Data Structures and Algorithms

Session 22. April 15, 2009

Instructor: Bert Huang

http://www.cs.columbia.edu/~bert/courses/3137

Announcements

- Homework 4 solutions on Courseworks
- Homework 5 due Monday
- Homework 6 out Monday; final homework
- Out of town Wed to Sun; will be online

Review

- High Level Introduction to Complexity Classes
  - P, NP, NP-Complete, NP-hard, Undecidable
  - Famous NP-Complete problems
    - Satisfiability, Hamiltionian Path, Graph Coloring, Boolean Traveling Salesman

Today's Plan

- Disjoint Set ADT
- Definition
- Implementation
- Analysis

Motivating Example

- One interpretation of Kruskal's Algorithm:
  - Think of trees as sets of connected nodes
  - Merge sets by connecting nodes
  - Never merge nodes that are in the same set
- Simple idea, but how can we implement it?

Equivalence Relations

- An equivalence relation is a relation operator that observes three properties:
  - **Reflexive**: (a R a), for all a
  - **Symmetric**: (a R b) if and only if (b R a)
  - **Transitive**: (a R b) and (b R c) implies (a R c)
- Put another way, equivalence relations check if operands are in the same **equivalence class**

Equivalence Classes

- Equivalence class: the set of elements that are all related to each other via an equivalence relation
- Due to transitivity, each member can only be a member of one equivalence class
- Thus, equivalence classes are **disjoint sets**
  - Choose any distinct sets S and T, $S \cap T = \emptyset$

Disjoint Set ADT

- Collection of objects, each in an equivalence class
- **find**(x) returns the class of the object
- **union**(x,y) puts x and y in the same class
  - as well as every other relative of x and y
- Even less information than hash; no keys, no ordering

Implementation Observations

- One simple implementation would be to store the class label for each element in an array
  - O(1) lookup for **find**, O(N) for **union**
- If we store equivalent elements in linked lists, we avoid scanning the whole set during **union**
  - We can change the labels of the smaller class

Data Structure

- Store elements in equivalence (general) trees
- Use the tree's root as equivalence class label
- **find** returns root of containing tree
- **union** merges tree
- Since all operations only search up the tree, we can store in an array

Implementation

- Index all objects from 0 to N-1
- Store a parent array such that **s[i]** is the index of **i**'s parent
- If **i** is a root, store the negative size of its tree*
- **find** follows **s[i]** until negative, returns index
- **union**(x,y) points the root of x's tree to the root of y's tree

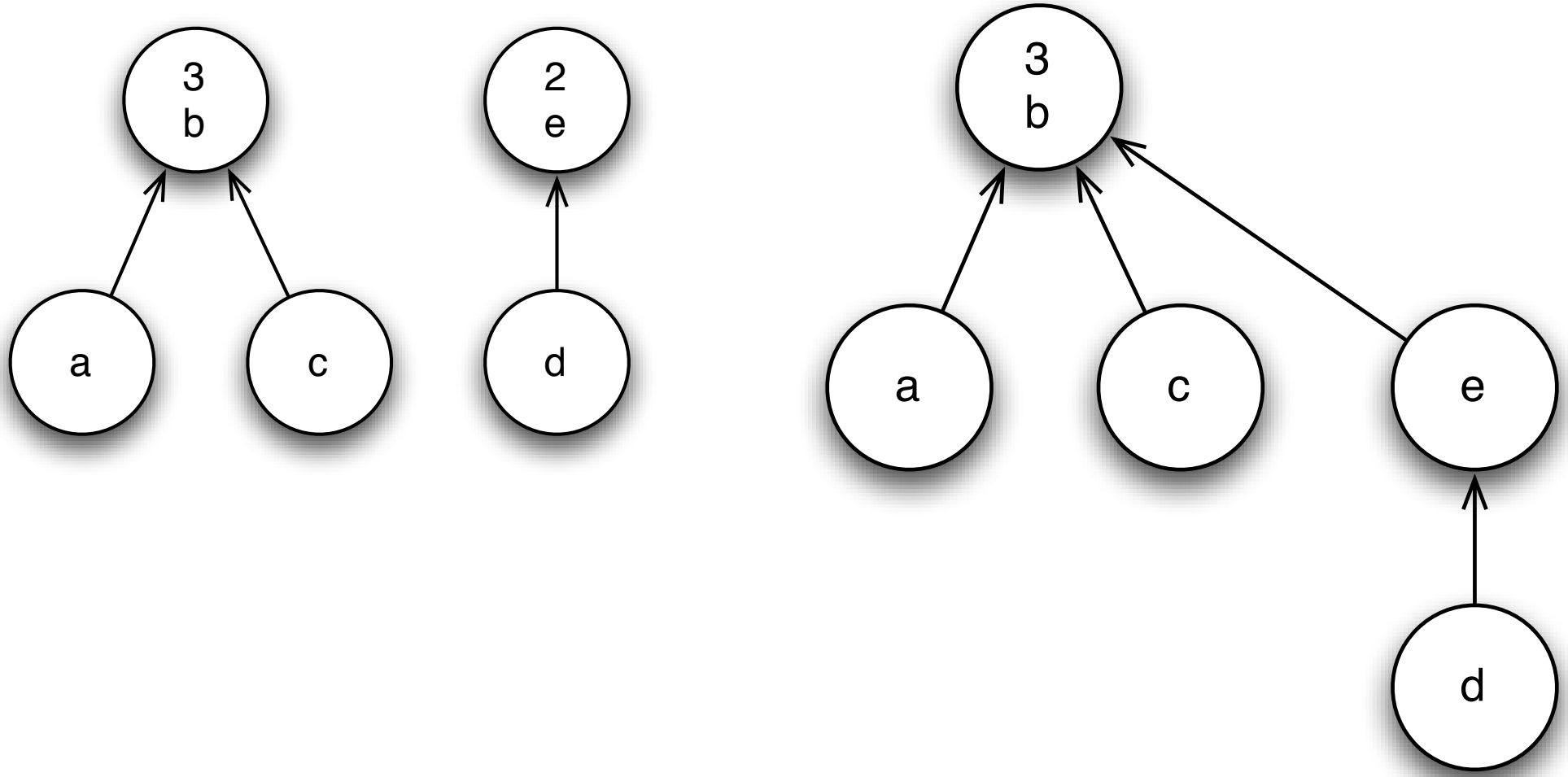
Analysis

- **find** costs the depth of the node
- **union** costs O(1) after **find**ing the roots
- Both operations depend on the height of the tree
- Since these are general trees, the trees can be arbitrarily shallow

Union by Size

- Claim: if we union by pointing the smaller tree to the larger tree's root, the height is at most log N
- Each union increases the depths of nodes in the smaller trees
- Also puts nodes from the smaller tree into a tree at least twice the size
  - We can only double the size log N times

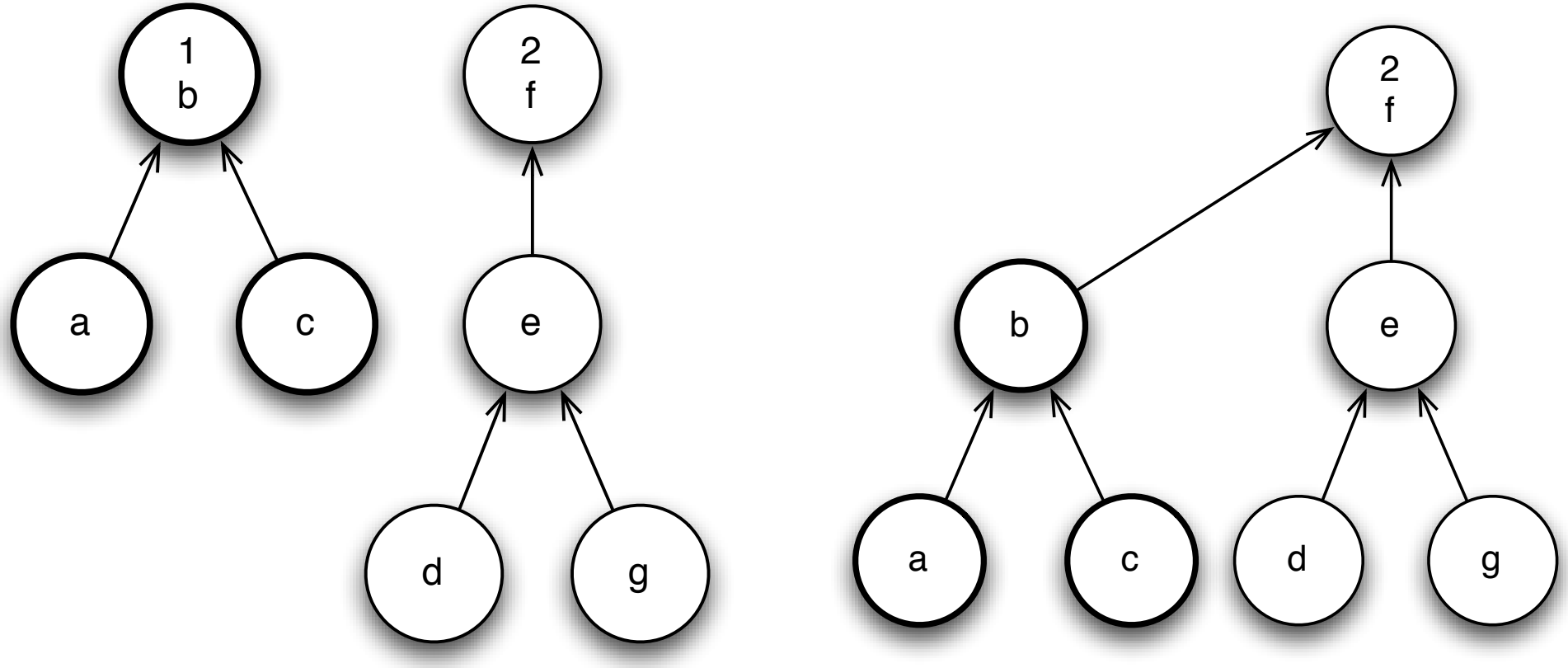
Union by Size Figure



Union by Height

- Similar method, attach the tree with less height to the taller tree
- Shorter tree's nodes join a tree at least twice the height, overall height only increases if trees are equal height

Union by Height Figure



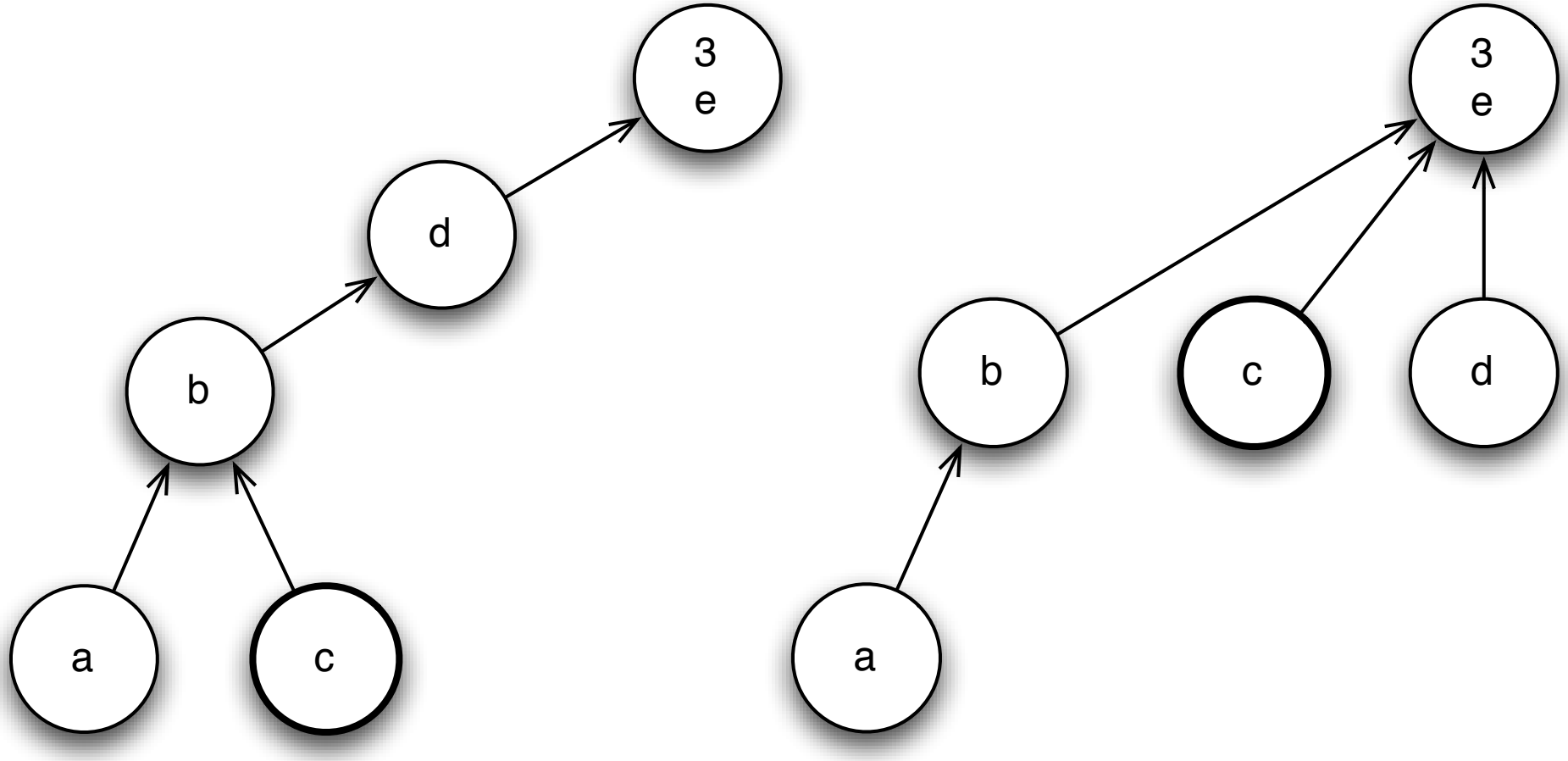
Union by Height proof

- Induction: tree of height **h** has at least 2^h nodes
- Let **T** be tree of height **h** with least nodes possible via union operations
- At last union, **T** must have had height **h-1**, because otherwise, it would have been a smaller tree of height **h**
- Since the height was updated, **T** unioned with another tree of height **h-1**, each had at least 2^{h-1} nodes resulting in at least 2^h nodes for **T**

Path Compression

- Even if we have log N tall trees, we can keep calling **find** on the deepest node repeatedly, costing O(M log N) for M operations
- Additionally, we will perform **path compression** during each **find** call
  - Point every node along the find path to root

Path Compression Figure



Union by Rank

- Path compression messes up union-by-height because we reduce the height when we compress
- We could fix the height, but this turns out to gain little, and costs **find** operations more
- Instead, rename to **union by rank**, where **rank** is just an overestimate of height
- Since heights change less often than sizes, rank/height is usually the cheaper choice

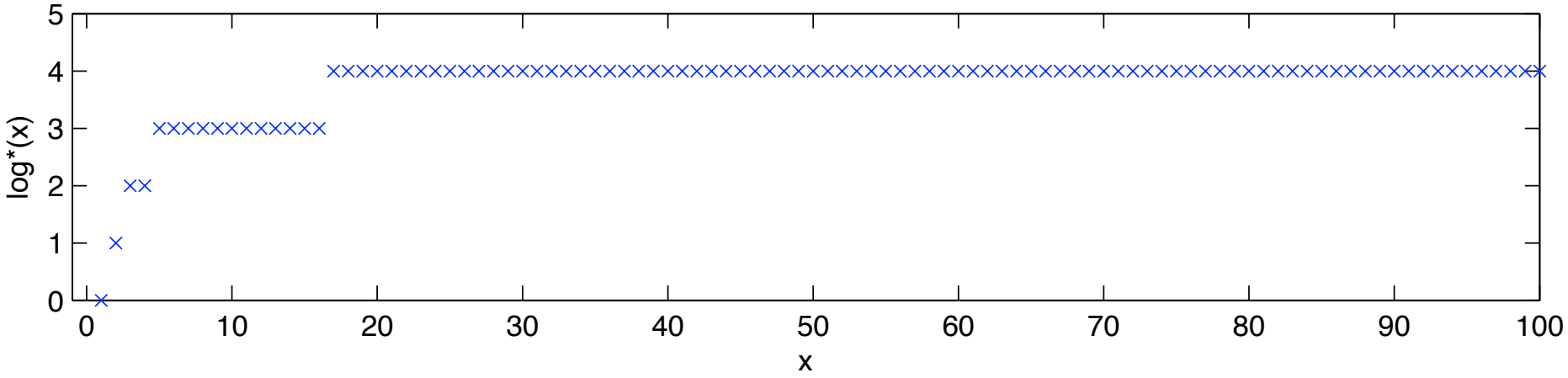
Worst Case Bound

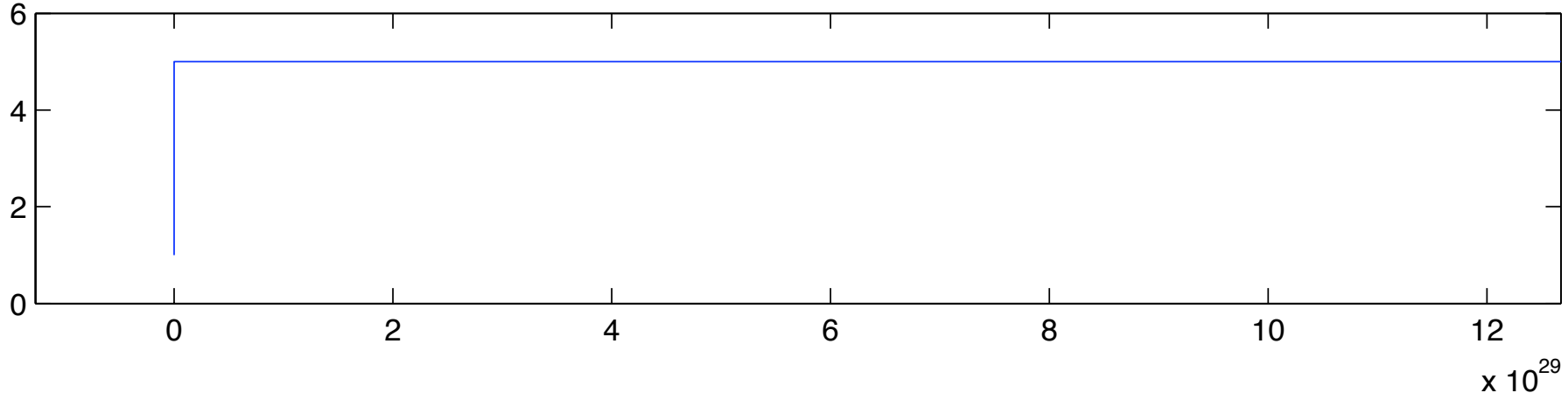
- The algorithms described have been proven to have worst case $\Theta(M\alpha(M,N))$ where α is the inverse of Ackermann's function:
  - $$\begin{aligned} A(1,j) &= 2^j \\ A(i,1) &= A(i-1,2) \\ A(i,j) &= A(i-1,A(i,j-1)) \end{aligned}$$
  - $\alpha(M,N) = \min\{i \geq 1 | A(i, \lfloor M/N \rfloor) > \log N\}$

Worst Case Bound

- A slightly looser, but easier to prove/understand bound is that any sequence of $M = \Omega(N)$ operations will cost **O(M log* N)** running time
- log* N is the number of times the logarithm needs to be applied to N until the result is ≤ 1
- e.g., log*(65536) = 4 because log(log(log(log(65536)))) = 1

Log* Plots


log*(x)
0
1
2
3
4
5
0
10
20
30
40
50
60
70
80
90
100
x


0
2
4
6
0
2
4
6
8
10
12
x 10^29

Log* Steps

log* N = 1	2^1	2
log* N = 2	2^2	4
log* N = 3	2^{2^2}	16
log* N = 4	$2^{2^{2^2}}$	65536
log* N = 5	$2^{2^{2^{2^2}}}$	>> googol

Note about Kruskal's

- With this bound, Kruskal's algorithm needs N-1 unions, so it should cost almost linear time to perform unions
- Unfortunately the algorithm is still dominated by heap deleteMin calls, so asymptotic running time is still O(E log V)

Proof Preliminaries

- Plan: upper bound the number of nodes per rank, partition ranks into groups
- Lemma 1: a node of rank **r** must have at least 2^r descendents
- Proof by induction, same as union-by-height proof
- Proof is unchanged because rank is exactly height-without-compression

Initial Lemmas

- Lemma 2: The number of nodes of rank **r** is at most $N/2^r$
- Proof. A node with rank **r** is the root of a subtree with at least 2^r nodes. Any other nodes with rank **r** must root other subtrees.
- Lemma 3: The ranks of nodes on a path from leaf to root increase monotonically

Rank Groups

- We will use some group function **G(r)**, which returns the group of rank **r**
- We refer to the inverse of this function as $F = G^{-1}$
  - i.e., for group **g**, **F(g)** is the maximum rank of group **g**.
  - $F(g) = \max\{r | G(r) = g\}$

Rank Groups
G(r) = log* r

	G(r)
r=1	1
r=2	2
r=[3,16]	3
r=[17,65536]	4

	F(g)
g=1	1
g=2	2
g=3	16
g=4	65536

Operation Accounting

- **union** operations cost O(1), so we won't even count them for this analysis
- **find** costs O(1) for each vertex along the path
- We "pay a penny" for each vertex, sometimes we pay an American penny and sometimes Canadian
- We will use groups to decide when to pay each

American vs. Canadian

- For vertex **v**, if **v** or the parent of **v** is the root, or if the parent of **v** is in a different rank group than **v**, pay one American penny to the bank
- Otherwise, deposit a Canadian penny into **v**
- In the end, we will count both totals for our bound
- Lemma 4: for a **find**(v), # pennies deposited = to the number of nodes along path from **v** to root

American Pennies

- Lemma 5: total deposits of American pennies are at most M(G(N)+2)
- Proof. Each find operation deposits two American pennies: one for the root and one for its child.
  - Also, one American penny is deposited for each change in group. Along any path, at most G(N) group changes can occur, so each find costs at most G(N)+2

Canadian Pennies I

- Lemma 6: The number of vertices **V(g)** in rank group **g** is at most $N/2^{F(g-1)}$
- Proof. Lemma 2 says at most $N/2^r$ nodes of rank **r**

$$
\begin{aligned}
V(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{N}{2^r} \leq \sum_{r=F(g-1)+1}^{\infty} \frac{N}{2^r} \\
&\leq N \sum_{r=F(g-1)+1}^{\infty} \frac{1}{2^r} \leq \frac{N}{2^{F(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \\
&\leq \frac{N}{2^{F(g-1)}}
\end{aligned}
$$

Canadian Pennies II

- Lemma 7: The maximum number of Canadian pennies deposited in all nodes in rank group **g** is at most $NF(g)/2^{F(g-1)}$
- Proof. Each vertex in the group can receive at most $F(g) - F(g-1) \leq F(g)$ Canadian pennies before its parent isn't in the rank group.
- Lemma 8: # Canadian pennies is at most

$$N \sum_{g=1}^{G(N)} F(g)/2^{F(g-1)}$$

Total Pennies

- Combing Lemmas 5 and 8, the cost of M operations is at most:

$$M(G(N)+2) + N \sum_{g=1}^{G(N)} F(g)/2^{F(g-1)}$$

- Choose, log* as G(r) function. The inverse F function is then $2^{F(i-1)}$, which nicely cancels out on the term on the right.

- Theorem: $M = \Omega(N)$ operations cost O(M log* N)

$$M(G(N)+2) + NG(N)$$

Reading

- Weiss Chapter 8