3 Design of Algorithm: By Data Structures

3.1 Data structures for disjoint sets

Reading: CLRS 21.1, 21.3, and 21.4

- Purposes: (1) Use good data structures to achieve algorithm efficiency; (2) Use amortization (it's the aggregate analysis) to give tight estimation of algorithm complexity.

- Problem: Initially, we are given n singletons: $S_i = \{i\}$ for $i = 1, \ldots, n$. We wish to execute a sequence of m operations of the following two types: (1) $union(S_i, S_j)$ returns $S_i \cup S_j$, where S_i and S_j are disjoint; (2) $find(i)$ returns the name of the set containing i. How can we organize the data (sets in this case) such that any sequence of intermixed *union*'s and $find$'s can be performed efficiently?

- Data structure: Set $\Rightarrow$ tree (arbitrary); Set name $\Rightarrow$ root; Sets $\Rightarrow$ forest (use parent array).

  Initially, there are n singletons, corresponding to n single-node trees in a forest, which is represented by a parent array of size n with 0 in each entry. ($parent[i]$ gives the parent of i in the forest. If $parent[i] = 0$, i is a root.)

  Implementation of *union* (by rank): Initially, each singleton node has a rank of 0. To union two sets, or trees, make the tree whose root has a smaller rank a subtree of the root of the other tree. If both roots before the union have the same rank, increment the rank of the root of the combined tree by one. $union(i, j)$ takes $O(1)$ time.

  Implementation of $find$ (with path compression): Two passes are needed between the node and its root. One is to find the root and the other is to do the path compression. $find(i)$ takes $O(d)$, where d is the depth of i in the tree, or the distance between i and the root.

  Note: *union* can also be done by size but it does not result in a different time complexity from union-by-rank.

- Complexity: Given an intermixed sequence σ of q *union*'s and p $find$'s, i.e., $m = q + p$. What is the worst-case time complexity for executing σ on an initial collection of n singletons $\{1\}, \ldots, \{n\}$?

  The worst-case analysis gives an overly pessimistic bound of $O(q + pn)$, or $O(mn)$. With an amortized analysis, Tarjan (1975) proved a worst-case time of $O(m \cdot \alpha(n))$, where $\alpha(n) = \min\{k | A_k(1) \geq n\}$ is the inverse Ackermann's function, which is defined as: $A_0(j) = j + 1$ and $A_k(j) = A_{k-1}^{(j+1)}(j)$ for $k \geq 1$. What we will show next is the slightly loose time of $O(m \cdot \log^* n)$, which is an earlier result given by Hopcroft and Ullman (1973). Both $\alpha(n)$ and $\log^* n$ are extremely slow-growing functions. We choose the $\log^* n$ version since its proof is a little bit easier.

- Properties about ranks:

  We observe that if there are only *union*'s but no $find$'s in σ, then the rank of a node is the height of the tree rooted at that node. However, if there are $find$'s in σ, then path compression may decrease the height of the root thus making its rank to be just an upper bound of the height. Additionally, the only time when the rank of a node may be changed (increased) is when it is the root of a tree. Once a node becomes a non-root, it will never be changed back to a root again, thus its rank will remain the same from that point. The following properties about ranks can be proved easily.

  P1: As the $find$ operation follows the path to the root, the ranks of nodes its encounters are strictly increasing.

  P2: Any node i with rank r has at least 2^r descendants (including itself).

  P3: There are at most $\frac{n}{2^r}$ nodes of rank r.

  Definitions: $F(0) = 1$ and $F(n) = 2^{F(n-1)}$ for $n \geq 1$. Let G be the inverse of F, i.e., $G(n) = \min\{k \geq 0 | F(k) \geq n\}$. Note that $G(F(n)) = n$ and $G(n) = \log^* n$.

n	0	1	2	3	4	5	$\cdots$	16	17
$F(n)$	1	2	4	16	65536	$2^{2^{2^2}}$ $\cdots$	$\cdots$	$\cdots$	$\cdots$
$G(n)$	0	0	1	2	2	3	$\cdots$	3	4

  We next distribute ranks into groups by putting rank r into group $G(r)$. Thus, group 0 contains ranks 0 and 1; group 1 contains just rank 2; group 2 contains ranks 3 and 4; group 3 contains ranks from 5 to 16; and finally group $G(n)$ contains the largest possible rank, n. See the table below.

Group	Ranks in the group
0	$0,1$
1	2
2	$3,4$
3	$5,\ldots,16$
$\cdots$	$\cdots$
$G(n)$	$\ldots,n$
g	$F(g-1)+1,\ldots,F(g)$

It is easy to see that in any sequence σ, the total cost of q *union*'s is $O(q)$. What is the total cost of p *find*'s? Observe that the total cost of *find*'s can be measured by the total number of hops via tree edges toward the root for all *find*'s. Let it be written as the sum of three types of cost, i.e., $T = T_1 + T_2 + T_3$.

First, T_1 is the total hops of those *find*'s that make only one hop to reach the root. Thus $T_1 = O(p)$.

Second, T_2 is the total hops between nodes with ranks in different groups. Recall that there are at most $G(n)+1$ different groups. For each *find*, the number of hops between nodes from different groups is at most the number of groups minus 1, i.e., $G(n)$. Thus $T_2 = O(p \cdot G(n))$.

Finally, T_3 is the total hops between nodes with ranks in the same group. Define S to be the total number of edges between nodes in the same groups during the entire course of executing σ. Imagine taking a snapshot after each operation and counts such edges in the snapshot. Then we have $T_3 \leq S$, since there may be edges not being hopped in executing the *find*'s.

Next, we try to estimate S. For any group g, there are $F(g) - F(g-1)$ ranks (from the table above). For any node u with a rank in group g, assume it is linked to a node (parent) in the same group, i.e., g. Every time a *find* is done, u is connected to a new parent with a higher rank (by path compression and by P1). This event will happen at most $F(g) - F(g-1)$ times, each time contributing one edge to S, until there is no higher ranks in group g, causing u to be connected to a parent in a different group. From then on, all edges containing u will have two nodes in different groups thus will not be considered. Therefore,

$$
\begin{aligned}
T_3 &\leq S \\
&\leq \sum_{\forall g} \sum_{\forall u \in g} (F(g) - F(g-1)) \\
&\leq \sum_{\forall g} \sum_{\forall u \in g} F(g) \\
&\leq \sum_{\forall g} (F(g) \cdot (\#\text{ nodes in group } g)) \\
&\leq \sum_{\forall g} (F(g) \cdot \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r}) \qquad \text{(By P3)} \\
&\leq \sum_{\forall g} (F(g) \cdot \frac{n}{F(g)}) \\
&\leq 2nG(n)
\end{aligned}
$$

The total cost for executing σ with $m = q + p$ operations is the sum of (1) $O(q)$ for q *union*'s; and (2) $O(p) + O(pG(n)) + O(nG(n)))$ for p *find*'s. So the total time is $O(q + pG(n) + nG(n))$, which is just a little more than $O(m+n)$ since $G(n) = \log^* n$ is an extremely slow-growing function, e.g. $G(2^{65534}) = 5$. Or $O(m \log^* n)$ if $m = \Theta(n)$, which is always the case in practice.

- Some classical papers:

1. Hopcroft, John E, and Ullman, Jeffrey D., Set Merging Algorithms, SIAM Journal on Computing 2(4), 1973.
2. Tarjan, Robert E., Efficiency of a Good But Not Linear Set Union Algorithm, Journal of ACM 22(2), 1975.
3. Tarjan, Robert E. and van Leeuwen, Jan, Worst-case Analysis of Set Union Algorithms, Journal of ACM 31(2), 1984.

3.2 Binary heaps

Reading: CLRS 6.1–6.5

- The (binary) heap is a data structure H of keys that supports $Insert(H,x)$, $ExtractMax(H)$, $FindMax(H)$, and $MakeHeap(H)$. It is represented as a left-complete binary tree with the heap property that the key of a parent is no smaller than the keys of its children (called max-heap). It can be implemented as an array. For example, $16,14,10,8,7,9,3,2,4,1$.
- A heap with n keys has height $\Theta(\log n)$ and the key of the root node is the maximum.
- Why use an array to implement a heap? In array $H[1\ldots n]$ that represents a heap, if the parent of $H[i]$ is $H[\lfloor i/2 \rfloor]$, the left child of $H[i]$ is $H[2i]$, and the right child of $H[i]$ is $H[2i+1]$.
- *Insert* and *ExtractMax* can be done in $O(\log n)$ time using the up-heap process and the down-heap process respectively. *FindMax* can be done in $O(1)$ time. *MakeHeap* can be interpreted in two ways. It may mean to make an empty heap, which can be done in $O(1)$ time. It may also mean to convert a binary left-complete tree (array) into a heap, which can be done using *Insert* n times or using down-heap $n/2$ times in $O(n\log n)$ time or $O(n)$ time respectively. We use $MakeHeap()$ and $MakeHeap(H)$ to distinguish the two alternatives.
- The heap-sort algorithm: $O(n\log n)$ worst-case time.

3.3 Fibonacci heaps

Raeding: CLRS 19.1–19.3

Here, we switch to the concept of min-heap to be consistent with the textbook.

- Mergeable heaps: A data structure that supports not only *Insert*, *ExtractMin*, *FindMin*, and *MakeHeap*, but also supports the following new operations: $Union(H_1,H_2)$ to make a new heap of elements in H_1 and H_2, $DecreaseKey(H,x,k)$ to decrease the key of x to a smaller value k, and $Delete(H,x)$ to delete x from H. Assume for *DecreaseKey* and *Delete*, the position of x in H is known.
- Mergeable heaps can be implemented by binary heaps with time $O(\log n)$, $O(\log n)$, $O(1)$, $O(1)$ or $O(n)$, $O(n)$, $O(\log n)$, $O(\log n)$ corresponding to the operations listed above. (Note: for *DecreaseKey*, apply up-heap at x and for *Delete*, replace the key of x with ∞ and apply down-heap.)
- A Fibonacci heap (Fredman and Tarjan 1986) is a collection of rooted trees (not necessarily binary) that are min-heap ordered. Pointers are used to connect the tree nodes in the following ways.
  - The roots are connected into a circular doubly linked list (in any order), with a pointer to the root with the minimum key
  - For any node, there is a pointer to its parent and a pointer to any one of its children.
  - All sibling nodes are connected into a circular doubly linked list
  - A few marked nodes (all unmarked initially, will only be marked in *DecreaseKey*, and may be unmarked again in *ExtractMin*)

  Notation: H (Fibonacci heap), n (number of nodes), $degree(x)$ (number of children of x), $degree(H)$ (maximum degree of any node in H), $trees(H)$ (number of trees in H), and $marks(H)$ (number of marked nodes in H).

  For the purpose of amortized analysis, define the potential of H to be $\Phi(H) = trees(H) + 2\cdot marks(H)$.
- Mergeable heap can be implemented by Fibonacci heaps.

  $MakeHeap()$: The actual time is $O(1)$, the increase in potential is zero, thus the amortized time is $O(1)$.

  FindMin: The actual time is $O(1)$, the increase in potential is zero, thus the amortized time is $O(1)$.

  Insert: Create a single-node tree (heap), inserted it to the root list, modify the pointer to the minimum if needed. The actual time is $O(1)$, the increase in potential is 1, thus the amortized time is $O(1)$.

  $Union(H_1,H_2)$: Concatenate the two root lists for $H1$ and H_2 into one root list for H and update the pointer to one of the two minimum keys. The actual time is $O(1)$, the increase in potential $\Phi(H)-(\Phi(H_1)+\Phi(H_2))$ is zero, thus the amortized time is $O(1)$.

ExtractMin: Delete the node with minimum key in the root list in $O(1)$, add its children into the root list in $O(degree(H))$, update the min pointer in $O(degree(H)+O(trees(H))$, and finally consolidate trees so that no two roots have the same degree using a pointer array indexed by degrees $0 \ldots degree(H)$ in $O(degree(H)+trees(H))$. So the actual time of *ExtractMin* is $O(degree(H)+trees(H))$.

See the example on CLRS pages 514 and 515.

Now consider the increase of potential caused by the operation. Let H' be the new Fibonacci heap and H be the old one before *ExtractMin*. Since the consolidation step may unmark some marked nodes, then $marks(H') \leq mark(H)$. Since no two trees have the same degree in H', then $trees(H') \leq degree(H)+1$. (Why?) Therefore, $\Phi(H')-\Phi(H) = (trees(H')-trees(H))+2\cdot(marks(H')-marks(H)) \leq trees(H')-trees(H) \leq degree(H)+1-trees(H)$.

So the amortized time for *ExtractMin* is $O(degree(H))$.

DecreaseKey: Self-study CLRS 19.3 to confirm that the amortized time for *DecreaseKey* is $O(1)$.

Delete: First call $DecreaseKey(H,x,-\infty)$ and then call $ExtractMin(H)$. So the amortized time for *Delete* is $O(1)+O(degree(H)) = O(degree(H))$.

- Bounding $degree(H)$: Wish to show $degree(H) = O(\log n)$, or more specifically, $degree(H) \leq \lfloor \log_\phi n \rfloor$, where $\phi = (1+\sqrt{5})/2 = 1.618\ldots$ is the golden ratio.

  Lemma 1: Let x be any node in the Fibonacci heap and assume that $degree(x) = k$. Let $y_1, y_2, \ldots, y_k$ be the children of x in the order in which they were linked to x, from the earliest to the latest. Then, $degree(y_1) \geq 0$ and $degree(y_i) \geq i-2$ for $i = 2,3,\ldots,k$.

  Lemma 2. For any integer $k \geq 0$, Fibonacci number follow $F_{k+2} = 1+\sum_{i=0}^{k} F_i$. (Proved by induction on k.)

  Lemma 3. For any integer $k \geq 0$, $F_{k+2} \geq \phi^k$. (Proved by induction on k.)

  Lemma 4. Let x be any node in a Fibonacci heap and let $k = degree(x)$. Let $size(x)$ be the number of nodes (including x) in the subtree rooted at x.Then $size(x) \geq F_{k+1} \geq \phi^k$.

  Theorem. $degree(H) = O(\log n)$.

  Proof. For any node x with degree k in the Fibonacci heap, by Lemma 4, we get $n \geq size(x) \geq \phi^k$. Thus $k \leq \log_\phi n$, or $k \leq \lfloor \log_\phi n \rfloor$ since k is an integer. So $degree(H) = \max_{\forall x}\{degree(x)\} \leq \lfloor \log_\phi n \rfloor$. So $degree(H) = O(\log n)$.