

# Chapter 21 Solusion

[github.com/frc123/CLRS](https://github.com/frc123/CLRS)

1/3/2022

## 21.1

### 21.1-1

Edge processed	Collection of disjoint sets										
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}	{k}
(d, i)	{a}	{b}	{c}	{d, i}	{e}	{f}	{g}	{h}		{j}	{k}
(f, k)	{a}	{b}	{c}	{d, i}	{e}	{f, k}	{g}	{h}		{j}	
(g, i)	{a}	{b}	{c}	{d, g, i}	{e}	{f, k}		{h}		{j}	
(b, g)	{a}	{b, d, g, i}	{c}		{e}	{f, k}		{h}		{j}	
(a, h)	{a, h}	{b, d, g, i}	{c}		{e}	{f, k}				{j}	
(i, j)	{a, h}	{b, d, g, i, j}	{c}		{e}	{f, k}					
(d, k)	{a, h}	{b, d, f, g, i, j, k}	{c}		{e}						
(b, j)	{a, h}	{b, d, f, g, i, j, k}	{c}		{e}						
(d, f)	{a, h}	{b, d, f, g, i, j, k}	{c}		{e}						
(g, j)	{a, h}	{b, d, f, g, i, j, k}	{c}		{e}						
(a, e)	{a, e, h}	{b, d, f, g, i, j, k}	{c}								

### 21.1-2

**Proof.** By contents in B.4, we know that the connected components of a graph are the equivalence classes of vertices under the “is reachable from” relation. The collection of the disjoint sets is exactly the quotient set of  $G.V$  by the “is reachable from” relation. It is not hard to find out that CONNECTED-COMPONENTS construct such the quotient set since the procedure unions vertices based on all edges, and edges connect two reachable vertices with the smallest length of the path (recall that a equivalence relation must be transitive). Two vertices are in the same connected component if and only if they are reachable from each other.  $\square$

### 21.1-3

FIND-SET:  $2 \cdot |E|$

UNION:  $|V| - k$

## 21.2

### 21.2-1

```
1  struct Set
2  {
3      Node *head;
4      Node *tail;
5      int size;
6  };
7
8  struct Node
9  {
10     int key;
11     Set *set;
12     Node *next;
13 };
14
15 void MakeSet(Node *x)
16 {
17     x->next = nullptr;
18     x->set = new Set; // need to be freed
19     x->set->head = x;
20     x->set->tail = x;
21     x->set->size = 1;
22 }
23
24 Node* FindSet(Node *x)
25 {
26     return x->set->head;
27 }
28
29 void Union(Node *x, Node *y)
30 {
31     Node *node;
32     if (x->set->size < y->set->size)
33     {
34         Union(y, x);
35     }
36     else
```

```
37     {
38         node = y->set->head;
39         x->set->size += y->set->size;
40         x->set->tail->next = node;
41         x->set->tail = y->set->tail;
42         delete y->set;
43         while (node)
44         {
45             node->set = x->set;
46             node = node->next;
47         }
48     }
49 }
```

## 21.2-2

collection before line 3:

$$\{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \{x_6\}, \{x_7\}, \{x_8\}, \{x_9\}, \{x_{10}\}, \{x_{11}\}, \{x_{12}\}, \{x_{13}\}, \{x_{14}\}, \{x_{15}\}, \{x_{16}\}\}$$

collection before line 5:

$$\{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5, x_6\}, \{x_7, x_8\}, \{x_9, x_{10}\}, \{x_{11}, x_{12}\}, \{x_{13}, x_{14}\}, \{x_{15}, x_{16}\}\}$$

collection before line 7:

$$\{\{x_1, x_2, x_3, x_4\}, \{x_5, x_6, x_7, x_8\}, \{x_9, x_{10}, x_{11}, x_{12}\}, \{x_{13}, x_{14}, x_{15}, x_{16}\}\}$$

collection before line 8:

$$\{\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}, \{x_9, x_{10}, x_{11}, x_{12}\}, \{x_{13}, x_{14}, x_{15}, x_{16}\}\}$$

collection before line 9:

$$\{\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}, \{x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}\}\}$$

collection before line 10:

$$\{\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}\}\}$$

Hence  $\text{FIND-SET}(x_2)$  and  $\text{FIND-SET}(x_{11})$  return a pointer points to  $x_1$ .

## 21.2-3

**Lemma 1.** Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of  $h$  UNION operations on a disjoint set that has never been operated UNION takes  $O(h \lg h)$  time.

**Proof.** We claim that, after  $h$  UNION operations, the largest set has at most  $h + 1$  members. Notice that the number of sets decreases by one each time UNION is called. Suppose that we have  $n$  sets in which each set contains one member in the beginning. After the  $h$  UNION operations, we have  $(n - h)$  sets. Note that each set must contain one member. In order to maximize the number of members in the the largest set, we let  $(n - h - 1)$  sets contains one member, and let the remaining set contains all remaining members. Then the remaining set contains

$$n - (n - h - 1) = h + 1$$

members.

We claim that each object's pointer back to its set object is updated at most  $\lceil \lg h \rceil$  times over all the UNION operations. Let  $x$  be an arbitrary object. By the similar approach in the proof of Theorem 21.1, we know that for any  $k \leq h + 1$ , after  $x$ 's pointers has been updated  $\lceil \lg k \rceil$  times, the resulting set must have at least  $k$  members. Since the largest set has at most  $h + 1$  members, each object's pointer is updated at most  $\lg(h + 1)$  times over all the UNION operations.

We claim that there are  $h$  elements have been updated their pointers back to their set objects at least once. Consider a set contains  $k$  members. Then within this set, there are  $(k - 1)$  members have been updated their pointers back to their set objects at least once since there must exists exactly  $(k - 1)$  members updated their pointers from the initial pointer to the current one. Let  $\mathcal{S}$  be our collection of sets. Then after the  $h$  UNION operations, the number of elements have been updated their pointers is

$$\sum_{A \in \mathcal{S}} (|A| - 1) = \sum_{A \in \mathcal{S}} |A| - |\mathcal{S}| = n - (n - h) = h$$

Since each object's pointer is updated at most  $\lceil \lg h \rceil$  times and there are  $h$  elements have been updated their pointers, we conclude  $h$  UNION operations on a disjoint set that has never been operated UNION takes  $O(h \lg h)$  time.  $\square$

**Claim 2.** The amortized time of MAKE-SET and FIND-SET is  $O(1)$ , and the amortized time of UNION is  $O(\lg n)$ .

**Proof.** Suppose that we performed  $h$  UNION operations. Since  $n$  MAKE-SET operations are performed, we know  $(m - n - h)$  FIND-SET operations are performed. By the lemma, we know that the total actual cost of UNION is  $O(h \lg h)$ . Hence the total actual cost of the sequence is

$$O(\underbrace{n}_{\text{MAKE-SET}} + \underbrace{(m - n - h)}_{\text{FIND-SET}} + \underbrace{h \lg h}_{\text{UNION}}) = O(m - h + h \lg h)$$

The total amortized cost of the sequence is

$$O(\underbrace{n}_{\text{MAKE-SET}} + \underbrace{(m - n - h)}_{\text{FIND-SET}} + \underbrace{h \lg n}_{\text{UNION}}) = O(m - h + h \lg n)$$

Since  $h < n$ , we have showed the claim successfully.  $\square$

## 21.2-4

In the  $i$ th UNION operation, we call  $\text{UNION}(x_{i+1}, x_i)$ . At this time, the size of set contains  $x_i$  contains  $i$  members, and the size of set contains  $x_{i+1}$  contains 1 members. Then we notice, for all  $i \geq 2$ , we append the list contains  $x_{i+1}$  onto the list contains  $x_i$  with the weighted-union heuristic, and this only takes  $\Theta(1)$  time for each operation. We operate  $n$  times MAKE-SET and  $(n - 1)$  times UNION, so the sequence takes  $\Theta(n + (n - 1)) = \Theta(n)$  time.

## 21.2-5

```
1  struct Node
2  {
3      int key;
4      Node *next;
5      // let the tail element be the set's representative
6      union
7      {
8          Node *tail; // for non-tail elements
9          Node *head; // for the tail element
10     } representative;
11     int size; // only for the tail element
12 };
13
14 void MakeSet(Node *x)
15 {
16     x->next = nullptr;
17     x->representative.head = x;
18     x->size = 1;
19 }
20
21 Node* FindSet(Node *x)
22 {
23     return x->next ? x->representative.tail : x;
24 }
25
26 void Union(Node *x, Node *y)
27 {
28     Node **node, *x_head, *y_head, *x_representative, *y_representative;
29     if (x->representative.tail->size < y->representative.tail->size)
30     {
31         Union(y, x);
```

```
32     }
33     else
34     {
35         x_representative = FindSet(x);
36         y_representative = FindSet(y);
37         x_head = x_representative->representative.head;
38         y_head = y_representative->representative.head;
39         x_representative->size += y_representative->size;
40         node = &y_head;
41         while (*node)
42         {
43             (*node)->representative.tail = x_representative;
44             node = &((*node)->next);
45         }
46         *node = x_head;
47         x_representative->representative.head = y_head;
48     }
49 }
```

## 21.2-6

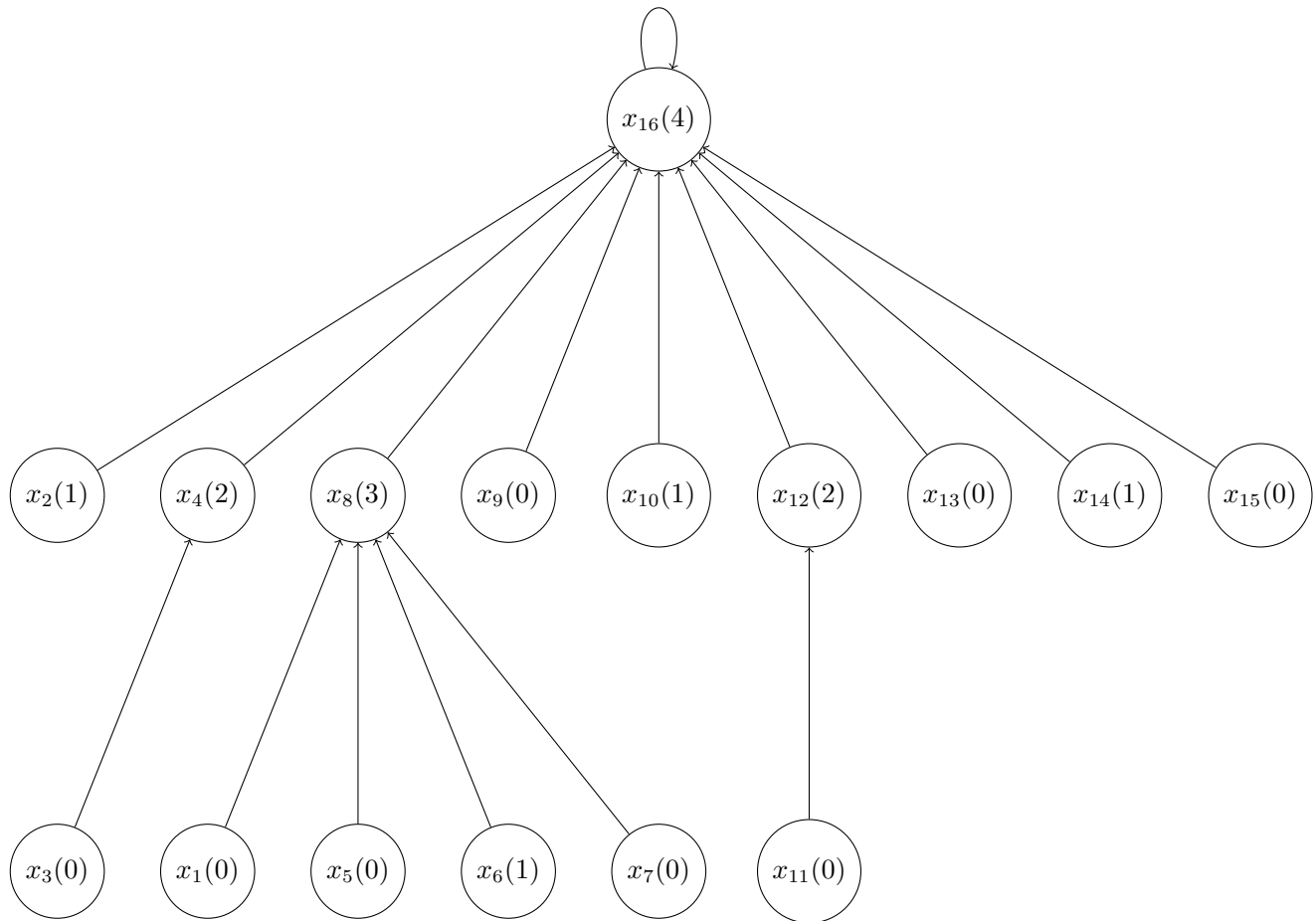
```
1  struct Set
2  {
3      Node *head;
4      int size;
5  };
6
7  struct Node
8  {
9      int key;
10     Set *set;
11     Node *next;
12 };
13
14 void MakeSet(Node *x)
15 {
16     x->next = nullptr;
17     x->set = new Set; // need to be freed
18     x->set->head = x;
19     x->set->size = 1;
```

```
20 }
21
22 Node* FindSet(Node *x)
23 {
24     return x->set->head;
25 }
26
27 void Union(Node *x, Node *y)
28 {
29     Node **node, *x_second;
30     if (x->set->size < y->set->size)
31     {
32         Union(y, x);
33     }
34     else
35     {
36         x->set->size += y->set->size;
37         x_second = x->next;
38         x->next = y->set->head;
39         node = &(x->next);
40         delete y->set;
41         while (*node)
42         {
43             (*node)->set = x->set;
44             node = &((*node)->next);
45         }
46         *node = x_second;
47     }
48 }
```

## 21.3

### 21.3-1

The data structure in the end (rank of each node is in the parentheses):



Hence  $\text{FIND-SET}(x_2)$  and  $\text{FIND-SET}(x_{11})$  return a pointer points to  $x_{16}$ .

### 21.3-2

```
1 Node* FindSet(Node *x)
2 {
3     Node *representative, *tmp;
4     representative = x;
5     while (representative != representative->p)
6     {
7         representative = representative->p;
8     }
9     while (x->p != representative)
10    {
11        tmp = x->p;
12        x->p = representative;
13        x = tmp;
14    }
```



```
15     return representative;
16 }
```

### 21.3-3

Note that we are proving that the upper bound  $O(m \lg n)$  is tight (least upper bound), instead of prove it is a tight bound  $\Theta(m \lg n)$ . In order to prove it, we just need to find an example that takes  $\Omega(m \lg n)$  time, which is what the question is asking for. We want our sequence to take as much as possible time. WLOG, assume that  $n = 2^k$  for some  $k \in \mathbb{N}$ . Consider the following sequence:

$$\begin{aligned} &\langle \text{MAKE-SET}(x_1), \text{MAKE-SET}(x_2), \dots, \text{MAKE-SET}(x_n), \\ &\quad \text{UNION}(x_1, x_2), \text{UNION}(x_3, x_4), \dots, \text{UNION}(x_{n-1}, x_n), \\ &\quad \text{UNION}(x_1, x_3), \text{UNION}(x_5, x_7), \dots, \text{UNION}(x_{n-3}, x_{n-1}), \\ &\quad \text{UNION}(x_1, x_5), \text{UNION}(x_9, x_{13}), \dots, \text{UNION}(x_{n-7}, x_{n-3}), \\ &\quad \vdots \\ &\quad \text{UNION}(x_1, x_{n/2+1}), \\ &\quad \text{FIND-SET}(x_1) \dots \quad (\text{until all } m \text{ operations are performed}) \rangle \end{aligned}$$

We performed  $n$  MAKE-SET,  $(n-1)$  UNION, and  $(m-2n+1)$  FIND-SET. We observed we performed  $n/2$  UNION( $x_i, x_{i+1}$ ),  $n/4$  UNION( $x_i, x_{i+2}$ ),  $n/8$  UNION( $x_i, x_{i+4}$ ),  $\dots$ . We conclude we performed  $n/2^j$  UNION( $x_i, x_{i+2^{j-1}}$ ) for all  $j = \{1, 2, \dots, \lg n\}$ . For each  $j$ , the height of each tree increases by 1. Hence after all  $(n-1)$  UNION operations, the height of the tree (for the only set) is  $\lg n$ . Note that  $x_1$  is the deepest element in the tree. Then, each of FIND-SET( $x_1$ ) operation takes  $\Theta(\lg n)$  time, and we perform  $(m-2n+1)$  times FIND-SET( $x_1$ ) operation. Hence all of FIND-SET( $x_1$ ) take  $(m-2n+1)\Theta(\lg n) = \Theta(m \lg n)$  time. We successfully find a sequence that takes  $\Omega(m \lg n)$  time.

### 21.3-4

We just need to modify LINK procedure to maintain the data structure.

```
1 void Link(Node *x, Node *y)
2 {
3     Node *y_next;
4     if (x->rank > y->rank)
5     {
6         y->p = x;
7     }
8     else
9     {
10        x->p = y;
11        if (x->rank == y->rank)
```

```
12         ++y->rank;
13     }
14     // maintain the circular list
15     y_next = y->next;
16     y->next = x->next;
17     x->next = y_next;
18 }
19
20 std::list<Node*> PrintSet(Node *x)
21 {
22     Node *node;
23     std::list<Node*> result;
24     result.push_back(x);
25     for (node = x->next; node != x; node = node->next)
26     {
27         result.push_back(node);
28     }
29     return result;
30 }
```

### 21.3-5

Let  $a_i$  be the number of nodes with depth greater than 0 (i.e. non-root) in the forest after the  $i$ th operation. Let  $b_i$  be the number of nodes with depth greater than 1 (i.e. non-root and non-child-of-root) in the forest after the  $i$ th operation. Suppose that we start to perform FIND-SET operations in the  $k$ th operation. Let the potential function be

$$\Phi(D_i) \begin{cases} a_i & \text{if } i < k, \\ b_i & \text{if } i \geq k \end{cases}$$

where  $D_i$  is the disjoint forest after the  $i$ th operation. Let  $\Phi(D_0) = 0$ . Observed each of MAKE-SET and LINK takes  $O(1)$  time. Denote  $\text{depth}_i(x)$  as the depth of the node  $x$  in the tree after the  $i$ th operation. Then FIND-SET( $x$ ) moves  $\max(0, \text{depth}_{i-1}(x) - 1)$  nodes to be the children of the root node. Denote  $c_i$  as the cost of the  $i$ th operation. Then we assume

$$c_i = \begin{cases} 1 & \text{if MAKE-SET is performed in the } i\text{th operation,} \\ 1 & \text{if LINK is performed in the } i\text{th operation,} \\ \max(1, \text{depth}_{i-1}(x)) & \text{if FIND-SET}(x) \text{ is performed in the } i\text{th operation} \end{cases}$$

**Case 1.** MAKE-SET is performed in the  $i$ th operation.

Then  $a_i = a_{i-1}$ , so

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 0 = 1$$

**Case 2.** LINK( $x, y$ ) is performed in the  $i$ th operation.

Note that  $x$  and  $y$  must be root nodes of different tree. This operation make either  $x$  or  $y$  be a non-root node. Then  $a_i = a_{i-1} + 1$ , so

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

**Case 3.** FIND-SET( $x$ ) is performed in the  $i$ th operation.

Note that  $b_i \leq a_i$  for all  $i$ . If  $i = k$ , then

$$\Phi(D_i) - \Phi(D_{i-1}) = b_i - a_{i-1} \leq b_i - b_{i-1}$$

If  $i > k$ , then

$$\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1}$$

Note that

$$b_i - b_{i-1} = -(\max(1, \text{depth}_{i-1}(x)) - 1) = 1 - \max(1, \text{depth}_{i-1}(x))$$

Hence

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq c_i + b_i - b_{i-1} = \max(1, \text{depth}_{i-1}(x)) + (1 - \max(1, \text{depth}_{i-1}(x))) = 1$$

We have shown that amortized of each operation is  $O(1)$  with the path-compression heuristic, no matter whether we use union by rank or not. Hence the sequence takes  $O(m)$  time with the path-compression heuristic, no matter whether we use union by rank or not.

## 21.4

### 21.4-1

**Proof.** We prove by induction on the number of operations.

(Base) Initially, there is no element in the disjoint sets, so it is trivial.

(Induction) Denote  $D_i$  be the data structure after  $i$ th operation. Denote  $p_i(x)$  be  $x.p$  after  $i$ th operation. Denote  $\text{rank}_i(x)$  be  $x.\text{rank}$  after  $i$ th operation. Suppose that, for all  $x \in D_{i-1}$ , ①  $\text{rank}_{i-1}(x) < \text{rank}_{i-1}(p_{i-1}(x))$  if  $x \neq p_{i-1}(x)$ , ②  $\text{rank}_{i-1}(x) = \text{rank}_{i-2}(x)$  if  $x \neq p_{i-1}(x)$ , and ③  $\text{rank}_{i-1}(p_{i-1}(x)) \geq \text{rank}_{i-2}(p_{i-2}(x))$ .

**Case 1.** MAKE-SET( $y$ ) is performed in the  $i$ th operation.

Then all structures of trees in  $D_{i-1}$  remain same in  $D_i$ , and ①②③ are vacuously true for  $y$ .

**Case 2.** FIND-SET( $y$ ) is performed in the  $i$ th operation.

Since no rank changes in FIND-SET, ② holds. Let  $z$  be the root of the tree contains  $y$ . Let  $A$  be the set contains all the nodes on the simple path from  $y$  to  $z$  except  $z$  in  $D_{i-1}$ . Let  $w \in A$  be arbitrary choice. We have  $w \neq p_{i-1}(w)$ . By ①, since  $p_i(w) = z$ , we have

$$\text{rank}_{i-1}(w) < \text{rank}_{i-1}(p_{i-1}(w)) \leq \text{rank}_{i-1}(z) = \text{rank}_{i-1}(p_i(w))$$

Since no rank changes in FIND-SET, we have  $\text{rank}_i(w) < \text{rank}_i(p_i(w))$  and  $\text{rank}_i(p_i(w)) \geq \text{rank}_{i-1}(p_{i-1}(w))$ . For all elements not in  $A$ , their  $p$  are not changed in the  $i$ th operation. Thus, we conclude ①③ holds.

**Case 3.** UNION( $y, z$ ) is performed in the  $i$ th operation. Let  $f$  be the root of the tree contains  $y$ , and let  $g$  be the root of the tree contains  $z$ . UNION( $y, z$ ) is acutally perfoming the following sequence

$$\langle \text{FIND-SET}(y), \text{FIND-SET}(z), \text{LINK}(f, g) \rangle$$

By case 2, we know IH holds after FIND-SET. Then we assume ①②③ holds for  $(i-1)$ th operation and we are performing LINK( $f, g$ ) in the  $i$ th operation. There are three subcases:  $\text{rank}_{i-1}(f) > \text{rank}_{i-1}(g)$ ,  $\text{rank}_{i-1}(f) < \text{rank}_{i-1}(g)$ , and  $\text{rank}_{i-1}(f) = \text{rank}_{i-1}(g)$ ; we notice the first two are similar, so we ignore the first subcase. First, we suppose that  $\text{rank}_{i-1}(f) < \text{rank}_{i-1}(g)$ . Notice that  $f$  is the only node that  $p$  attribute is changed in the  $i$ th operation. We have  $p_{i-1}(f) = f$  and  $p_i(f) = g$ . Then

$$\text{rank}_{i-1}(p_{i-1}(f)) = \text{rank}_{i-1}(f) < \text{rank}_{i-1}(g) = \text{rank}_{i-1}(p_i(f))$$

Since no rank changes in this subcase, we have  $\text{rank}_i(f) < \text{rank}_i(p_i(f))$  and  $\text{rank}_i(p_i(f)) \geq \text{rank}_{i-1}(p_{i-1}(f))$ . Thus, we conclude ①②③ holds. Now, we suppose that  $\text{rank}_{i-1}(f) = \text{rank}_{i-1}(g)$ . By the similar approach to the last subcase, we have

$$\text{rank}_{i-1}(p_{i-1}(f)) = \text{rank}_{i-1}(f) = \text{rank}_{i-1}(g) = \text{rank}_{i-1}(p_i(f))$$

By line 5 of the procedure, we have  $\text{rank}_i(g) = \text{rank}_{i-1}(g) + 1$ , and ranks of all nodes except  $g$  remain same in the  $i$ th operation. Then

$$\text{rank}_{i-1}(p_{i-1}(f)) = \text{rank}_i(f) = \text{rank}_{i-1}(f) = \text{rank}_{i-1}(g) = \text{rank}_i(g) - 1 < \text{rank}_i(g) = \text{rank}_i(p_i(f))$$

Hence ①②③ holds for  $f$ . Since  $p_{i-1}(g) = p_i(g) = g$ , we have

$$\text{rank}_{i-1}(p_{i-1}(g)) = \text{rank}_{i-1}(g) = \text{rank}_i(g) - 1 < \text{rank}_i(g) = \text{rank}_i(p_i(g))$$

Hence ①②③ holds for  $g$  (①② are vacuously true). For all nodes other than  $f$  and  $g$ , rank and  $p$  attributes remain same in  $i$ th operation. Thus, we conclude ①②③ holds.  $\square$

## 21.4-2

**Lemma 3.** If there exist a node has rank of  $k$ , then there exists at least  $2^k$  nodes in the forest.

**Proof.** (Base) If there exist a node has rank of 0, there is at least one node in the forest.

(Induction) Suppose that the lemma is true for  $k = h$  for some  $h \geq 0$ . Consider  $k = h + 1$ . If there exist a node has rank of  $h + 1$ , then, by line 4 of LINK, there must were at least two node have rank of  $h$  before. By the inductive hypothesis, there exists at least  $2 \cdot 2^h = 2^{h+1}$  nodes in the forest.  $\square$

**Claim 4.** In an  $n$  disjoint sets using union by rank and path compression heuristic, every node has rank at most  $\lfloor \lg n \rfloor$ .

**Proof.** Suppose that there exist a node has rank of  $\lfloor \lg n \rfloor + 1$ , for the purpose of contradiction. By the lemma, there exists at least  $2^{\lfloor \lg n \rfloor + 1}$  nodes in the forest.

$$2^{\lfloor \lg n \rfloor + 1} > 2^{\lg n} = n$$

Contradiction. □

### 21.4-3

We need  $\lceil \lg(k+1) \rceil$  bits to store value  $k$ . Hence  $\lceil \lg(\lfloor \lg n \rfloor + 1) \rceil$  bits are necessary to store  $x.rank$ .

### 21.4-4

**Proof.** Taking advantage of Lemma 21.7, we convert the sequence into a sequence of  $m$  MAKE-SET, LINK, and FIND-SET. Without path compression, rank of each node is exactly the height of the node. Observed MAKE-SET and LINK take  $O(1)$  time, and FIND-SET( $x$ ) takes  $O(x.rank)$  time. Hence the sequence run in  $O(m \lg n)$  time. □

### 21.4-5

No. Consider  $x.rank = 1$ ,  $x.p.rank = 7$ , and  $x.p.p.rank = 8$ . Clearly,  $x.rank > 0$  and  $x.p$  is not a root. Since

$$A_2(x.rank) = A_2(1) = 7$$

and

$$A_3(x.rank) = A_3(1) = 2047 \quad ,$$

we have

$$A_2(x.rank) \leq x.p.rank < A_3(x.rank) \quad ,$$

so

$$\text{level}(x) = 2 \quad .$$

Since

$$A_0(x.p.rank) = A_0(7) = 8$$

and

$$A_1(x.p.rank) = A_1(7) = 15 \quad ,$$

we have

$$A_0(x.p.rank) \leq x.p.p.rank < A_1(x.p.rank) \quad ,$$

so

$$\text{level}(x.p) = 0 \quad .$$

Therefore,

$$\text{level}(x) > \text{level}(x.p) \quad .$$

## 21.4-6

Since  $A_3(1) = 2047$ ,  $\alpha'(n) \leq 3$  implies

$$A_3 = 2047 \geq \lg(n+1) \quad .$$

Then

$$n \leq 2^{2047} - 1 = \frac{2^{2048}}{2} - 1 = \frac{(2^4)^{512}}{2} - 1 = 16^{511} - 1 \gg 10^{80} \quad .$$

Replace all  $\alpha(n)$  with  $\alpha'(n)$  in the argument. The only modification we need to make is at the bound (21.1). We claim that

$$0 \leq \text{level}(x) < \alpha'(n) \quad .$$

We need to modify the argument to prove  $\text{level}(x) < \alpha'(n)$ .

$$\begin{aligned} A_{\alpha'(n)}(x.\text{rank}) &\geq A_{\alpha'(n)}(1) && \text{(because } A_k(j) \text{ is strictly increasing)} \\ &\geq \lg(n+1) && \text{(by the definition of } \alpha'(n)) \\ &> \lfloor \lg(n) \rfloor \\ &\geq x.p.\text{rank} && \text{(by exercise 21.4-2)} \end{aligned}$$

## Chapter 21 Problems

### 21-1

(a)

[4, 3, 2, 6, 8, 1]

(b)

**Proof.** (*Base*) Let  $j \in \{1, 2, \dots, m\}$ , and let  $t \in K_j$ . Suppose that  $\text{extracted}[k] = t$  for some  $k \in \{1, 2, \dots, j-1\}$ , for the purpose of contradiction. Then  $\text{extracted}[k]$  is extracting the value from  $K_j$  where  $j > k$ . Contradiction.

(*Induction*) Assume the value is removed from the set after we extract it. Suppose that ①

$\forall j \in \{1, 2, \dots, m\}, \forall t \in K_j, \text{extracted}[k] = t$  for some  $k \in \{j, j+1, \dots, m\}$  or  $t$  will not be extracted ,

and suppose that ② any values in  $\{1, 2, \dots, i-1\}$  have been extracted correctly (hence these values are removed from  $\bigcup_{j \in \{1, 2, \dots, m\}} K_j$ ). Then at this time,  $i$  is the smallest value that is not in the *extracted* array. Now, we determine  $j$  such that  $i \in K_j$ . By the hypothesis ①, we know that  $\text{extracted}[k] = i$  for some  $k \in \{j, j+1, \dots, m\}$  or  $i$  will not be extracted. Since  $\text{extracted}[j]$  extracted value early than  $\text{extracted}[j+1], \text{extracted}[j+2], \dots, \text{extracted}[m]$ , we know  $\text{extracted}[j]$  extracted the smallest

possible value  $i$ , and this value is what OFF-LINE-MINIMUM will choose, so we have shown that  $i$  is extracted correctly. Hence ② holds. Let  $l$  be the smallest value greater than  $j$  for which set  $K_l$  exists (i.e.  $extracted[l]$  was empty). Let  $A = K_j$  for facilitating our analysis to  $K_j$  after destroying  $K_j$ . Now, we performed  $K_l = K_k \cup K_l$  and destroyed  $K_j$ , we claim the hypothesis ① holds. By the way we choosed  $l$ , we knew  $extract[j], extracted[j+1], \dots, extracted[l-1]$  were nonempty, so for all  $t \in A$ ,  $extracted[k] = t$  only if  $k \neq j, j+1, \dots, l-1$ . Then for all  $t \in A$ ,  $extracted[k] = t$  for some  $k \in \{j, j+1, \dots, m\}$  or  $t$  will not be extracted. Hence ① holds.  $\square$

(c)

```
1  struct Node
2  {
3      int p;
4      int rank;
5      // set info:
6      int subsequence_index_lower; // only root
7      int subsequence_index_upper; // only root
8      int prev; // only root
9      int next; // only root
10 };
11
12 int FindSet(std::vector<Node>& forest, int x)
13 {
14     if (forest[x].p != x)
15         forest[x].p = FindSet(forest, forest[x].p);
16     return forest[x].p;
17 }
18
19 // keep y's set info
20 void Link(std::vector<Node>& forest, int x, int y)
21 {
22     forest[forest[x].prev].next = forest[x].next;
23     forest[forest[x].next].prev = forest[x].prev;
24     if (forest[x].rank > forest[y].rank)
25     {
26         forest[y].p = x;
27         forest[x].subsequence_index_lower = forest[y].subsequence_index_lower;
28         forest[x].subsequence_index_upper = forest[y].subsequence_index_upper;
29         forest[x].prev = forest[y].prev;
30         forest[x].next = forest[y].next;
```

```

31         forest[forest[x].prev].next = x;
32         forest[forest[x].next].prev = x;
33     }
34     else
35     {
36         forest[x].p = y;
37         if (forest[x].rank == forest[y].rank)
38             ++forest[y].rank;
39     }
40 }
41
42 void Union(std::vector<Node>& forest, int x, int y)
43 {
44     return Link(forest, FindSet(forest, x), FindSet(forest, y));
45 }
46
47 // operations: E represents by -1
48 // n: domain size
49 std::vector<int> OffLineMinimum(const std::vector<int>& operations, int n)
50 {
51     int i, j, m, root, last_root;
52     std::vector<Node> forest(n + 1); // index start by 1
53     // init disjoint-set forest
54     for (i = 1; i <= n; ++i)
55     {
56         forest[i].p = i;
57         forest[i].rank = 0;
58     }
59     // init subsequence
60     m = 1;
61     root = 0;
62     last_root = 0;
63     for (i = 0; i < operations.size(); ++i)
64     {
65         if (operations[i] < 0)
66         {
67             // extract
68             forest[root].subsequence_index_upper = m;
69             ++m;
70             last_root = root;

```



```
71     }
72     else if (last_root == root)
73     {
74         // first insert in the subsequence
75         root = operations[i];
76         forest[root].subsequence_index_lower = m;
77         forest[root].prev = last_root;
78         forest[last_root].next = root;
79     }
80     else
81     {
82         // non-first insert in the subsequence
83         forest[operations[i]].p = root;
84         forest[root].rank = 1;
85     }
86 }
87 m = forest[last_root].subsequence_index_upper;
88 forest[last_root].next = 0; // for situation of that the last operation is extract
89 forest[0].subsequence_index_lower = m + 1;
90 // compute extracted array
91 std::vector<int> extracted(m);
92 for (i = 1; i <= n; ++i)
93 {
94     root = FindSet(forest, i);
95     j = forest[root].subsequence_index_lower;
96     if (j <= m)
97     {
98         extracted[j - 1] = i;
99         if (forest[root].subsequence_index_lower < forest[root].subsequence_index_upper)
100             ++forest[root].subsequence_index_lower;
101         else
102             Union(forest, root, forest[root].next);
103     }
104 }
105 return extracted;
106 }
```

In the worst-case, the running time is  $O(n\alpha(n))$ .

## 21-2

(a)

**Proof.** Suppose that we performed  $n$  times MAKE-TREE. To maximize the depth we performed  $n - 1$  times GRAFT to form a tree where the depth of leaf is  $n - 1$  (the tree become a linked list contains all  $n$  nodes). Then we performed  $m - 2n + 1$  FIND-DEPTH on the leaf. Each of MAKE-TREE and GRAFT takes  $\Theta(1)$  time, and each of FIND-DEPTH takes  $\Theta(n)$  time. Hence the sequence takes

$$\underbrace{n \cdot \Theta(1)}_{\text{MAKE-TREE}} + \underbrace{(n - 1) \cdot \Theta(1)}_{\text{GRAFT}} + \underbrace{(m - 2n + 1) \cdot \Theta(n)}_{\text{FIND-DEPTH}} = O(mn)$$

time in total. Consider  $n = \frac{m}{3}$ . In this case, the sequence takes  $\Theta(m^2)$  time. □

(b)

```
1 void MakeTree(Node *x)
2 {
3     x->p = x;
4     x->rank = 0;
5     x->d = 0;
6 }
```

(c)

```
1 // after the function return, v->p is the root of the tree in the disjoint sets
2 void Compression(Node *v)
3 {
4     if (v->p != v->p->p)
5     {
6         Compression(v->p);
7         v->d += v->p->d;
8         v->p = v->p->p;
9     }
10 }
11
12 // after the function return, v->p is the root of the tree in the disjoint sets
13 int FindDepth(Node *v)
14 {
15     Compression(v);
16     return (v == v->p) ? v->d : (v->d + v->p->d);
17 }
```

(d)

```
1 void Graft(Node *r, Node *v)
2 {
3     Node *r_set, *v_set;
4     Compression(r);
5     // r_set is the root node of the disjoint set contains r
6     r_set = r->p;
7     // add depths of all nodes in the tree contains node r by the depth of node v
8     // correctness: disjoint set contains r is
9     // exactly the set contains all elements in the tree contains r
10    r_set->d += (FindDepth(v) + 1);
11    // v_set is the root node of the disjoint set contains v
12    v_set = v->p;
13    if (r_set->rank > v_set->rank)
14    {
15        v_set->p = r_set;
16        // since r_set becomes parent of v_set in the disjoint set,
17        // we need to subtract pseudodistance of v_set by that of r_set
18        v_set->d -= r_set->d;
19    }
20    else
21    {
22        r_set->p = v_set;
23        // since v_set becomes parent of r_set in the disjoint set,
24        // we need to subtract pseudodistance of r_set by that of v_set
25        r_set->d -= v_set->d;
26        if (r_set->rank == v_set->rank)
27            ++v_set->rank;
28    }
29 }
```

(e)

$\Theta(m\alpha(n))$  where  $n$  is the number of nodes in the forest.

## 21-3

(a)

**Proof.** LCA is actually doing a postorder tree walk (the walk executes lines 8 - 10 for the root after doing so for the subtrees). Then each pair in  $P$  will be processed twice at line 8, 9. Note

that the procedure blanken the node after the node is visted. Let  $\{u, v\}$  be an arbitrary choice. WLOG, assume  $u$  is visited before  $v$ . At the time of  $u$  is being visiting,  $v$  has not been visited, so  $v.color \neq BLACK$ , and line 10 does not execute for the pair  $\{u, v\}$ , therefore. At the time of  $v$  is being visiting,  $v$  has already been visited, so  $v.color == BLACK$ , and line 10 executes for the pair  $\{u, v\}$ , therefore. Thus, line 10 executes exactly once for each pair  $\{u, v\} \in P$ .  $\square$

(b)

**Proof.** Denote  $height(u)$  as the height of the node  $u$  in  $T$ , and denote  $depth(u)$  as the depth of the node  $u$  in  $T$ .

(Base) Let  $u \in T$  where  $height(u) = 0$ . Then  $u$  do not have any descendant. Suppose that at the time of the call  $LCA(u)$ , the number of sets is  $depth(u)$ . Then at the time of  $LCA(u)$  return, the number of sets is  $depth(u) + 1$ , where the new set is from  $Make-Set(u)$  at line 1. Thus the inductive hypothesis (see below) holds for all  $u \in T$  where  $height(u) = 0$ .

(Induction) Inductive hypothesis: for all  $u \in T$  where  $height(u) \leq k$ , if at the time of the call  $LCA(u)$ , the number of sets is  $depth(u)$ , then at the time of the call  $LCA(v)$  where  $v$  is a descendant of  $u$ , the number of sets is  $depth(v)$ , and at the time of  $LCA(u)$  return, the number of sets is  $depth(u) + 1$ .

Let  $u \in T$  where  $height(u) = k + 1$ . Suppose that at the time of the call  $LCA(u)$ , the number of sets is  $depth(u)$ . Then at the time after line 2 of  $LCA(u)$  is executed, the number of sets is  $depth(u) + 1$ . We claim that after the **for** loop (line 3 - 6), the number of sets is  $depth(u) + 1$ . At line 4, we call  $LCA(v)$  where  $v$  is a child of  $u$ . Then  $height(v) < height(u)$ , so  $height(v) \leq k$ . And the number of sets is  $depth(u) + 1 = depth(v)$ . By the inductive hypothesis, at the time of  $LCA(v)$  return, the number of sets is  $depth(v) + 1 = (depth(u) + 1) + 1 = depth(u) + 2$ . And we have at the time of the call  $LCA(w)$  where  $w$  is a descendant of  $v$ , the number of sets is  $depth(w)$ . At line 5, we perform  $UNION(x, y)$ , so at this time, the number of sets is  $depth(u) + 2 - 1 = depth(u) + 1$ . Thus, after the **for** loop, the number of sets is  $depth(u) + 1$ , and the number of sets is same at the time of  $LCA(u)$  return. Hence the inductive hypothesis holds for all  $u \in T$  where  $height(u) \leq k + 1$ .

(Conclusion) By mathematical induction, since at the time of the call  $LCA(T.root)$ , the number of sets is  $depth(T.root) = 0$ , and all nodes in  $T$  is a descendant of  $T.root$ , we conclude at the time of the call  $LCA(u)$ , the number of sets is  $depth(u)$ .  $\square$

(c)

Denote  $T_u$  as the subtree rooted at  $u$ .

**Lemma 5.** Suppose that the program is currently running in  $LCA(a)$  (i.e. the top of the call stack is  $LCA(a)$ ) at the end of the line 6. Let  $B$  be the set contains all black color children of  $a$ . Let  $A = a \cup \bigcup_{b \in B} T_b$ . Then the disjoint set of node  $a$  is exactly set  $A$  and  $FIND-SET(a).ancestor = a$ .

**Proof.** (Base) Let  $a \in T$  where  $height(a) = 0$ . Then  $a$  does not have any children. Thus, at the end of line 6 (consider the start of line 7 here) of  $LCA(a)$ , the disjoint set of node  $a$  only contains

$a$ , and  $\text{FIND-SET}(a).ancestor = a$  by line 2.

(*Induction*) Suppose that the lemma is true for all  $a$  where  $\text{height}(a) \leq k$ . Let  $a \in T$  where  $\text{height}(a) = k + 1$ . Let  $B$  be the set contains all black color children of  $a$ . Then for all  $b \in B$ ,  $\text{height}(b) \leq k$ . Since  $b.color$  is black, the colors of all children of  $b$  are black. By the hypothesis, all elements in  $T_b$  are in a same disjoint set. At line 5 - 6, we union all elements in  $T_b$  with  $a$ . Thus, the disjoint set of node  $a$  is  $a \cup \bigcup_{b \in B} T_b$ . We have  $\text{FIND-SET}(a).ancestor = a$  by line 6 immediately.  $\square$

**Claim 6.** Suppose the program is currently running in  $\text{LCA}(u)$  (i.e. the top of the call stack is  $\text{LCA}(u)$ ). Then for all  $v \in T$  where  $u \neq v$ , if  $v.color$  is BLACK, then  $\text{FIND-SET}(v).ancestor$  is the least common ancestor of  $u$  and  $v$ .

Note that it is impossible to have  $v$  to be a proper ancestor of  $u$ .

**Proof.** Let  $u, v \in T$  where  $u \neq v$ , and let  $a$  be the least common ancestor of  $u$  and  $v$ .

Suppose that the program is currently running in  $\text{LCA}(a)$ ,  $u$  has not been visited, and  $v$  has been visited. Then  $u.color$  is WHITE, and  $v.color$  is BLACK. Let  $b \in T$  such that  $b$  is a child of  $a$  and  $v$  is a descendant of  $b$  (i.e.  $v \in T_b$ ). Since the program is currently running in  $\text{LCA}(a)$ ,  $b.color$  must be BLACK also. By the lemma, we have  $\text{FIND-SET}(v) = \text{FIND-SET}(a)$  and  $\text{FIND-SET}(v).ancestor = \text{FIND-SET}(a).ancestor = a$ .

Suppose that the program keeps running and is currently running in  $\text{LCA}(u)$ . When the top of the call stack is not  $\text{LCA}(u)$  and the call stack still contains  $\text{LCA}(u)$ , the disjoint set contains  $a$  must remains same. Thus,  $\text{FIND-SET}(v).ancestor$  and  $\text{FIND-SET}(a).ancestor$  still equal to  $a$ .  $\square$

(d)

operation	location	times
MAKE-SET	line 1	$ T $
FIND-SET	line 2	$ T $
UNION	line 5	$ T  - 1$
FIND-SET	line 6	$ T  - 1$
FIND-SET	line 10	$ P $ (by part (a))

Hence LCA takes  $O((|T| + |P|)\alpha(|T|))$ . Note that we can bound  $|P|$  from above:

$$|P| \leq \binom{|T|}{2} = \Theta(|T|^2)$$

Then we can say that LCA takes  $O(|T|^2\alpha(|T|))$  also.