COMPSCI 330: Design and Analysis of Algorithms November 7, 2017

Lecture 18: Amortized Analysis of Disjoint Sets

Lecturer: Rong Ge *Scribe: Yumin Zhang*

1 Potential Argument

Problem. We are still talking about the amortized cost of $add(x)$ method in the dynamic array problem (for more details about the problem, please refer to the previous lecture's note). In addition to the aggregate method, accounting (charging) method, we have this third way to analyze the amortized cost, which is the **potential argument**.

Potential Argument. Define a "potential function" $\Phi, \Phi \geq 0$. Suppose an operation took (real) time T_i, changed the status from x_i to $x_(i+1)$. The amortized cost A of an operation is the following.

$$A_i = T_i - \Phi(x_i) + \Phi(x_{i+1})$$

The we claim the following.

$$\sum_{i=1}^{n} T_i = \sum_{i=1}^{n} A_i + \Phi(x_1) - \Phi(x_{n+1})$$

Proof for the claim.

$$\begin{aligned} A_i &= T_i - \Phi(x_i) + \Phi(x_{i+1}) \\ &= T_i - \underbrace{(\Phi(x_i) - \Phi(x_{i+1}))}_{\text{potential difference}} \end{aligned}$$

$$\begin{cases} A_1 = T_1 - \Phi(x_1) + \Phi(x_2) \\ A_2 = T_2 - \Phi(x_2) + \Phi(x_3) \\ \dots \\ A_n = T_n - \Phi(x_n) + \Phi(x_{n+1}) \end{cases}$$

$$\underbrace{\sum_{i=1}^{n} A_i}_{\text{total amortized}} = \sum_{i=1}^{n} A_i + \Phi(x_1) - \Phi(x_{n+1})$$

1.1 Design the Potential Function

Idea. First make sure $\Phi(x_i) - \Phi(x_{i+1})$ is large.
If $i = 2^k + 1$ is a heavy operation. ($T_i = 2^{k+1}$). Φ is a function of <u>number of elements</u> and <u>capacity</u>, we denote them as l and c

$$\begin{array}{lcc} & l & c \\ x_i: & 2^k & 2^k \\ & \uparrow & \\ \multicolumn{3}{c}{\text{before } 2^k+1 \text{ add operation, data structure has } 2^k \text{ elements}} \\ x_{i+1}: & 2^k+1 & 2^{k+1} \\ \text{difference } \Delta: & 1 & 2^k \end{array}$$

I want to pay for the cost of $T_i = 2^{k+1}$ using the difference in number of elements and number of capacity. Since the difference in capacity is much larger than the difference in number of elements, I want to use the difference in capacity. This imply the following idea: if capacity increase by 2^k, potential Φ decreases by 2^{k+1}. Then the potential function will look like

$$\Phi = \textcircled{?} - 2 \cdot c$$

step2. We need to make sure $\Phi \geq 0$. We first make the following observation:

$$l \cdot 2 \geq c \ (except for the initial state)$$
$$1 + 2 \cdot l \geq c \ (include the initial state)$$

So we can use $\Phi = 2 + 4l - 2c$ and we know $\Phi \geq 0$.

compute amortized running time.

① heavy operation $i = 2^k + 1$, $T_i = 2^{k+1}$

$$\Phi(x_i) = 2 + 4(2^k) - 2(2^k)$$
$$\Phi(xx+1) = 2 + 4(2^k+1) - 2(2^{k+1})$$
$$\Phi(x_i) - \Phi(xx+1) = 2^{k+1} - 4$$
$$A_i = T_i - (\Phi(x_i) - \Phi(x_{i+1})) = 2^{k+1} - (2^{k+1} - 4) = 4$$

② light operation $i \neq 2^k + 1$, $T_i = 1$

$$\begin{array}{lccc} & l & c & \Phi \\ x_i: & i-1 & \text{c} & 2+4(i-1)-2c \\ x_{i+1}: & \text{i} & \text{c} & 2+4i-2c \end{array}$$

$$\Phi(x_i) - \Phi(x_{i+1}) = -4$$

$$A_i = T_i - (\Phi(x_i) - \Phi(x_{i+1})) = 1 - (-4) = 5$$

$$amortized\ cost = 5 = O(1)$$

2 Union-Find

2.1 Data Structure for Disjoint Sets

Problem. There are n elements, each in a separate set. Want to build a data structure that supports two operations:

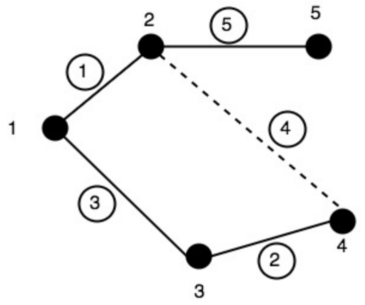
Find. Given an element, find the set it is in. (Each set is identified by a single head element in the set)

Merge. Given two elements, merge the sets that they are in.

2.2 Kruskal's Algorithm

For each edge we want to know whether adding the edge creates a cycle.

set $\iff$ connected component in graph



Initially all vertices are separated, the tree has no edges. Each vertex in separated connected component.

add an edge $\iff Find(u) = Find(v)$ (u,v in the same set)

edge(u,v) creates a cycle $\iff$ union on two connected components

$$\begin{aligned}
&edge\ \textcircled{1} : Find(1) = 1, Find(2) = 2, all\ sets = \{\{1\},\{2\},\{3\},\{4\},\{5\}\\
&\qquad Union(1,2), all\ sets = \{1,2\},\{3\},\{4\},\{5\}\\
&edge\ \textcircled{2} : Find(3) = 3, Find(4) = 4\\
&\qquad Union(3,4), all\ sets = \{1,2\},\{3,4\},\{5\}\\
&edge\ \textcircled{3} : Find(1) = 1, Find(3) = 3\\
&\qquad Union(1,3), all\ sets = \{1,2,3,4\},\{5\}\\
&edge\ \textcircled{4} : Find(2) = 3, Find(4) = 1\\
&\qquad do\ not\ add\ the\ edge\\
&edge\ \textcircled{5} : Find(2) = 1, Find(5) = 5\\
&\qquad Union(1,5), all\ sets = \{1,2,3,4,5\}
\end{aligned}$$

2.3 Implementation of Union Find

2.3.1 Naïve Implementation

Using Linked Lists.

- Merge: connects two linked lists, O(1)
- Find: needs to find the head/tail of the list, O(length)

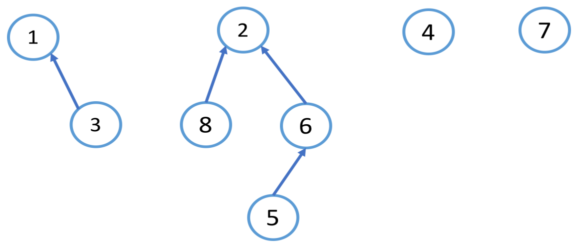
Using Array.

- a[i]: the set that item i is in
- Find: O(1)
- Merge(i,j): needs to update many entries in the array, O(length)

2.3.2 Representing Sets using Trees

- For each element, think of it as a node.
- Each subset is a tree, and the head element is the root.
- Find: find the root of the tree
- Merge: merge two trees into a single tree.

Example.



- Sets 1, 3, 2, 5, 6, 8, 4, 7
- Note: not necessarily binary trees.

Find Operation. Follow pointer to the parent until reach the root.

Merge Operation. Make the root of one set as a child for another set **Running Time.**

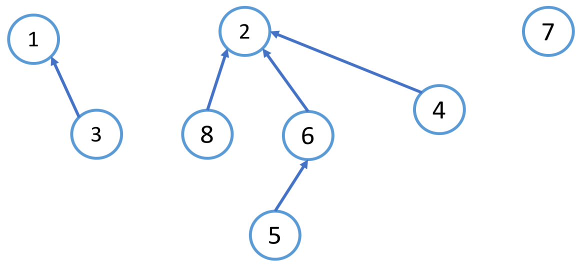
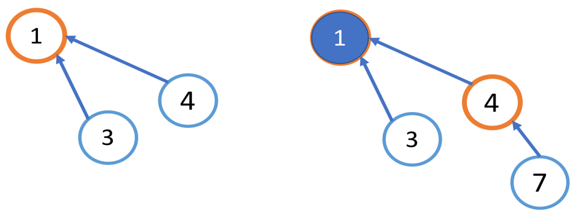


Figure 1: {4} merge with {2,8,6,2}

- Find: Depth of the tree.
- Merge: First need to do two find operation, then spend O(1) to link the two trees.
- In the worst-case, the tree is just a linked list
- Depth = n.

2.3.3 Union by Rank

- For each root, also store a rank
- When merging two sets, always use the set with higher rank as the new root.
- If two sets have the same rank, increase the rank of the new root after merging.



If we use union-by-rank, then the depth of the tree can be bounded by the following lemma:

Lemma 1. If the root of a tree has rank k, then the tree contains at least 2^k vertices.

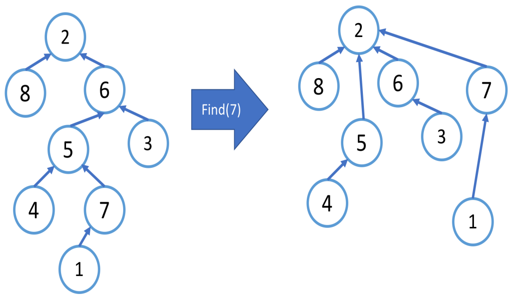
Proof. This can be proved by induction. Obviously this is true when $k = 0$.

Suppose this is true for all $k \leq t$. Now consider a tree whose root has rank $t+1$. The only way a node of rank $t+1$ can be created is if two trees of rank t are merged. By induction hypothesis, both trees have at least 2^t nodes, so the merged tree has at least 2^{t+1} nodes. □

The rank is also an upper-bound on the depth of the tree (when we are using only union-by-rank, rank is exactly equal to the depth). As an immediate corollary, the maximum rank/depth of a tree is bounded by $\log_2 n$.

Path Compression.

- After a find operation, connect everything along the way directly to the root.



Running Time.

- Union by rank only
  - Depth is always bounded by log n
  - O(log n) worst-case, O(log n) amortized

- Union by rank + path compression
  - Worst case is still O(log n)
  - Amortized: $O(\alpha(n)) = o(log^* n)$.

Here α is the inverse Ackermann function, which is a function that grows really slowly. The function $\log^*(n)$ is defined to be the number of $\log_2$ operations that it takes to make n a constant (say 1), which also grows very slowly. For all practical purposes, $\alpha(n) \leq 5$ and the data structure has essentially a constant amortized running time.