**CengBook By Erişken the Jr. Macho Programmer**

***Q1.1 sortBook:***

We can say that *SortBook* simply scans the users from *0* to *x-1* where *x* is the number starting from the number of users registered to *CengBook* and going to *1* and checks whether users need to be swapped due to the result of comparison of two users by comparator function given as pointer.

If result requires users to be swapped, then their indexes are changed. By every step, the one that should go to the last place gets closer to its destination. When we do it times, we get a sorted list. An illustration of the process is given below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
| *i* |  |  |  |  | *x* |
| Yennefer | Triss | Dandelion | Ciri | Geralt | Vesemir |

Suppose that we want to sort this Book alphabetically in ascending order. For every *x* from *t* to *1*, *i* will go from *0* to *x-1* and we will check whether needs to be swapped with . If yes, then we swap them.

In first step, as user *“Yennefer”* is alphabetically greater than *“Triss”*, users need to be swapped. After swapping, we increase *i.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
|  | *I* |  |  |  | *x* |
| Triss | Yennefer | Dandelion | Ciri | Geralt | Vesemir |

Again, *“Yennefer”* is alphabetically greater than *“Dandelion”* so they are swapped and *i* is increased.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
|  |  | *I* |  |  | *x* |
| Triss | Dandelion | Yennefer | Ciri | Geralt | Vesemir |

If we follow this steps, after *i* reaches 4, table will be like that:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
|  |  |  |  | *i* | *x* |
| Triss | Dandelion | Ciri | Geralt | Vesemir | Yennefer |

Then we decrease *x* by *1* and return *i* to *0.*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
| *i* |  |  |  | *x* |  |
| Triss | Dandelion | Ciri | Geralt | Vesemir | Yennefer |

If we continue following our steps, then we should change the places of *“Triss”* and *“Dandelion”*.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
|  | *i* |  |  | *x* |  |
| Dandelion | Triss | Ciri | Geralt | Vesemir | Yennefer |

It can be seen that this algorithm will lead to a table like that before decreasing *x* and resetting *i*:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
|  |  |  | *i* | *x* |  |
| Dandelion | Ciri | Geralt | Triss | Vesemir | Yennefer |

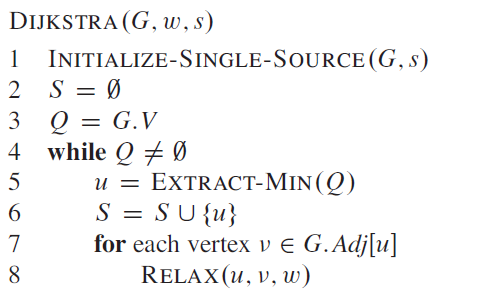
Then it will work until *x* reaches *0* and when the process is finished, our table will be sorted alphabetically in ascending order like that:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *0* | 1 | 2 | 3 | 4 | 5 |
| *i,x* |  |  |  |  |  |
| Ciri | Dandelion | Geralt | Triss | Vesemir | Yennefer |

(in implementation, *s* is *“userPointers”* array, *x* is *index1* and *i* is *index2 variable*).

***Q1.2 shortestPath:***

In this function, we try to reach our target using solution steps of commonly known graph algorithm *“Djikstra’s Shortest Path Algorithm”.* We prefer it over another popular algorithm *“Bellman-Ford Algorithm”* because we have single source and single destination and also our edges are positive-weighted so *“Djikstra’s Shortest Path Algorithm”* is the optimal choice for our need.



Pseudo-Code for *Djikstra’s Shortest Path Algorithm*, taken from *Introduction to Algorithms (Rivest et al., 1989)*

As an interpretation, we have a FIFO (first in first out) queue *Q* holding our users to be visited. At the beginning, as we should visit our source user *S* firstly, we add them to our *Q.*

Then we enter our loop which will run until *Q* is emptied or target user *T* is reached. We get the next user from *Q.* In standard method of *Djikstra’s Shortest Path Algorithm (DSPA),* we use a priority-queue in order to get the minimum distanced node. However, as in our situation all edges are equally long so we don’t need a priority-queue as the first element in our *Q* will also be the closest element.

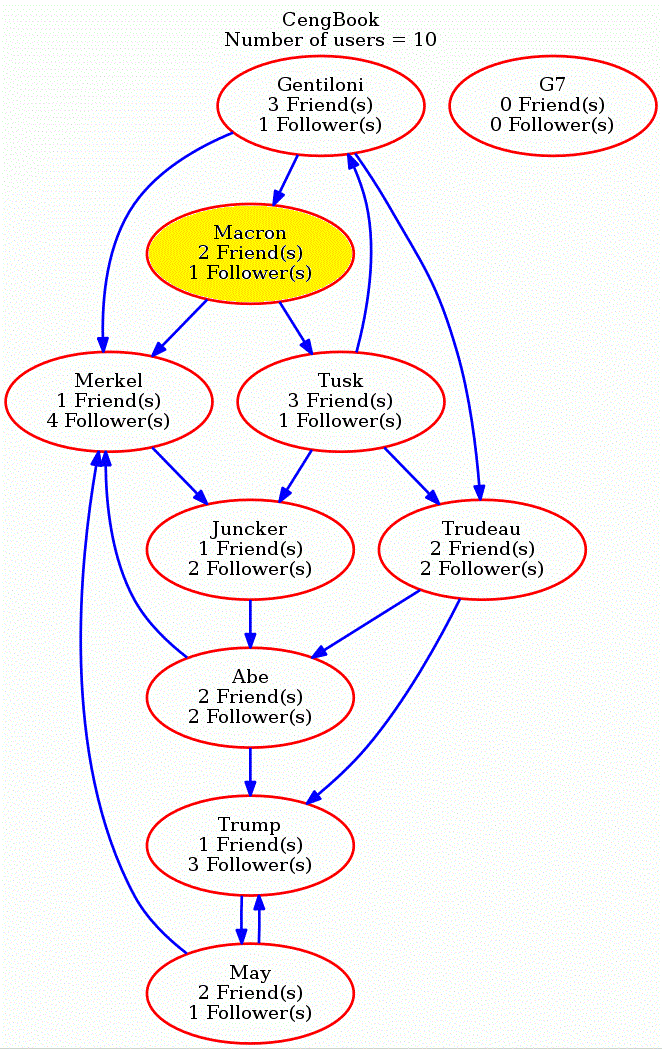
After selecting the user we are going to visit this time, we check their friends. If they are not marked as visited before, we mark them with our current user’s index. After marking, we put them into *Q* to visit them when the time comes.

If *Q* gets emptied without *T* being visited (marked as *-2* in our implementation), that means a route from *S* to *T* cannot be established so we return **NULL** as our function has failed.

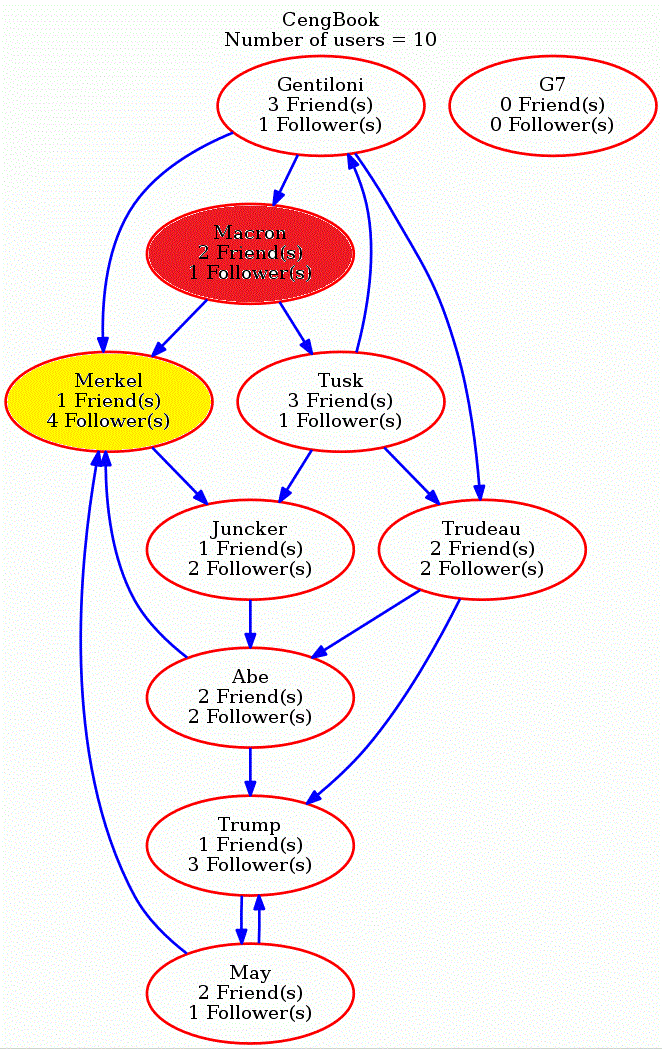
If we could establish a route from *S* to *T,* we can find the path by traversing inversely from *T* until we reach *S.* We can do this as we marked our users with their followers in our loop. If we get *-1* that means we reached our *S* so we can return our *friendPtrChainArray.*

An illustration of *DSPA* in the graph to find the shortest path between *“Macron”* and *“Trump”* is given as follows:

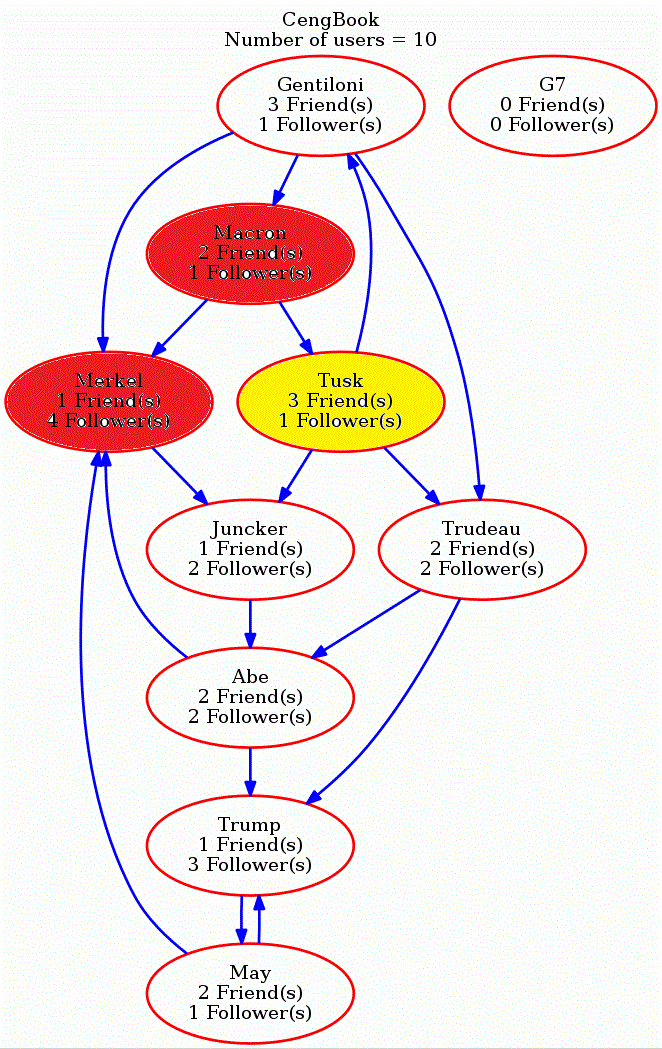
(Yellow colored user is the user we are currently visiting; red ones are the ones we passed)

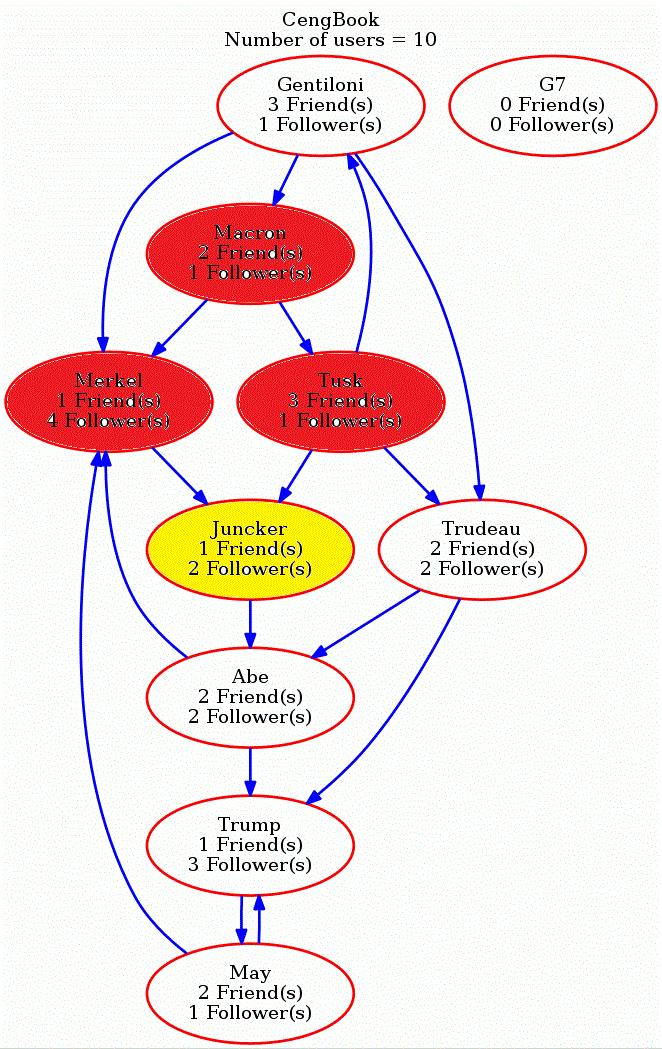


Q:Merkel, Tusk

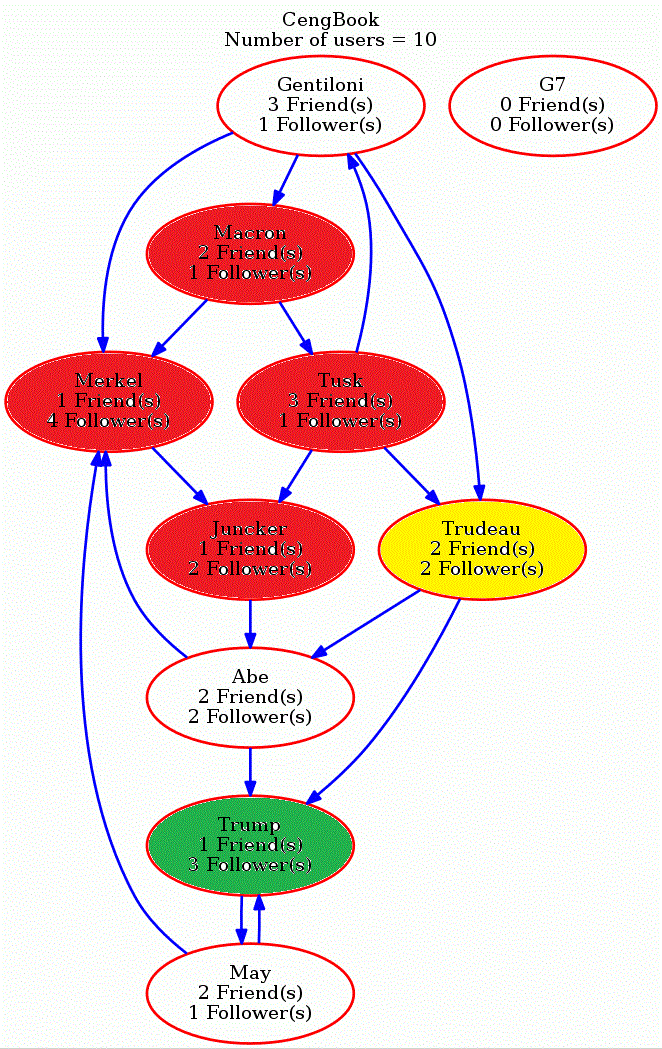


Q: Tusk, Juncker

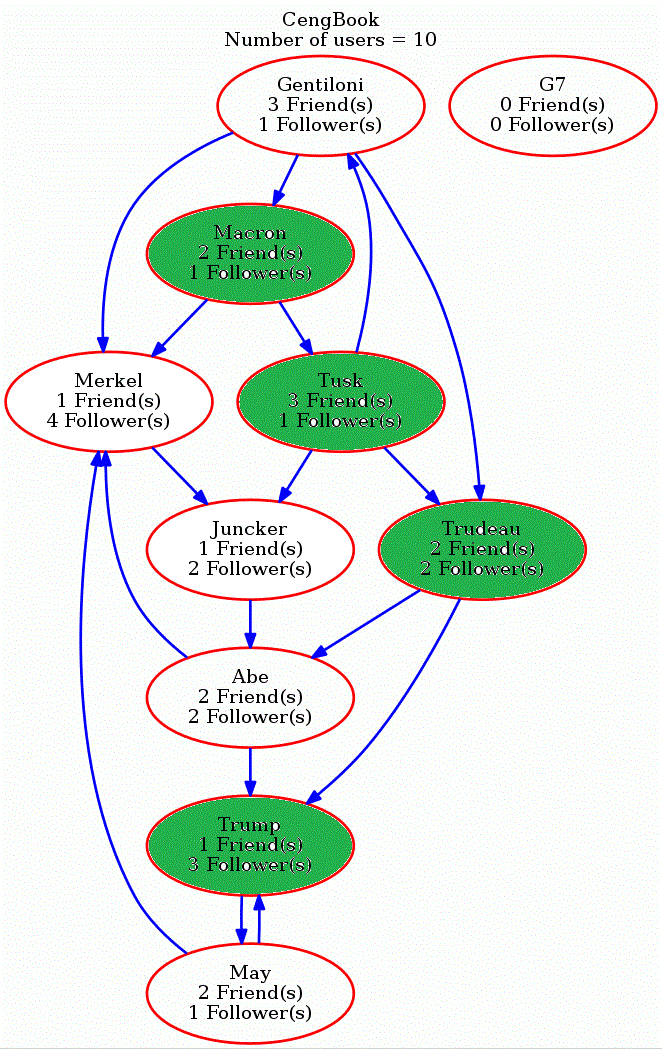
**

Q: Juncker, Trudeau

Q: Trudeau, Abe



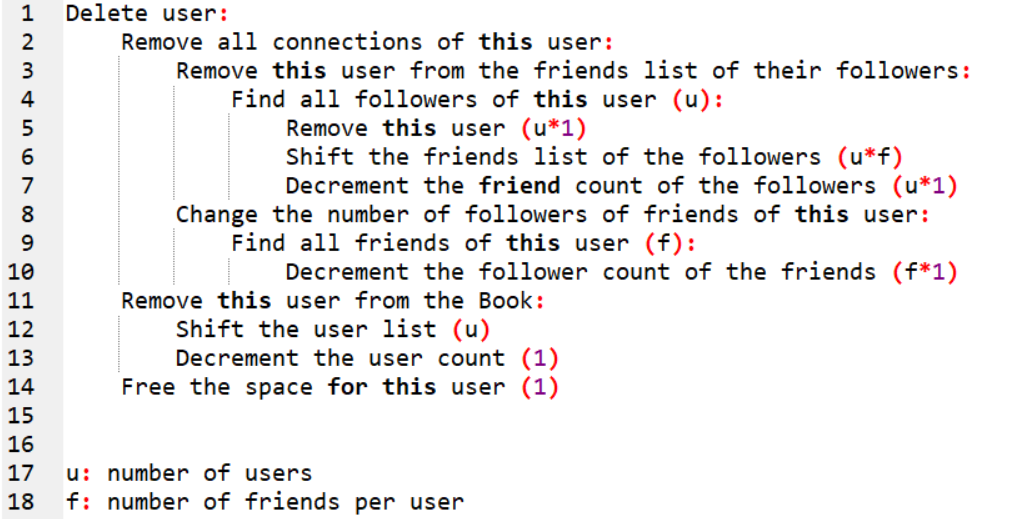
As *“Trump”* is friend of *“Trudeau”,* we can conclude that we reached our target so we terminate our loop. Now *“Trump”* has follower *“Trudeau”, “Trudeau”* has follower *“Tuck”* and *“Tuck”* has follower *“Macron”* also known as our source *S*. So our shortest path from *“Macron”* to *“Trump”* will be as follows:



Finally, we can return our route by going backwards from *“Trump”* to *“Macron”*.

**Q2 *Time Complexity of Deleting User:***

The pseudo-code of algorithm for deleting a user is as follows:



We can find the time complexity function

= *u\*(f+2) + f + u + 2 = O(u\*f)*

# References

Thomas H. Cormen, Charles E. Leiserson, Ron Rivest, Clifford Stein(1989). *Introduction to Algorithms* (p. 658). MIT Press.