Announcements

• Homework 5 due 11/24
Review

- Minimum Spanning Tree
- Prim’s algorithm: similar to Dijkstra
- Kruskal’s algorithm
- Disjoint Set ADT
Today’s Plan

• Review Disjoint Set ADT
• Start Discussion of Sorting
  • Lower bound
  • Breaking the lower bound
Analysis

- **find** costs the depth of the node
- **union** costs $O(1)$ after **finding** 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

<table>
<thead>
<tr>
<th>0=a</th>
<th>l=b</th>
<th>2=c</th>
<th>3=d</th>
<th>4=e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-3</td>
<td>1</td>
<td>4</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>-5</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Union by Height

- Similar method, attach the tree with less height to the taller tree
- Overall height only increases if trees are equal height
Union by Height Figure

<table>
<thead>
<tr>
<th></th>
<th>0=a</th>
<th>1=b</th>
<th>2=c</th>
<th>3=d</th>
<th>4=e</th>
<th>5=f</th>
<th>6=g</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>-2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

1 = b

1 = b

5
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

0=a  l=b  2=c  3=d  4=e

\begin{array}{|c|c|c|c|c|}
\hline
1 & 3 & 1 & 4 & -3 \\
\hline
\end{array}

\begin{array}{|c|c|c|c|c|}
\hline
1 & 4 & 4 & 4 & -3 \\
\hline
\end{array}
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

• 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 $\leq 1$

• So for all realistic intents, each operation is amortized constant time
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 $O(E \log V)$. 
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

```
 a>b
 b>a
```

```
 b>a
```

```
 a>b
```

```
 a>b
```
Decision Tree: N=3

Diagram showing decision tree with the comparison of elements a, b, and c. The tree is divided into branches based on the comparisons a > b, a > c, b > a, b > c, and c > a, leading to leaves indicating the order of the elements in different paths.
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

\[
\log(N!)
= \log(N(N - 1)(N - 1) \ldots (2)(1))
= \log N + \log(N - 1) + \log(N - 2) + \ldots + \log 2 + \log 1
\geq \log N + \log(N - 1) + \log(N - 2) + \ldots + \log(N/2)
\geq \frac{N}{2} \log \frac{N}{2}
\geq \frac{N}{2} \log N - \frac{N}{2}
= \Omega(N \log N)
\]
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

- TrieSort 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
  - tries 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: $O(Nk)$
Comparison Sorting

- Nevertheless, comparison-based sorting is much more general
- Well-studied problem, lots of different algorithms with various tradeoffs
- We’ll examine some of the famous algorithms
Reading

- Disj. Sets:
  Weiss Ch. 8 (skim proof in 8.6)

- Sorting:
  Weiss Section 7.8 (lower bound)
  Rest of Section 7 for next two classes