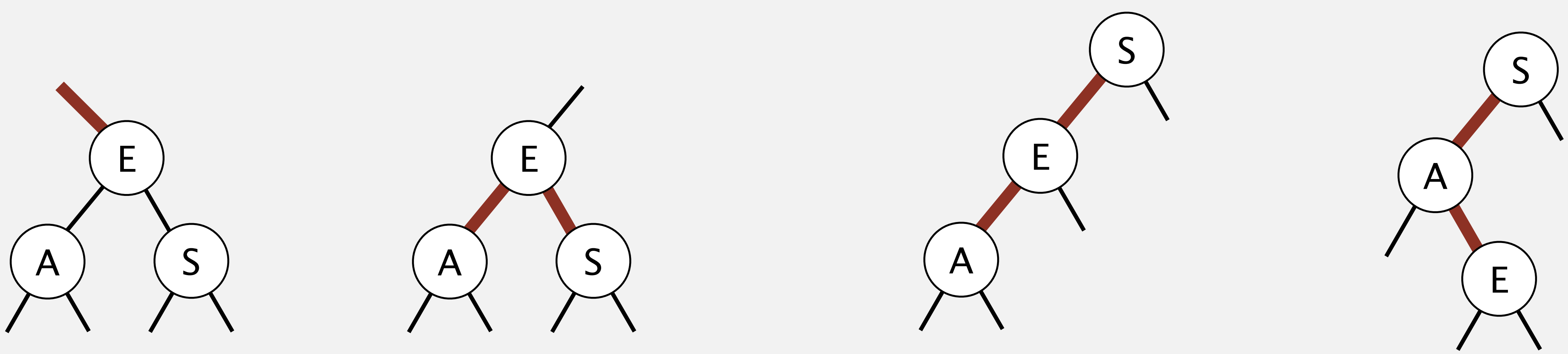
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

During internal operations, maintain:

- Symmetric order.
- Perfect black balance.

  [but not necessarily color invariants]



right-leaning red link

two red children (a temporary 4-node)

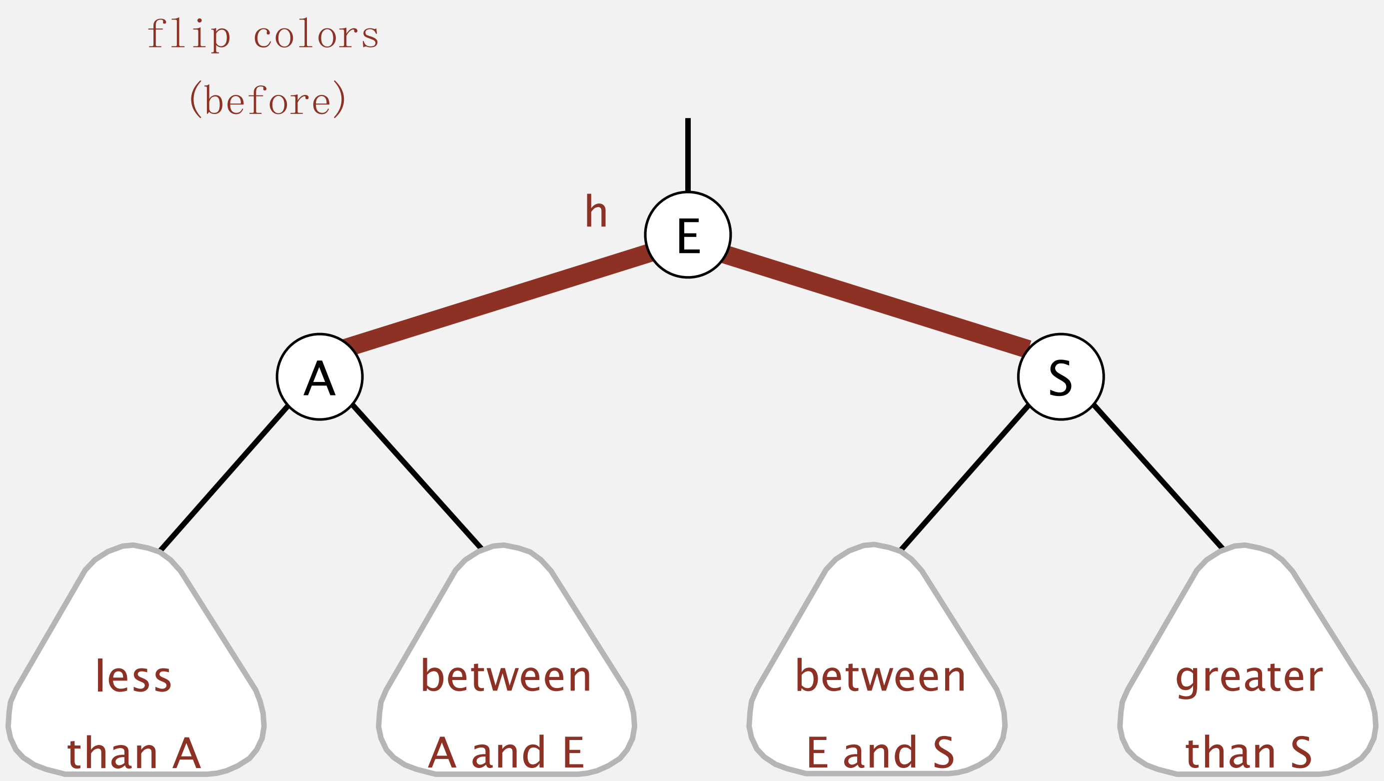
left-left red (a temporary 4-node)

left-right red (a temporary 4-node)

How? Apply elementary red-black BST operations: rotation and color flip.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

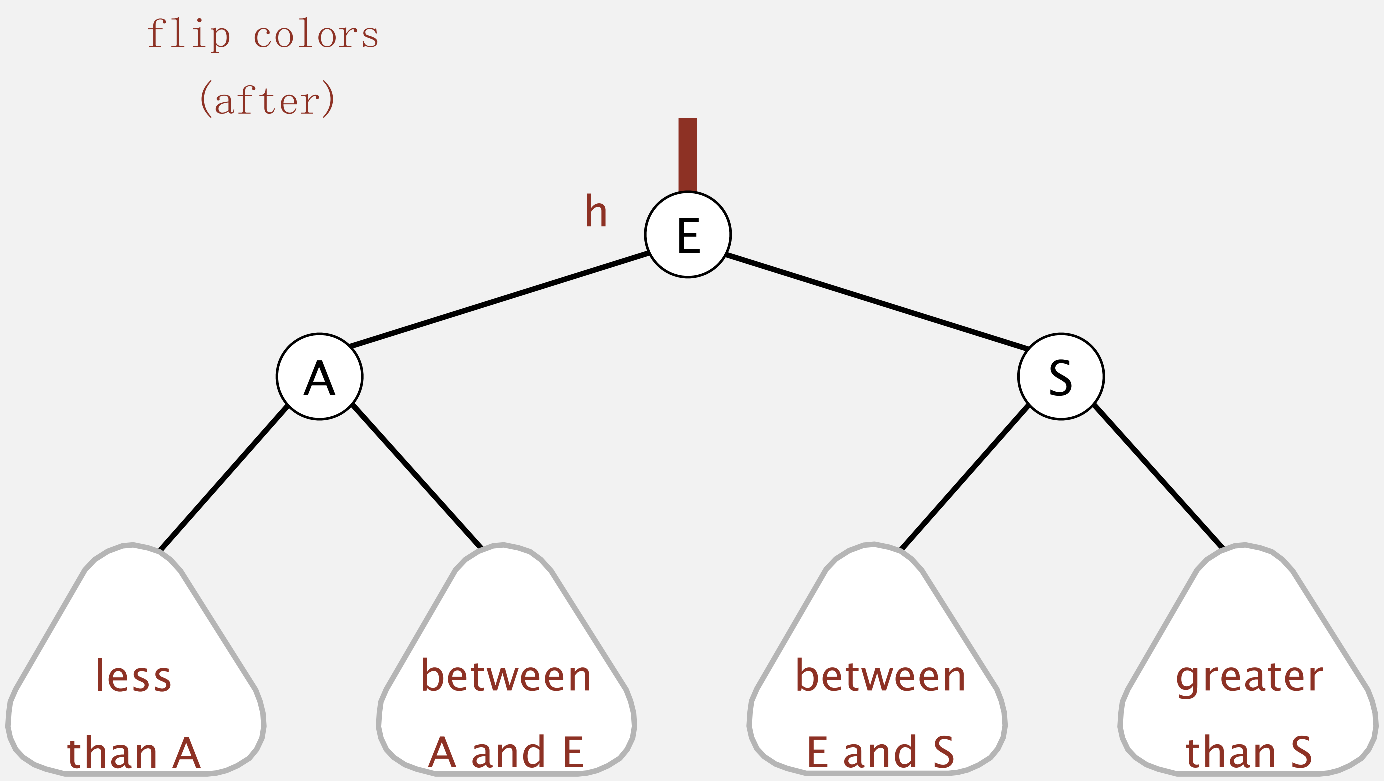



```java
private void flipColors(Node h)
{
   h.color = RED;
   h.left.color = BLACK;
   h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

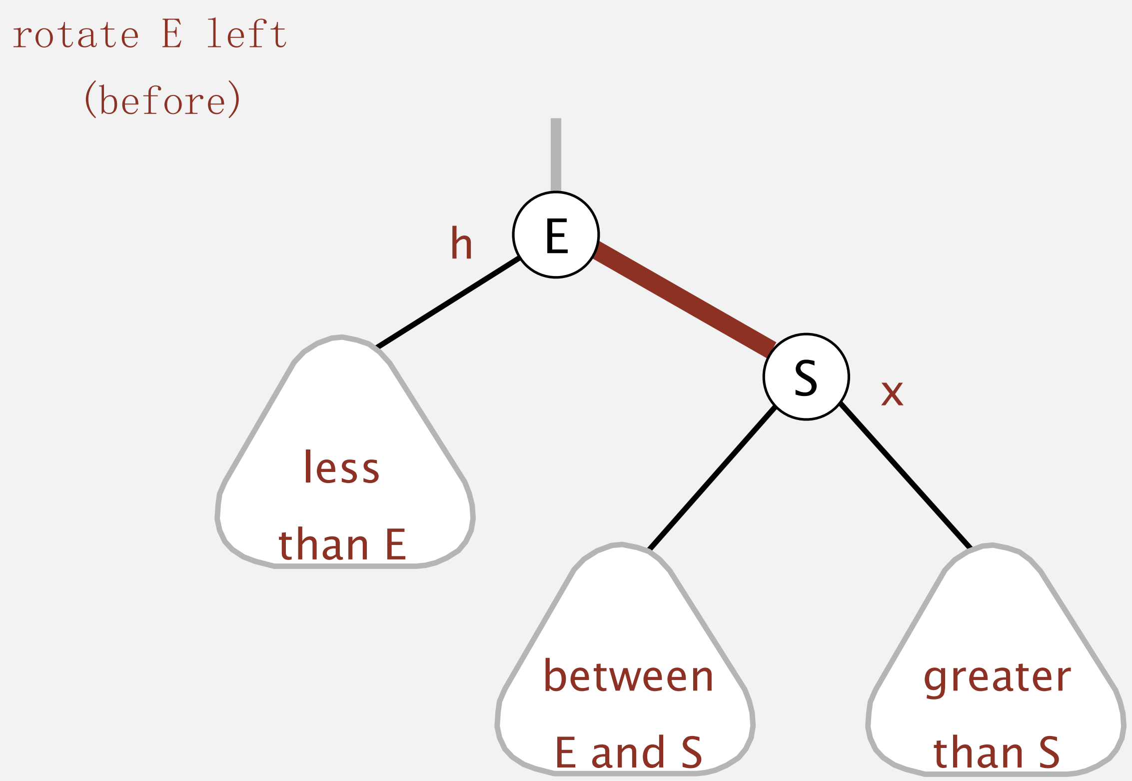
Color flip. Recolor to split a (temporary) 4-node.




```
private void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

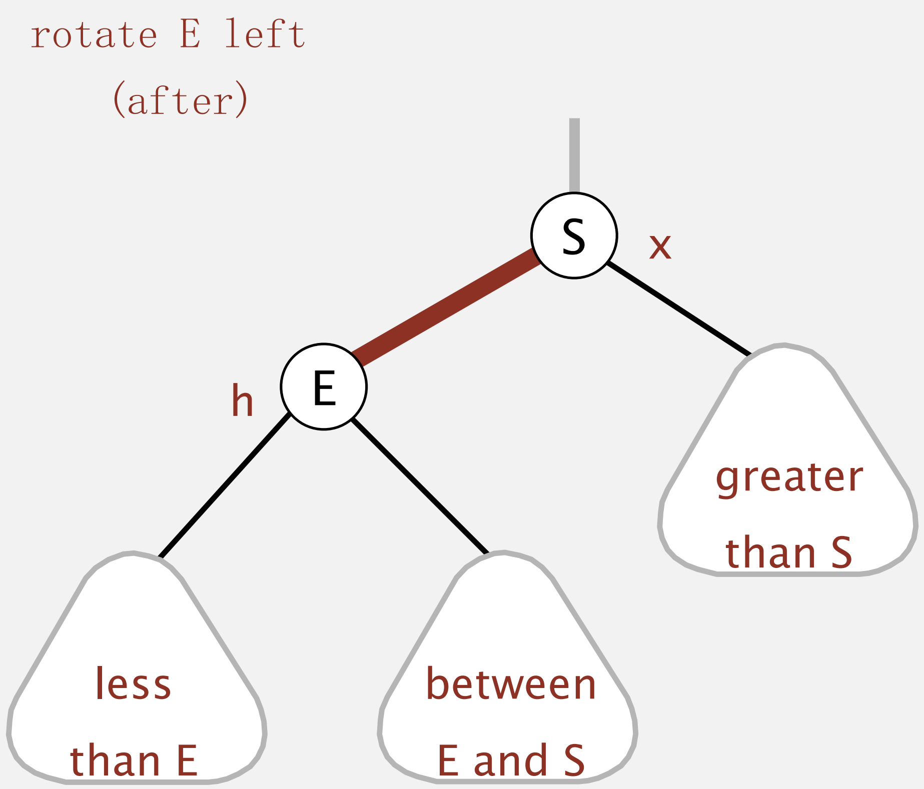



```
private Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

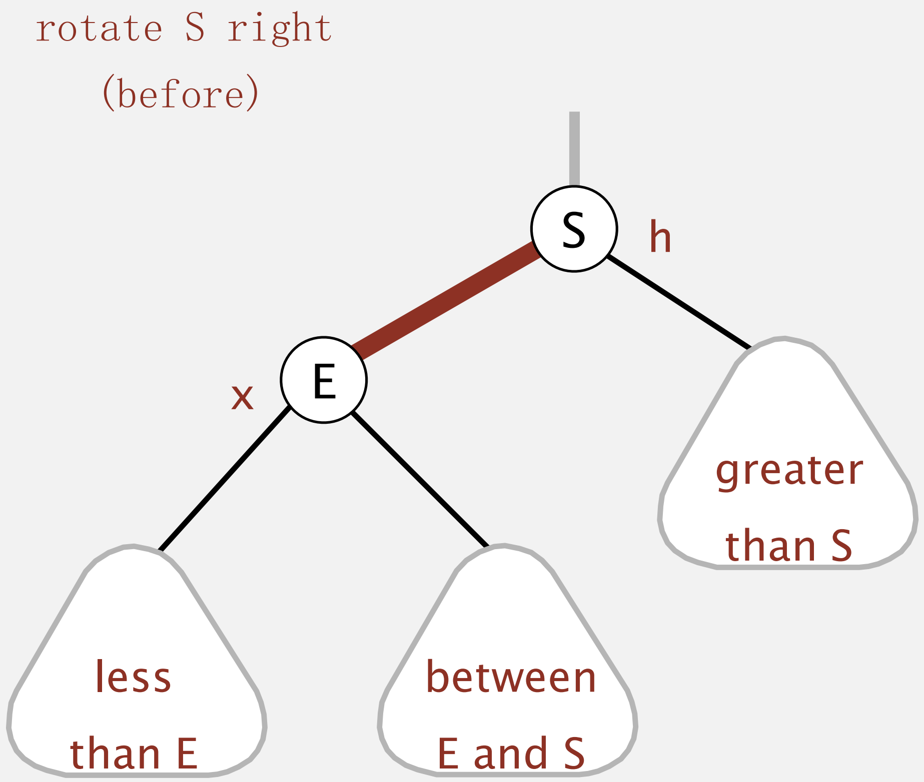
Left rotation. Orient a (temporarily) right-leaning red link to lean left.




```
private Node rotateLeft(Node h)
{
   Node x = h.right;
   h.right = x.left;
   x.left = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

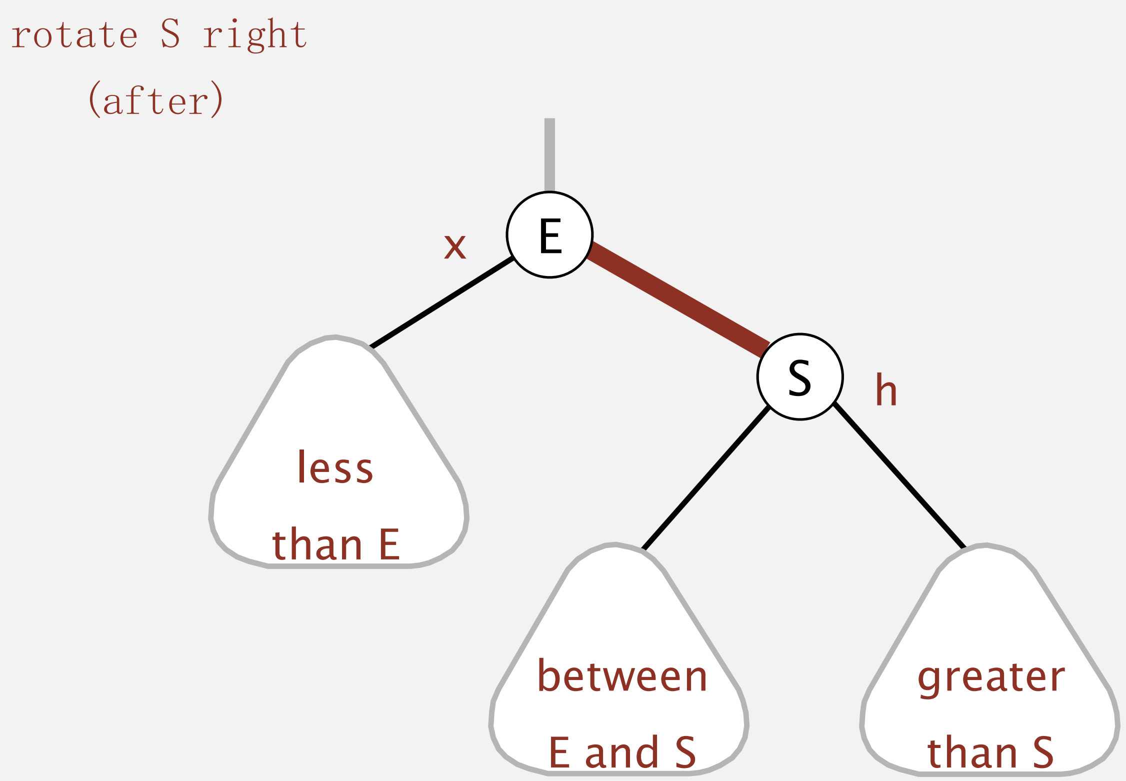
Right rotation. Orient a left-leaning red link to (temporarily) lean right.




```
private Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

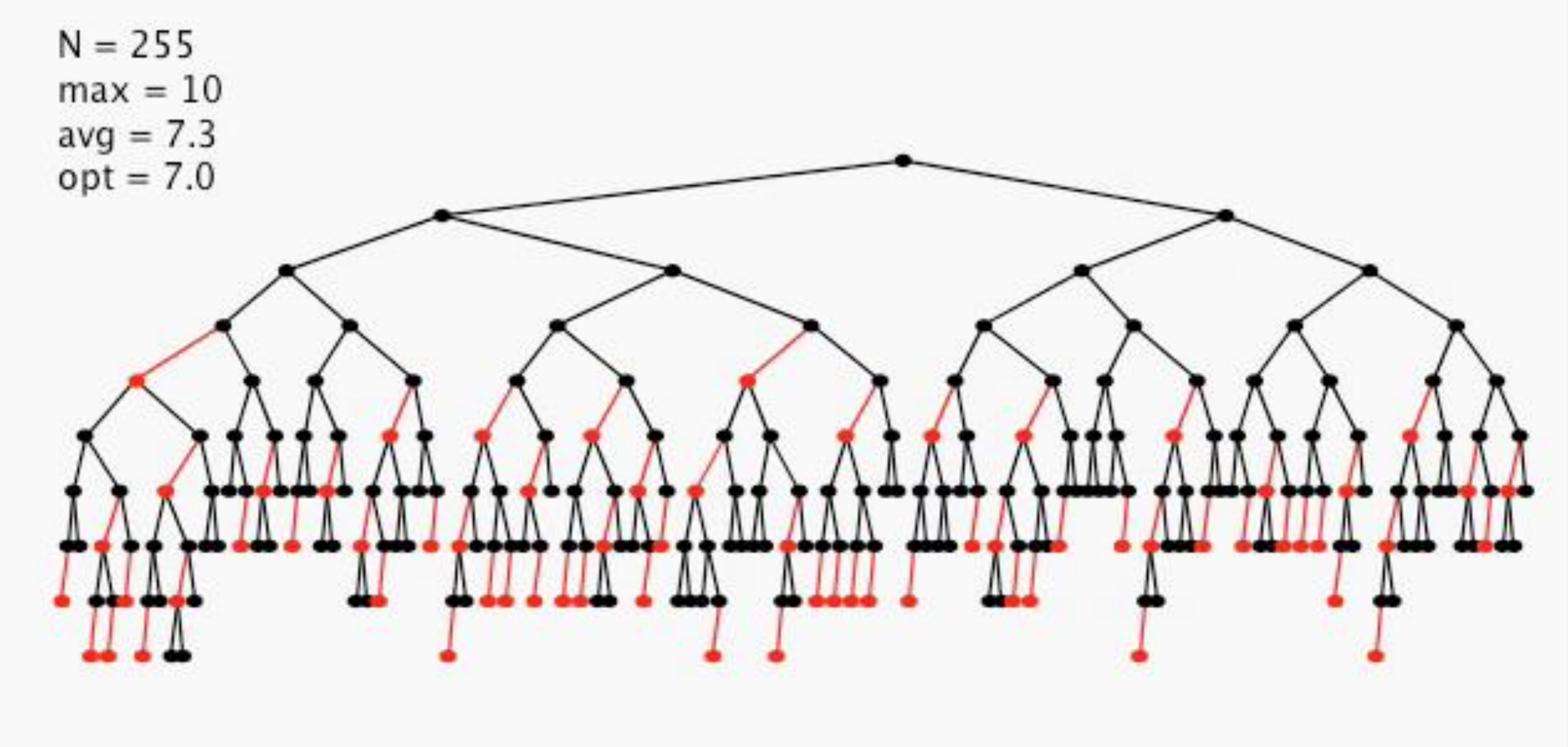
Right rotation. Orient a left-leaning red link to (temporarily) lean right.




```
private Node rotateRight(Node h)
{
   Node x = h.left;
   h.left = x.right;
   x.right = h;
   x.color = h.color;
   h.color = RED;
   return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree: visualization



255 random insertions

Balanced trees in the wild

Red-black trees: widely used as system symbol tables

- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: `linux/rbtree.h`.

B-Trees: widely used for file systems and databases

- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL

Bottom line: ST implementation with lg N guarantee for all ops.

- Algorithms are variations on a theme: rotations when inserting.
- Easiest to implement, optimal, fastest in practice: LLRB trees
- Abstraction extends to give search algorithms for huge files: B-trees

Weighted Graphs

- A ***weighted graph*** is a graph model where we associate ***weights*** (or ***costs***) with each edge

- Minimum spanning trees

- Shortest Paths

Minimum Spanning Tree Definition

- *Input*: A weighted connected graph $G = (V, E)$ consisting of vertices (or nodes), V, and edges, E, with positive integer edge weights

- *Output*: A *minimum spanning tree* (*MST*) $T = (V, E_T)$, that is T is a connected subgraph of G ($E_T \subseteq E$) such that T is acyclic, and T is *lightest*

Minimum Spanning Tree algorithms

- 1926 Barůvka $O(m \log n)$
- 1930 Prim-Jarník's
  - 1930 Jarník
  - 1957 Dijkstra
  - 1959 Prim
  - 1964 with Heaps $O(m \log n)$
  - 1987 Fredman and Tarjan with Fibonacci Heaps $O(m+n \log n)$
- 1956 Kruskal's algorithm
  - 1956 Kruskal
  - 1974 Aho, Hopcroft and Ullman with Union-Find Disjoint Set $O(m \log n)$
- 1975 Yao $O(m \operatorname{loglog} n)$
- 1976 Cheriton and Tarjan $O(m \operatorname{loglog} n)$
- 1995 Karger, Klein and Tarjan Randomized MST based on Barůvka and Kruskal $O(m)$
- 2000 Chazelle $O(m\ \alpha(m,n))$

n: number of vertices
m: number of edges

Prim's Algorithm
Idea

- Initialize tree with single chosen vertex
- Grow tree by finding lightest edge not yet in tree and connect it to tree; repeat until all vertices are in the tree
- *Example of greedy algorithm*

Kruskal's Algorithm Idea

- Initialize a forest consisting of all nodes
- Pick a (non-selected) minimum weight edge and, if it connects two different trees of the forest, select it, otherwise discard it; repeat
- *Example of greedy algorithm*

Borůvka's Algorithm Idea

- Often assume every edge has a unique weight.
- Initially, each vertex is considered a separate component.
- The algorithm merges disjoint components as follows; repeating the step until only one component exists.
- In each step, every component is merged with some other using the cheapest outgoing edge of the given component.

Kruskal's algorithm requires an efficient way of testing whether an edge creates a cycle with the edges already selected.

- The union-find data structure helps do this.

Quick-find [eager approach]

Data structure.

if and only if

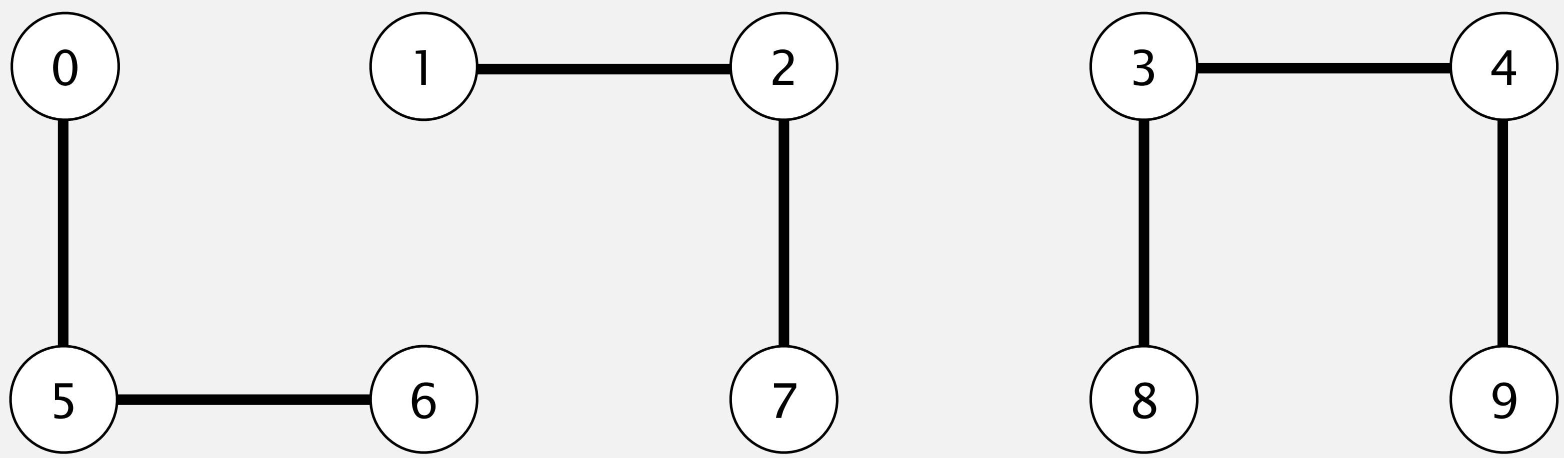
- Integer array `id[]` of length n.
- Interpretation: `id[p]` is the id of the component containing `p`.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

0, 5 and 6 are connected

1, 2, and 7 are connected

3, 4, 8, and 9 are connected



Quick-find [eager approach]

Data structure.

- Integer array id[] of length n.
- Interpretation: id[p] is the id of the component containing p.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

Find. What is the id of p?

Connected. Do p and q have the same id?

id[6] = 0; id[1] = 1

6 and 1 are not connected

Union. To merge components containing p and q, change all entries whose id equals id[p] to id[q].

	0	1	2	3	4	5	6	7	8	9
id[]	1	1	1	8	8	1	1	1	8	8

after union of 6 and 1

problem: many values can change

Quick-union [lazy approach]

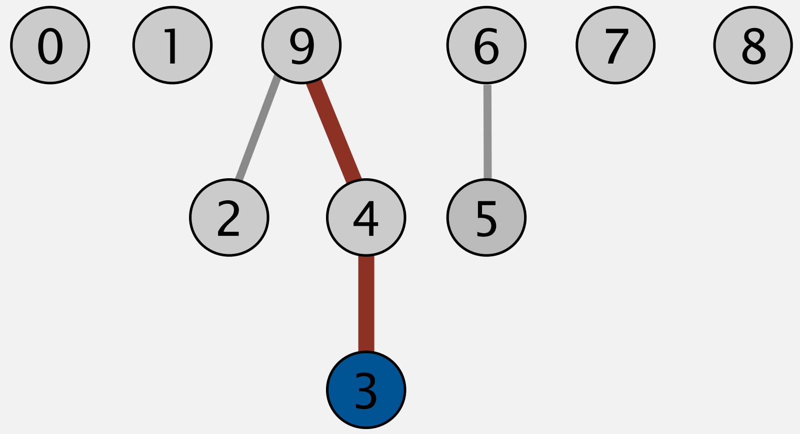
Data structure.

- Integer array id[] of length n.
- Interpretation: id[i] is parent of i.
- Root of i is id[id[id[...id[i]...]]].

keep going until it doesn't change

(algorithm ensures no cycles)

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



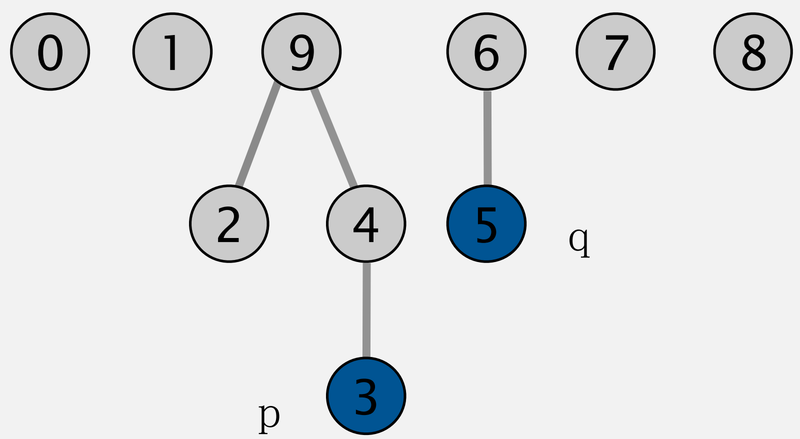
parent of 3 is 4

root of 3 is 9

Data structure.

- Integer array `id[]` of length n.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



root of 3 is 9

root of 5 is 6

3 and 5 are not connected

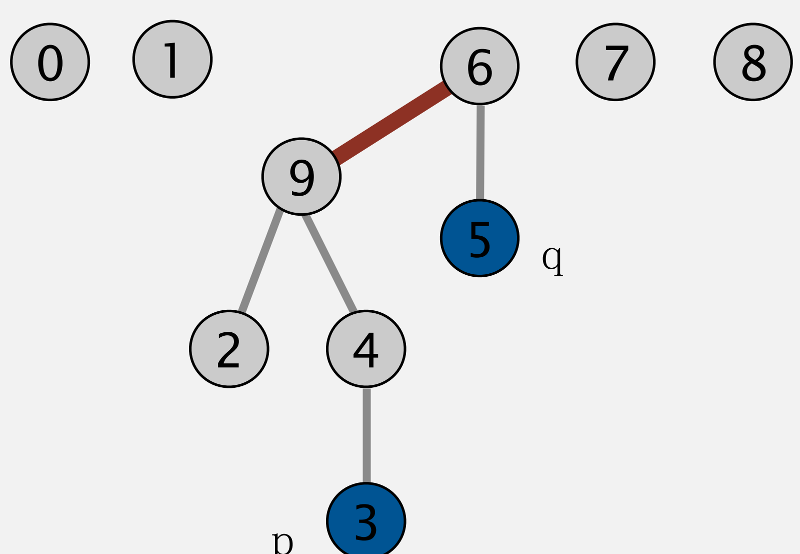
Find. What is the root of `p`?

Connected. Do `p` and `q` have the same root?

Union. To merge components containing `p` and `q`, set the id of `p`'s root to the id of `q`'s root.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	6

only one value changes



Key point. Weighted quick union (and/or path compression) makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	$O(m\, n)$
quick-union	$O(m\, n)$
weighted QU	$O(n\ +\ m \log n)$
QU + path compression	$O(n\ +\ m \log n)$
weighted QU + path compression	$O(n\ +\ m \log^* n)$

order of growth for m union-find operations on a set of n objects

Ex. [10^9 unions and finds with 10^9 objects]

- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

That’s all for today

- Thank you and good luck!