Data Structures and Algorithms

Session 23. April 20, 2009

Instructor: Bert Huang

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

Announcements

- Homework 6 up later today;
  Last take-home assignment
    - due before last class: May 4th
- Final Review May 4th
- Exam Wednesday May 13th 1:10-4:00 PM, 633
    - cumulative, closed-book/notes

Review

- Disjoint Set ADT
  - **find**(i): return the equivalence class of i'th object
  - **union**(i,j): make i's relatives equivalent to j's
  - stored in trees with parent pointers; implemented with array
  - **Union-by-rank** and **path compression**

Today's Plan

- Prove $O(M \log^* N)$ running time for M unions/finds
- Sorting lower bound
- Radix Sort

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

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=2	1
r=[3,4]	2
r=[5,16]	3
r=[17,65536]	4

	F(g)
g=1	2
g=2	4
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)

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

Mazes

- For HW6, we'll be using the Disjoint Set ADT to build random mazes
- The method starts with a grid graph, where vertical and horizontal neighbors share edges
- Then, essentially, you run Kruskal's algorithm randomly (random spanning tree)
- Refer to hw6 pdf and Weiss Section 8.7 for more

Data Structures

- At this point, we have covered all the data structures in the course curriculum
- We can reflect upon how stronger our toolbox is now that we know of these structures
- And we have a flavor of how to intelligently design our own data structures

Sorting

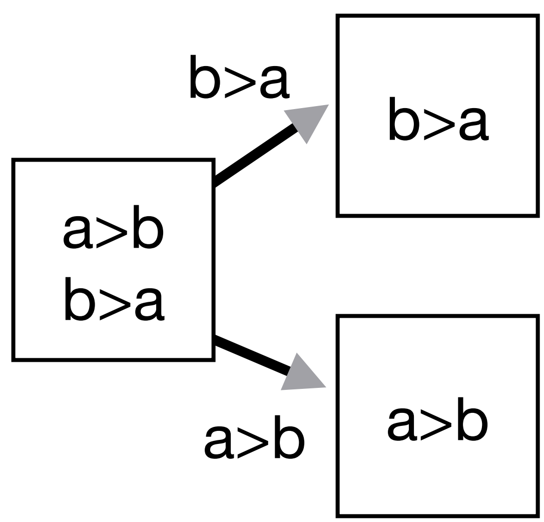
- Given array A of size N, reorder A so its elements are in order.
  - "In order" with respect to a consistent comparison function

The Bad News

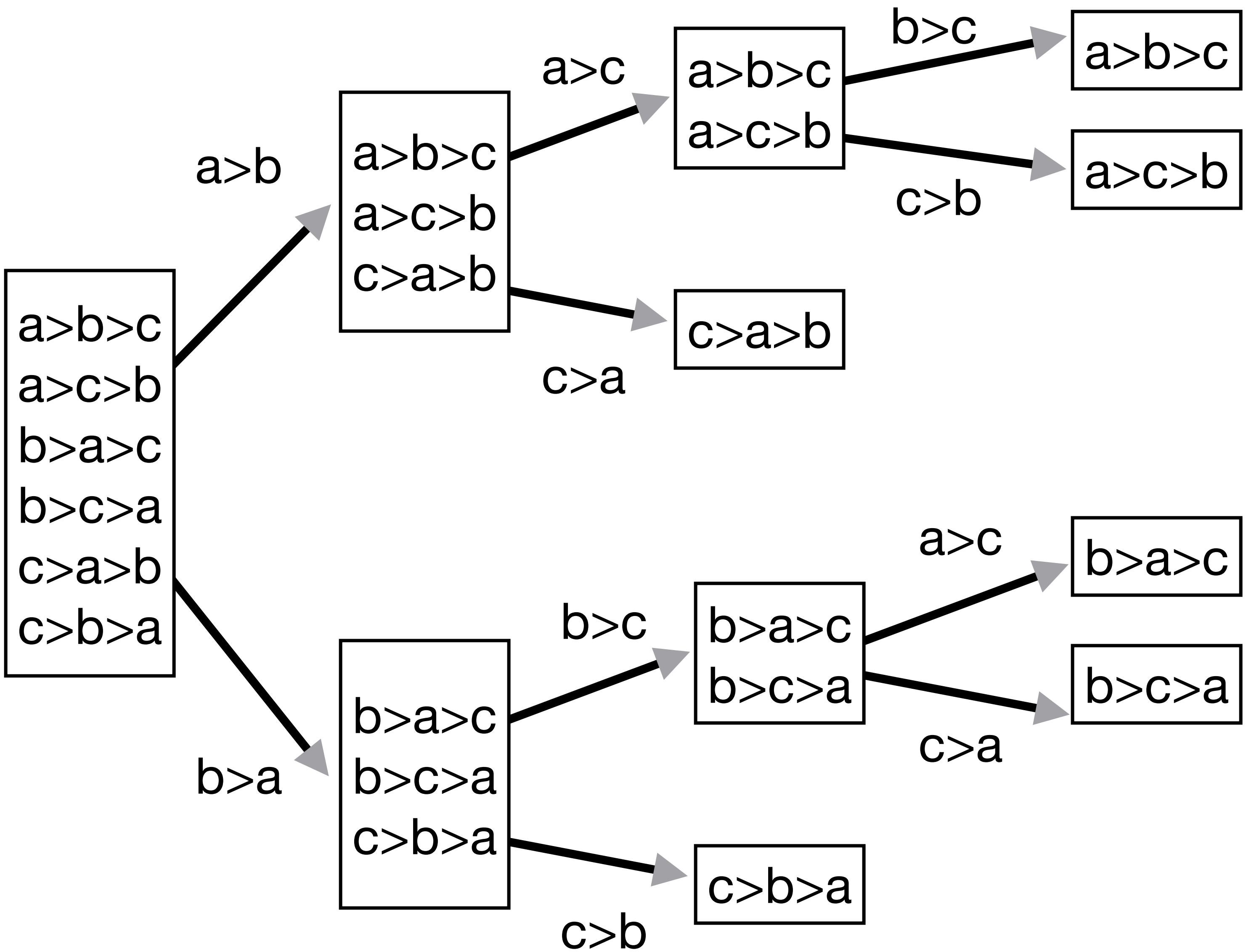
- Sorting algorithms typically compare two elements and branch according to the result of comparison
- Theorem: An algorithm that branches from the result of pairwise comparisons must use $\Omega(N \log N)$ operations to sort worst-case input
- Proof. Consider the decision tree

Comparison Sort Decision Tree: N=2

- Each node in this decision tree represents a state
- Move to child states after any branch
- Consider the possible orderings at each state



Decision Tree: N=3



Lower Bound Proof

- The worst case is the deepest leaf; the height
- Lemma 7.1: Let **T** be a binary tree of depth **d**. Then **T** has at most 2^d leaves
- Proof. By induction. Base case: d = 0, one leaf
    - Otherwise, we have root and left/right subtrees of depth at most **d-1**. Each has at most 2^{d-1} leaves

Lower Bound Proof

- Lemma 7.1: Let **T** be a binary tree of depth **d**. Then **T** has at most 2^d leaves
- Lemma 7.2: A binary tree with **L** leaves must have [height] at least $\lceil \log L \rceil$
- Theorem proof. There are N! leaves in the binary decision tree for sorting. Therefore, the deepest node is at depth $\log(N!)$

Lower Bound Proof

$$
\begin{aligned}
&\log(N!) \\
&\quad = \quad \log(N(N-1)(N-1)\ldots(2)(1)) \\
&\quad = \quad \log N + \log(N-1) + \log(N-2) + \ldots + \log 2 + \log 1 \\
&\quad \geq \quad \log N + \log(N-1) + \log(N-2) + \ldots + \log(N/2) \\
&\quad \geq \quad \frac{N}{2} \log \frac{N}{2} \\
&\quad \geq \quad \frac{N}{2} \log N - \frac{N}{2} \\
&\quad = \quad \Omega(N \log N)
\end{aligned}
$$

Comparison Sort Lower Bound

- Decision tree analysis provides nice mechanism for lower bound
- However, the bound only allows pairwise comparisons.
- We've already learned a data structure that beats the bound
  - What is it?

Trie Running Time

- Insert items into trie then preorder traversal
- Each insert costs **O(k)**, for length of word **k**
- **N** inserts cost **O(Nk)**
- Preorder traversal costs **O(Nk)**, because the worst case trie has each word as a leaf of a disjoint path of length **k**
  - This is a very degenerate case

Counting Sort

- Another simple sort for integer inputs
- 1. Treat integers as array indices (subtract min)
- 2. Insert items into array indices
- 3. Read array in order, skipping empty entries
- 4. Laugh at comparison sort algorithms

Bucket Sort

- Like Counting Sort, but less wasteful in space
- Split the input space into **k** buckets
- Put input items into appropriate buckets
- Sort the buckets using favorite sorting algorithm

Radix Sort

- Trie method and CountingSort are forms of Radix Sort
- Radix Sort sorts by looking at one digit at a time
- We can start with the least significant digit or the most significant digit
    - least significant digit first provides a **stable** sort
    - trie's use most significant, so let's look at least...

Radix Sort with Least Significant Digit

- BucketSort according to the least significant digit
- Repeat: BucketSort contents of each multi-item bucket according to the next least significant digit
- Running time: **O(Nk)** for maximum of **k** digits
- Space: **O(Nk)**

Reading

- http://www.sorting-algorithms.com/
- Weiss Chapter 7