CSC265 Homework 9 | Fall 2021
Name: Haojun Qiu | Student Number: 1006900622
Who I discussed with: Darcy Wang, Runshi Yang, Leo Sun

Q1

There are two main parts of my data structure.

1. How are sets in restricted disjoint sets ADT with different size being represented in my data structure?

I use data structure S, and bunch of linked lists (for which every node wraps some (2 indeed, the reason why will be explained in the LINK operation) sets in S).

- Small sets, denoted s_S (any set with $\leq k$ elements) is stored directly as a set inside data structure S, representative of s_S is just the representative of this set in S.
- Large sets, denoted s_L (any set with $> k$ elements) is being represented by a **circular doubly linked list** of some length.
  - Each node in the list stores (wraps) some (2) sets in S, and has a pointer *representative* points to the representative of s_L.
  - Every linked list has a special node, called it **head node**, that wraps the set in S containing representative of s_L, and simultaneously, it has attribute *listlength* as the length of this linked list.

  For example, a linked list of length 1 (just 1 node) represents a set of size $k+1$ to $2k$. The only node wraps (or points) to two sets s_1, s_2 in S, and has a pointer *representative* points to the representative of set s_L, which is any of the representative of s_1 or s_2 in S. And this node has $listlength = 1$.

2. Using an array A of nodes to store information needed when doing operations on the data structure. (Indeed, we can write something like $S'.A$ to indicate the array is pointed by/stored in our data structure S', but for simplicity and clarity I will use A directly in this homework)

The array $A[1...k2^k]$ has every slot stores a pointer to a node that has two attributes. Let $1 \leq i \leq k2^k$. If i is a representative of a set in S, denote this set as $s_i \in S$. Then,

- $A[i].setsize$ that represents the size of $s_i \in S$, 0 otherwise (exactly when i is not a representative in S).
- $A[i].setnode$ that points to a node where set $s_i \in S$ is put (wrapped), NIL otherwise (exactly when i is not a representative in S OR s_i is a small set so not being put into any node of linked list yet).

For clarity, the attributes naming with "set" actually suggest set object rather than the verb set.

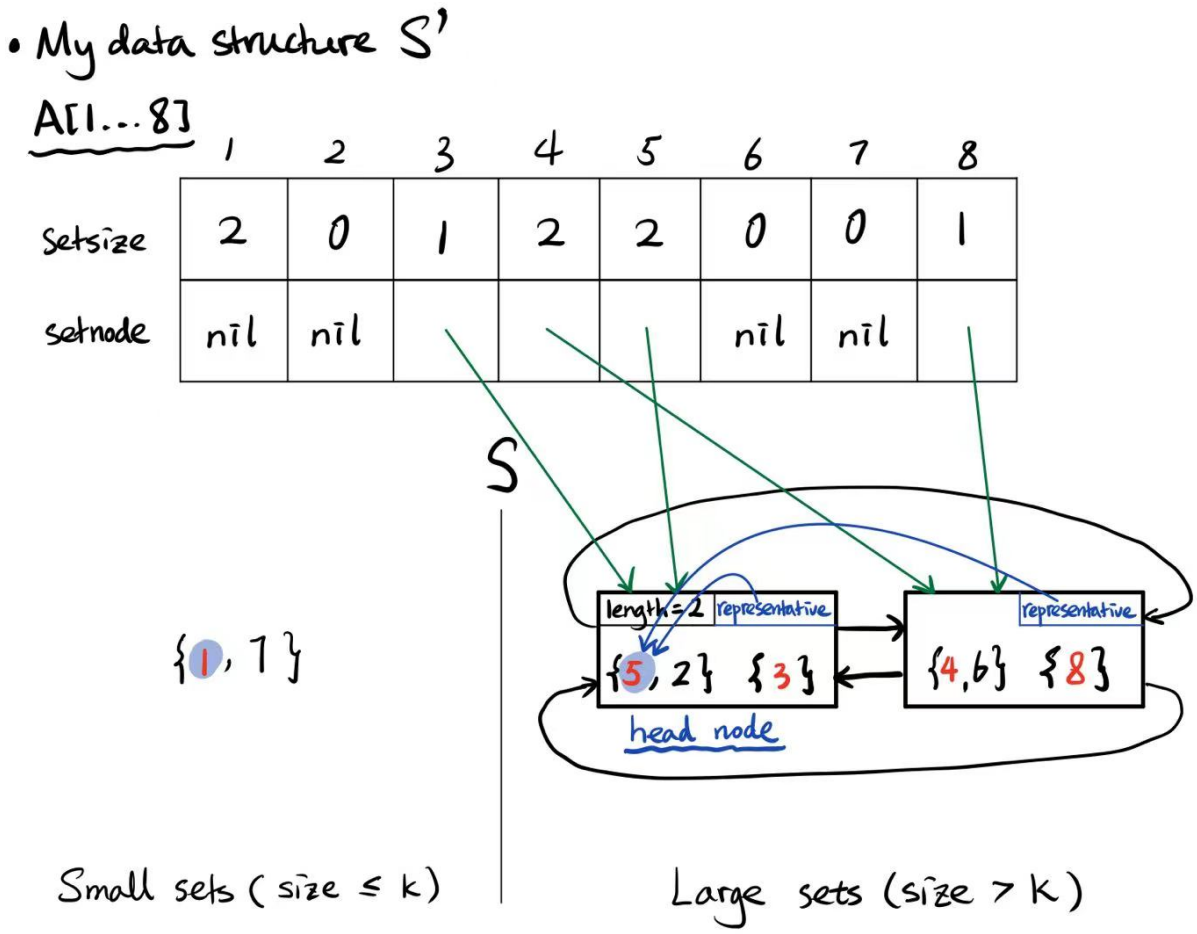
Initialization of my data structure:

Initialize the set representation: After $k2^k$ of MAKESETs operations, we have data structure S containing $k2^k$ (small) sets of size 1 containing $i \in \{1, \ldots, k2^k\}$ respectively, where each element in the set is the representative of the set. Notice that there is no large set yet, so we do not have node of linked list yet.

Initialize the array $A[1...k2^k]$: For $1 \leq i \leq k2^k$, $A[i].setsize = 1$ and $A[i].setnode =$ NIL.

Q2

Notice that all sets denoted in the form $\{\ldots\}$ are sets in S. Elements labeled in red are representatives of sets in S, elements highlighted in blue are representatives of sets in our data strucutre, 1 for set $\{1, 7\}$, 5 for set $\{2, 3, 4, 5, 6, 8\}$. The array A with nodes is drawn as two rows, the first row are the *setsize* attributes and the second are *setnode* attributes. The small set $\{1, 7\}$ (which is just represented as a set in S) is drawn separately from the large set $\{2, 3, 4, 5, 6, 8\}$ that is represented by linked list of length 2.



Q3

FINDSET(x) operation is actually divided into 2 cases, FINDSET with small set and FINDSET with large set. Firstly, in our data structure, every element is in some set in S, which means we can involke findset(x) for every x in $\{1, \ldots, k2^k\}$ to find the set representative i of the set containing x in S. Same as above, denote the set in S with representative i as s_i

- If $A[i].setnode = \text{NIL}$, then s_i is a small set, just return i.
- Else ($A[i].setnode$ points to a node, say v, in the linked list), so we know x is in a large set, we return $v.representative$ (if elements in S are wrapped by nodes then we return the elements store in node $v.representative$).

FINDSET(x) does not mutate our data structure, so all the properties of our data structure holds after it.

Q4

LINK(x, y) operation is a little bit more complicated, for which we have 4 cases to deal with.
Let us denote the sets in S with representative x and y as s_x and s_y respectively. Firstly, we check $A[x].setnode$ and $A[y].setnode$,

- Case 1: If $x = y$, or that one of $A[x].setnode$ and $A[y].setnode$ is NIL and the other is not (one small set one large set), then we simply return NIL.
- Case 2: Else if both $A[x].setnode = A[y].setnode = \text{NIL}$, then by our data structure property, s_x and s_y are both small sets. Then, we calculate the sum of sizes of s_x and s_y by $sizesum = A[x].setsize + A[y].setsize$.
  - Case 2.1: If $sizesum \leq k$, then the resulting set is still a small set, we can do link(x, y), and the resulting set s_{new} has representative i being returned by link(x, y). Hence, we need to mutate the

array by $A[x].setsize = 0$ and $A[y].setsize = 0$ and $A[i].setsize = sizesum$ (notice that it also works if i is any of x or y). And then, we returned i.

- Case 2.2: Else ($sizesum > k$), then the resulting set is a large set, we should constructed a doubly cicular linked list of one node, that stores s_x and s_y.
  (1) Without lose, let x (or y) be the representative of the new set (can be others but one of these two are easiest to implement), and this only node of the linked list is the head node containing *representative* pointer points to element x and a *listlength* attribute = 1.
  (2) And, we mutate array by making $A[x].setnode$ points to this newly constructed node.
  (3) Return representative x of the new set.

- Case 3: Else − both $A[x].setnode$ and $A[y].setnode$ point to some nodes in some linked list(s), then by data structure property, we know both x and y are representative of two large sets. We denote the list containing the s_x and s_y (thus x and y themselves) as $list_x$ and $list_y$. We do linking of sets (represented by lists) as following:
  (1) By our data structure property, we know that the head node of $list_x$ and $list_y$ are $A[x].setnode$ and $A[y].setnode$, we get the $n_1 = A[x].setnode.listlength$ and $n_2 = A[y].setnode.listlength$ as lengths of $list_x$ and $list_y$ respectively.
  (2) Then, we compare n_1 and n_2. Without loss of generality, assume that $n_1 \geq n_2$, then $list_x$ is at least long as $list_y$, so concatenate $list_y$ to $list_x$.
  (3) Firstly, we let x be the representative of linked (new) set, so we need to changing every pointers *representative* of nodes in $list_y$ points to x, by traversing through $list_y$ starting from $list_y$'s head node we got. So, $list_x$'s head node is the new head node of the concatenated linked list.
  (4) Then, since we have references to head nodes of $list_x$ and $list_y$, we perform linking of two circular doubly linked lists, by having the node previous to $list_x$ (the "tail" of $list_x$) mutually linked with the head of $list_y$ (the node wraps the set s_y in S containing y). And then, have the head node of $list_x$ mutually linked with "tail" of $list_y$ (node previous to head of $list_y$).
  (5) Lastly, we change the *listlength* pointer in head node of $list_x$ (which now is head node of linked list) to be $n_1 + n_2$.
  (6) The array A needs NO mutation since size of sets in S not change, and no set being newly wrapped into a node (or taking out from a node) in any list.
  (7) Return representative x of the new set.

After all, we know why every node stores two sets in S. A node is constructed only when two sets are stored in it after entering case 2.2 (link two small set to form a large set). And after that, every such large set only link with other large set, for which we do linking of linked lists thus not change the number of sets in a node anymore.

Q5

1. Credit Usages: We define 1 credit can pay for any one of following:

- FINDSET(x);
- Testing equality or inequality (i.e., comparison) and computing addition;
- Testing whether two sets being linked are both small, or both large or, one small one large;
- link(x, y);
- Mutating/Accessing constant numbers of attributes (pointers) of nodes in array;
- Constructing a list with one node and putting two sets in S into it;
- Updating/setting one *representative* pointer of a node, or one *listlength* pointer of a (head) node;
- Linking of two doubly circular linked list as described above.

Note that all above credit usage are valid, since every credit pays for operation(s) runs in $O(1)$ (espectially, FINDSET(x) is $O(1)$ since in both cases we only have accessing array A constant times and accessing some attributes and in addition a return statement).

2. Allocated Charges:

- Each FINDSET operation is given 1 credit.
- Each LINK operation is given 6 credits.

3. Credit Invariant: Denote $|s|$ as size of set s.

(1) All small sets s ($|s| \leq k$) contains $|s| - 1$ credits.

(2) All linked lists (large sets) contains at least $nk - n \log_2 n$ credits, where n is the length of the linked list.

Proof: In prior to any operation, the data structure contains $k2^k$ small sets. And the entire data structure has no credit stored into it yet, so does every small set s for which $|s| = 1$. The credit invariant 1 therefore holds. The credit invariant 2 vacuously holds since there is no linked list (large set) in our data structure yet.

Consider an arbitrary FINDSET(x) operation in the sequence. Suppose the credit invariant holds before this operation. Then, since it costs 1 credit, and is given 1, it pays for itself. The data structure is not being mutated, so credit invariant still holds.

Consider an arbitrary LINK(x, y) operation in the sequence. Suppose the credit invariant holds before this operation. There are four cases in our algorithm. The notation below is aligned with notation used in description of the algorithm in Q4.

- If $x = y$ or one of s_x and s_y is small set and the other is large. Then case 1 is executed, we have paid 1 credit for the checking to enter this case, and pay 1 credit for the return statement, donate the rest that is given to LINK to the charity. The data structure does not mutate thus credit invariants still holds.
- If both s_x and s_y are small sets and $sizesum \leq k$, then case 2.1 is executed, and the condition checking to enter this case, accessing/mutating array A, computing sum, link(x, y) operation, and return statement together uses 5 credits given to the operation. We have 1 credit remains, and we store it into the resulting set s_{new} of this LINK. Then, by induction hypothesis and newly assigned 1 credit, the resulting set s_{new} has $(|s_x| - 1) + (|s_y| - 1) + 1 = |s_x| + |s_y| - 1 = |s_{new}| - 1$. The other small sets does not mutate, so credit invariant 1 holds. No linked list is mutate either, so credit invariant 2 holds.
- If both s_x and s_y are small sets and $sizesum > k$, then case 2.2 is executed, and the condition checking to enter this case, together with (1) − (3) uses 4 credits. We store 1 into the node, and the other to charity.

  In this case, we know that $|s_x| + |s_y| \geq k + 1$. The node (linked list of length 1) has at least $(|s_k| - 1) + (|s_k| - 1)$ (from sets s_x and s_y stored into the node) plus 1 (from this LINK) credits, which is at least $(k + 1) - 2 + 1 = k$ credits and $k = 1 \cdot k - 1 \cdot \log_2 1$. Since there is no mutation to other linked lists/large sets, combined with induction hypothesis, credit invariant 2 still holds. Other small sets (those other than s_x and s_y) does not change so combine with induction hypothesis, credit invariant 1 holds afterwards.
- If x and y are representative of two distinct large sets, then case 3 is executed. By entering this case (checking condition) we use 1 credit. Notice that labels (6) are explanation rather than real-cost algorithm operation. Labels (1), (2), (4), (5), (7) each uses 1 credit, a total of 5, so we have used all 6 credits that is given to LINK. We have operation (5) (again, assume $n_1 \geq n_2$, so updating all *representative* pointers of nodes in $list_y$ points to x) that uses n_2 credits from data structure. So, together with the induction hypothesis, the concatenated linked list (of length $n_1 + n_2$) contains at least $(n_1 k - n_1 \log_2 n_1) + (n_2 k - n_2 \log_2 n_2) - n_2$ credits, we want to show which is at least $(n_1 + n_2)k - (n_1 + n_2) \log_2(n_1 + n_2)$. To show that, we have

$$
\begin{aligned}
&(n_1 k - n_1 \log_2 n_1) + (n_2 k - n_2 \log_2 n_2) - n_2 - \Big((n_1 + n_2)k - (n_1 + n_2) \log_2(n_1 + n_2)\Big) \\
&= -n_1 \log_2 n_1 - n_2 \log_2 n_2 - n_2 + (n_1 + n_2) \log_2(n_1 + n_2) \\
&= \big(n_1 \log_2(n_1 + n_2) - n_1 \log_2 n_1\big) + \big(n_2 \log_2(n_1 + n_2) - n_2 \log_2 n_2\big) - n_2
\end{aligned}
$$

$$
\begin{aligned}
&\geq \big(n_2 \log_2(n_1 + n_2) - n_2 \log_2 n_2\big) - n_2 \\
&\geq \big(n_2 \log_2(2n_2) - n_2 \log_2 n_2\big) - n_2 && \text{(since } n_1 \geq n_2\text{)} \\
&= n_2 \log_2(2n_2/n_2) - n_2 \\
&= 0
\end{aligned}
$$

Since other linked lists and all small sets are not changed, combining with induction hypothesis, credit invariants 1 and 2 both hold afterwards. ■

4. Conclusion: Since any nodes has at least $k+1$ elements, so in the universe of $\{0, \ldots, k2^k\}$, there are at most $\frac{k2^k}{k+1}$ nodes, which is at most $\frac{k2^k}{k} = 2^k$ nodes. That is, for every linked list of length n, $n \leq 2^k$ thus $n \log_2 n \leq nk$ (i.e., the quantity $nk - \log_2 n$ stated in credit invariant 2 always ≥ 0). Therefore, we have shown that we always have enough to pay for all kinds of sequence of operations containing FINDSET and LINK. So, we conclude that the amortized cost of FINDSET and LINK are at most their allocated charges, which are all in $O(1)$ thus constant.