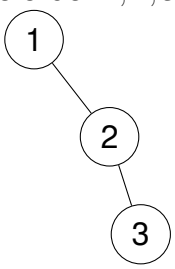
DSA Homework #5

5.1 Skip List, Binary Search Tree

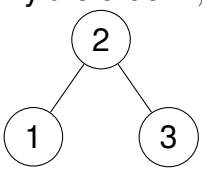
1. **Modified Skip List Structure:** When finding an elemnt in a general skip list of n levels, we have to traverse from the top level. During the travelsal, in each node, we can maintain and store the number of elements in between two neighbor nodes. That is, when instering or removing an element, as we are comparing the target element and node in the list of a level, we can always update that number affected by the operation (inserting or removing). And by this information stored in each node, the number of elements in between one node and its neighbor, we can find the median element in $\log n$.
   Find the Midian Point: We knows the number of elements n in a skip list. Thus, we can also calculate $\lfloor n \rfloor$. So when we traverse from the top level, by the information stored in each node described above, we can always know whether the median node is before or after a node (according to the accumulated amount of elements before a node). Namely, the number of nodes in the path is the number of the levles in the list, which is expected to be $\log n$, and so is the time complexity.

2. The binary search tree containing the set $\{1, 2, 3\}$ can varies because of the order the elements are instered.

   By the order $1, 2, 3$:

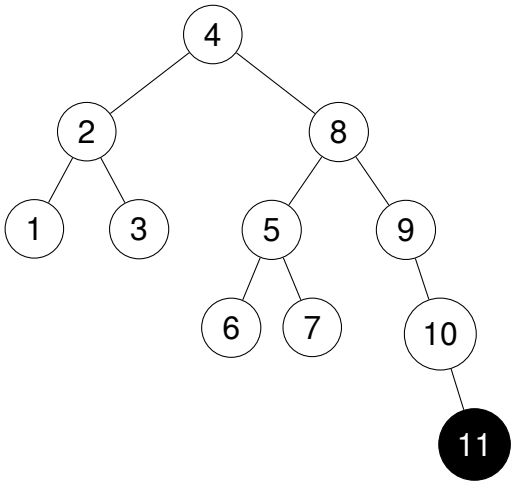
   

   By the order $2, 1, 3$:

3.

Before the new element with value 11 is inserted, the avl tree is balanced for any nodes. As the element with value 11 is inserted, the path from the newly inserted node to the root is $4, 8, 9, 10$. And the element with value $4, 9$ becomes unbalanced, while $8, 10$ remain balanced.
Therefore, that is an example of an avl tree whose nodes become unbalanced during an insert operation being non-consecutive.



5.2 Balanced Binary Search Trees

5.3 Disjoint Set

1. Assume there are two trees having n_1, n_2 elements with depth d_1, d_2, and the height of the new tree they merge into is d_3.
   We can get these properties:

   - If $|d_1 - d_2| > 1$, then d_3 will be $max(d_1, d_2)$.
     The number of elements in the new tree is $n_1 + n_2$.
   - If $|d_1 - d_2| \leq 1$, then d_3 will be $max(d_1, d_2) + 1$.
     And the number of elements in the new tree is $n_1 + n_2$ as well.
   - Therefore, we know that the depth of a tree grows only when their depth differ no more than one.

   Claim: The tree of depth n has at least $2^n - n$ nodes.

   - When $n = 1$:
     The claim is true, because the tree with only one node is of height 1.
   - Assume when $n = k$, the tree of depth k has at least $2^k - k$ nodes..
   - When $n = k + 1$: A tree can only grows form merging two trees many times. From the property above, a tree's depth can only grows from merging two trees whose depths $|d_1 - d_2| \leq 1$. Therefore, to produce a tree of depth $k + 1$, we can only merge two tree of depth either $\{k, k\}$ or $k, k + 1$. Taking the minimum case, so the tree of depth k has at least 2^{k+1} nodes.
     The claim is true.

   From the mathematical conduction, it proves that the tree of depth n has at least 2^n nodes.
   That implies that the depth of a tree with 2^n nodes is no more than n. Therefore, the depth of a tree is $O(\log n)$, and so is the time complexity of *find* and *union*.

2.

```
Array disjoint_sets[nFriends];
Array setOwner[nFriends];
while( not EOF ){
  if( Incident_Type == 1 ){

    tediour_game = GetInput();
    interesting_game = GetInput();

    tedious_game_set_root = set_find_root_ind( tediour_game );
    interesting_game_set_root = set_find_root_ind( interesting_game );

    tedious_game_set = disjoint_sets[tedious_game_set_root];
    interesting_game_set = disjoint_sets[interesting_game_set_root]

    disjoint_set::merge( tedious_game_set, interesting_game_set );
    setOwner[interesting_game_set_root] = setOwner[tedious_game_set_root];

  }else{

    friend_game = GetInput();

    friend_game_set_root = disjoint_set::find_root_ind( friend_game );

    Print( setOwner[friend_game_set_root] );

  }
}
```

3.

```
function( max_game_cnt( node, money ){
  if( node == NULL ){
    return 0;
  }
  left_cnt = 0;

  if( money < node -> left_subtree_sum ){
    return max_game_cnt( node -> left_subtree , money );
  }

  money -= node -> left_subtree_sum;
  left_cnt += node -> left_subtree_count;
  node_sum = ( node -> value ) * ( node -> count );
  //node -> count is the number of this node in the tree.
  //(The tree contains repeated value )

  if( money  >= node_sum ){
    money -= node_sum;
    left_cnt += node -> count;
  }else{
    return left_cnt + ( money / root -> value );
  }

  return max_game_cnt( root -> avl_link[1], money ) + left_cnt;
}
```

4. Define that merging two tree of n_1, n_2 nodes takes $min(n_1, n_2)\log_2(n_1+n_2)$ operations.
Claim: The number of operations needed to build a binary tree containing n nodes is at most:

$$\frac{n}{2}(\frac{\lceil(\log_2 n)\rceil\lceil(\log_2 n)-1\rceil}{2}+\log_2 n)$$

- When $n=1$:
Because when $k=1$, there is only a node, no merges are needed. It takes 0 operations.
- Assume when $n=h\leq k$, building the binary containing k nodes takes at most

$$\frac{h}{2}(\frac{\lceil(\log_2 h)\rceil\lceil(\log_2 h)-1\rceil}{2}+\log_2 h)$$

- When $n=k+1$: A tree with $k+1$ elements can be build from merging two tree of size n_1, n_2, where $n_1+n_2=h+1$ and $n_1\geq n_2$.
The number of operaion to merge the two trees is

$$n_2\lceil\log(n_1+n_2)\rceil=n_2\lceil\log(k+1)\rceil$$

And the time complexity of building the two trees with n_1, n_2 elements is

$$\frac{n_1}{2}(\frac{\lceil(\log_2 n_1)\rceil^2}{2}+\log_2 n_1)+\frac{n_2}{2}(\frac{\lceil(\log_2 n_2)\rceil^2}{2}+\log_2 n_2)$$

Therefore, the total number of operations is

$$\begin{aligned}
&n_2\lceil\log(k+1)\rceil\\
&+\frac{n_1}{2}(\frac{\lceil(\log_2 n_1)\rceil\lceil(\log_2 n_1)-1\rceil}{2}+\log_2 n_1)\\
&+\frac{n_2}{2}(\frac{\lceil(\log_2 n_2)\rceil\lceil(\log_2 n_2)-1\rceil}{2}+\log_2 n_2)\\
=&n_2\lceil\log(k+1)\rceil\\
&+\frac{(k+1-n_2)}{2}(\frac{\lceil(\log_2(k+1-n_2))\rceil\lceil(\log_2(k+1-n_2))-1\rceil}{2}+\log_2(k+1-n_2))\\
&+\frac{n_2}{2}(\frac{\lceil(\log_2 n_2)\rceil\lceil(\log_2 n_2)-1\rceil}{2}+\log_2 n_2)
\end{aligned}$$

Derivative it:

$$f'(x)=\frac{\log(16)\log(k+1)-(\log(k-n_2+1)-\log(n_2))(\log(k-n_2+1)+\log(2n_2)+2)}{4\log^2(2)}$$

Derivative it second time:

$$\begin{aligned}
&f''(x)\\
=&\frac{2x\log(k-x+1)+2(k-x+1)logx+(k+1)(2+\log 2)}{4x\log^2 2(k-x+1)}>0\\
&(\because n_2\geq 1\ and\ (k-n_2+1)\geq 1)
\end{aligned}$$

That implies that the first time derivitive function grows as n_2 grows up.
Thus, to find the minimun value of the first time derivitive function, make $n_2 = 1$.
We get:

$$f'(1) = \frac{\log(16)\log(k+1) - \log k(\log k + \log 2 + 2)}{4\log^2(2)}$$

Because $n_1 + n_2 = k + 1$ and $n_1 \geq n_2$, the maximum value of n_2 is $\lfloor \frac{k+1}{2} \rfloor$.
And we can also find the maximun value of the first derivative by make $n_2 = \lfloor \frac{k+1}{2} \rfloor$.

$$\begin{aligned}
& f'(n_2) \\
=& \frac{\log(16)\log(1+k) - (\log(1+k-n_2) - \log(n_2))(2 + \log(k-n_2+1) + \log(2n_2))}{4\log^2(2)} \\
& (\because \log(1+k-n_2) \approx \log(n_2).) \\
\approx& \frac{\log(16)\log(1+k) - 0 \times (2 + \log(k-n_2+1) + \log(2n_2))}{4\log^2(2)} \\
=& \frac{\log(16)\log(1+k)}{4\log^2(2)} > 0
\end{aligned}$$

Because the second derivative function is positive in $[1, \frac{1+k}{2}]$, and $f'(1) < 0 < f'(n_2)$, the maximum value of $f(n_2)$ must occurs when $n_2 = 1$ or when $n_2 = \frac{1+k}{2}$.

$$\begin{aligned}
& f(1) \\
=& \lceil \log(k+1) \rceil + \frac{k}{2}\left(\frac{\lceil \log_2 k \rceil \lceil \log_2 k - 1 \rceil}{2} + \lceil \log_2 k \rceil\right) + \frac{1}{2}\left(\frac{\lceil \log_2 1 \rceil \lceil \log_2 1 - 1 \rceil}{2} + \log_2 1\right) \\
=& \lceil \log(k+1) \rceil + \frac{k}{2}\left(\frac{\lceil \log_2 k \rceil \lceil \log_2 k - 1 \rceil}{2} + \lceil \log_2 k \rceil\right)
\end{aligned}$$

$$
\begin{aligned}
& f(\frac{k+1}{2}) \\
\approx & \frac{k+1}{2}\lceil\log(k+1)\rceil + \frac{k+1}{2}(\frac{\lceil\log_2\frac{k+1}{2}\rceil\lceil\log_2\frac{k+1}{2}-1\rceil}{2} + \log_2\frac{k+1}{2}) \\
= & \frac{k+1}{2}[\frac{\lceil(\log_2(k+1)-1)\rceil\lceil(\log_2(k+1)-2)\rceil + 2\log(k+1)}{2} + \lceil\log_2(k+1)\rceil - 1] \\
= & \frac{k+1}{2}[\frac{\lceil\log_2(k+1)\rceil\lceil(\log_2(k+1)-1)\rceil}{2} + \lceil\log_2(k+1)\rceil] \\
= & \frac{k}{2}[\frac{\lceil\log_2(k+1)\rceil\lceil(\log_2(k+1)-1)\rceil}{2} + \lceil\log_2(k+1)\rceil] \\
& + \frac{1}{2}[\frac{\lceil\log_2(k+1)\rceil\lceil(\log_2(k+1)-1)\rceil}{2} + \lceil\log_2(k+1)\rceil] \\
\geq & \frac{k}{2}[\frac{\lceil\log_2(k+1)\rceil\lceil(\log_2(k+1)-1)\rceil}{2} + \lceil\log_2(k+1)\rceil] \\
& + \frac{1}{2}[\frac{\lceil\log_2(k+1)\rceil\times 2}{2} + \lceil\log_2(k+1)\rceil] \ (as\ \lceil\log_2(k+1)-1\rceil \geq 2) \\
\geq & \lceil\log(k+1)\rceil + \frac{k}{2}(\frac{\lceil\log_2 k\rceil\lceil\log_2 k-1\rceil}{2} + \lceil\log_2 k\rceil) \\
= & f(1)
\end{aligned}
$$

Therefore when $\lceil\log_2(k+1)-1\rceil \geq 2$, $f(\frac{k+1}{2}) \geq f(1)$.
And for $0 \leq \lceil\log_2(k+1)-1\rceil < 2$:
When $\lceil\log_2(k+1)-1\rceil = 0$

$$f(1) = f(\frac{k+1}{2}) = 1$$

When $\lceil\log_2(k+1)-1\rceil = 1$

$$
\begin{aligned}
& f(1) = 2 + \frac{3}{2}k \\
& f(n_2) = \frac{3k+3}{2} \\
& \lfloor f(1)\rfloor = \lfloor f(n_2)\rfloor
\end{aligned}
$$

Therefore, the maximun value of $f(n_2)$ occur when $n_2 = \lfloor\frac{k+1}{2}\rfloor$.
Also,

$$
\begin{aligned}
& f(\frac{k+1}{2}) \\
= & \frac{k+1}{2}[\frac{\lceil(\log_2(k+1)-1)\rceil\lceil(\log_2(k+1)-2)\rceil + 2\log(k+1)}{2} + \log_2(k+1) - 1] \\
= & \frac{k+1}{2}[\frac{\lceil\log_2(k+1)\rceil\lceil(\log_2(k+1)-1)\rceil}{2} + \log_2(k+1)]
\end{aligned}
$$

Thus, the claim is true when $n = k+1$.

From mathematical induction, it proves that the claim is true. Therefore the total worst time complexity of making a tree with n nodes by merging trees is

$$\frac{n}{2}(\frac{\lceil(\log_2 n)\rceil\lceil(\log_2 n)-1\rceil}{2}+\log_2 n)$$

Obviously, it equals to $O(n(\log n)^2)$.