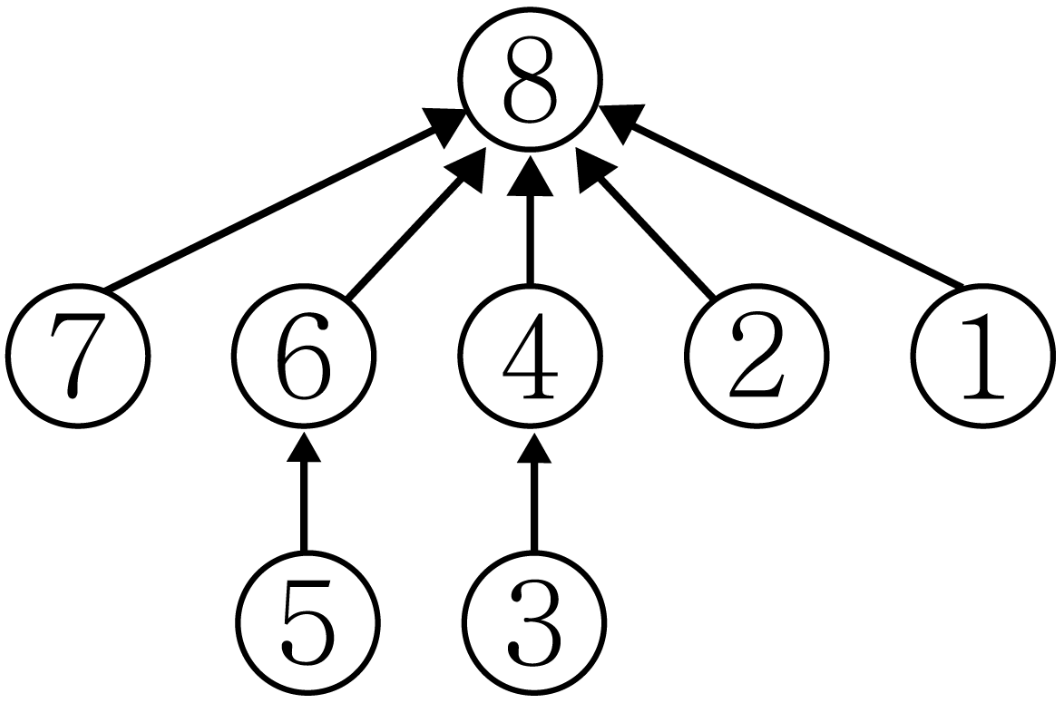
Introduction to Algorithms Prof. Helmut Alt
Spring Semester 2017 Team 31. 20140174 Hyungrok Kim
HW8

Exercise 1

(a)

state of the UNION-FIND data structure after the instructed sequence of operations.



arrow : child -> parent pointer
circle with number on it represent the node containing the number.
Node on the top, which is (8), is the root

(b)

we use UNION-by-height function and FIND with path-compression function.
For proving that the total runtime of the instructed sequence in exercise(sequence of first some UNION and some FIND operations in number m), is $\theta(m)$

First, remind that, height of a node is the number of edges of the longest path to leaf node.
to prove it we need 2 lemmas to use,

Lemma 1. Tree with height of h has at least 2^h nodes.

for induction

when h=0, tree has just 1 node, $1 \geq 2^0$, correct.

Assume that tree of height h has at least 2^h nodes.

think about tree of height (h+1), because of UNION-by-height,

it is only made when 2 trees with height of h UNION.

$number\ of\ nodes\ \geq\ 2^h + 2^h = 2^{h+1}$

By induction, Lemma 1 is correct.

Lemma 2. The number of nodes of height h is at most $n/2^h$

By Lemma 1, the number of nodes in tree of height h is at least 2^h,

Then, each node of height h will form a tree of height h including itself and subtrees, so for the minimum number of nodes of height h,

Each node of height h should be the root of the trees and the trees should have 2^h nodes each. Then, the number of nodes of height h is $n/2^h$

By that, Lemma 2 is correct.

Now divide the nodes by its height,

Group 0 contains nodes of height 0,

Group 1 contains nodes of height of [1, 2^1-1] = 1

Group 2 contains nodes of height of [2, 2^2-1] = [2, 3]

(which is height 2 and 3)

Group 3 contains nodes of height of [4, 2^4-1] = [4, 15]

(which is height 4, 5, ... ,14 ,15

And so on,

When Group n contains nodes of height of [x, 2^x-1]

Then Group (n+1) contains nodes of height of [2^x, 2^{2^x}-1]

For 1 increase of group number, the height lower bound becomes exponent of 2 of next group's lower bound (x -> 2^x)

The boundaries of height <u>increase extremely fast</u>, that even
When there are 10^{19000} nodes, we need 5 groups.
So, we can say that <u>the number of groups is very close to O(1).</u>
By Lemma 2, number of nodes of height h is at most $n/2^h$,
When a group contains nodes of height of [x, 2^x-1],
The group has at most $(n/2^x + n/2^{x+1} + n/2^{x+2} + \ldots + n/2^{2^x-1} + n/2^{2^x})$
$(n/2^x + n/2^{x+1} + n/2^{x+2} + \ldots + n/2^{2^x-1} + n/2^{2^x}) \le 2n/2^x$
because $(n/2^x + n/2^{x+1} + n/2^{x+2} + \ldots + n/2^{2^x-1} + n/2^{2^x})$ is a geometric sequence of ration of 1/2.
So <u>that group of height boundary [x, 2^x-1], has at most $2n/2^x$ nodes</u>
(n is the number of nodes)

The runtime of FIND Operations is proportional to the number of times a parent-pointer in the structure is accessed.
When m number of executions of FIND operation, we can categorize the pointer accession by,
1) links to the root
2) links from group to another group
3) links in same group

for 1),
because every FIND operation has one link to the root at the end, so the runtime is O(m).
for 2),
the previous group should have larger group number because FIND operation goes upward(height increasing) so the number of groups are the upper bound of accessions. And the number of groups is very close to O(1), so say it that the runtime is close to m*O(1)=O(m).
for 3),
when the group has height boundary of [x, 2^x-1],
the maximum number of nodes in the group is $2n/2^x$

And when link to a node from one to another, there can be maximum
$(2^x - 1 - x)$ links because height increases during the FIND operation.
by path-compression, every node we traverse changes its position to direct link to the root, so independent to m, the maximum pointer accession is
$(2^x - 1 - x) * 2n/2^x = O(n)$ and there are O(1) groups,
the runtime of total cases of 3) is O(n).
in m number of UNION-FIND operations, the number of nodes involved in operations $n \leq m$, so O(n) = O(m).
therefore, total runtime of m FIND operations = 1)+2)+3) = O(m)
so the amortized runtime per operation of FIND is O(1), and because O(1) means the constant time complexity, O(1) here is same with $\theta(1)$.
And we know that UNION operation has runtime of $\theta(1)$ per execution.

So when a sequence of UNION-FIND operations is operated,
the amortized runtime per operation is $\theta(1)$.

Exercise 2

(a)

in way of greedy algorithm,
from the beginning of the highway, the new station should be placed farthest possible position in 5 kilometers away from the nearest point with no available station.

L : total length of the long straight stretch of the highway
Array x : set $\{x_1, x_2, \dots, x_n\}$ of possible station positions (distance from the beginning of the highway)
P : nearest place where has no available station.
N : counting the number of stations
placed : Boolean value to check if there's place-able x_i
k : integer data to contain index of x to be made (i's value)

```
P = 0 at first (place with no available station, nearest from beginning)
N = 0;
with array x sort it in ascending order
i = 0;
k;
placed = false;
(0)while P<L
(1)     while x[i]<P+5
(2)             k = i
(3)             placed=true;
(4)             i++;
(5)     if placed is false,
(6)             return error;
(7)     P = x[k]+5;
(8)     N++;
(9)     placed=false;
return N;
```

execution of (1) is O(n) independent to outer while-loop(0),
rest (2)~(9) is constant for execution,
and (0) is loop for O(n) times so (0) ~ (9) is in O(n)
and above, sort for x is in O(nlogn), rest is in constant, so,
the time complexity = O(nlogn)

(b)
Think of a greedy algorithm which serves customers in ascending order of their required service time.
at first, everyone will be waited 0 minutes, when the first customer with t_i get in service, rest n-1 customers will wait t_i each and (n-1)*t_i at whole.
In that regulation,
the total waiting time will be

$(n-1) * t_{1st} + (n-2) * t_{2nd} +, \dots, +1 * t_{(n-1)th}$
(t_{i-th}: required service time for customer served i-th)
so in optimal, customers should be served in ascending order of t_i to make minimum, because the coefficient is in descending order.
Which is same with the greedy algorithm above.

n : number of customers waiting
array t : set $\{t_1, t_2, \dots, t_n\}$ which t_i is the needed service time for customer i.
array result : blank linear array of length n, which to fill with customer number i in order of the optimal process.

```
with array t, sort the array in ascending order, replace it in array t.
for (int i=0; i<n; i++)
        result[i] = k , from when t[i]= $t_k$
return result;
```

sorting for the array t is in O(nlogn),
for-loop for putting customer numbers in the returning array is in O(n)
the time complexity = O(nlogn)