Union() Method

The Union method starts by creating a new DLLSet which is used to keep the union between the given two sets. Then the head node is set to point to the tail node and vice versa. Three reference nodes are used, a and b are used traverse the elements in the two given sets, and the third, newInsert, will be used to traverse the union of the two sets. After the new set and reference nodes have been assigned, a for loop is executed with will loop through m times, where m equals the size of the two given sets. The goal of this loop is to compare two elements in the given sets and react accordingly.

The first condition that is checked out of three is if ((a.element is less than b.element and a is not at the tail of the list) or b is at the end of the list) if either of these two are true, this means that a.element should be inserted into the set. In this condition a new DLLNode, insert, will be created with a.element as its element, a previous DLLNode pointer to the previous node in the list newInsert, and points to null. The previous node in the list, newInsert, is set to point to this newly created node, the size of the set is increased, the node pointer newInsert is set to point to the newly created node and since a.element was inserted, a is set to point to the next node in its set.

The second condition is if ((a.element is greater than b.element and b is not at the tail of the list) or a is at the tail of the list) than this means that b.element should be input in the set. A new DLLNode is created with element b.element, previous node pointer set to the previous node newInsert, and points to null. The previous node newInsert is set to point to the new node Insert. The size of the set in increased. The reference node newInsert is set to point to the new Insert node, and b is set to point to the next node in its set.

The last condition will only be executed if the elements in both a and b are equal. If this is true then a and b are both set to point to the next node in their lists and the value that is used to count the number of elements checked in the lists, I, is increased on top of the increase that will execute from the for loop because two values were covered in this branch of the if/else statement.

After this for loop is executed, the previous of the new set tail pointer is set to point to the last node that was created, and the next pointer of the last node that was created will point to the tail of the list. The new DLLSet that was made will be returned to the function caller.

This method will run *O*(m) at the worst case, where m is the sum of the sizes of the two passed DLLSets. This is because the for loops condition is from 0 to m-1. The additional memery usage is *O*(1) because there is always only 3 reference nodes used in this method.

Intersection() Method

The Intersection() method starts be creating a new DLLSet which is used to store the intersection of the two DLLSets. The size of the DLLSet is set to zero, and a new head at tail node are created which point to eachother. Three reference nodes are used, a and b which are used to traverse the two DLLSets that are passed to the function, and a third newInsert which is used to traverse the new DLLSet that is created. After these have been executed a for loop is executed which will loop m times, where m is the addition of the two sizes of the passed DLLSets. The goal of this loop is to compare an element from one set to the other.

The first condition of four is if a.element is equal to b.element. If this is true a new node is created, with element value of a.element, which has a previous pointer to newInsert, the previous node in the new DLLSet, and has a next pointer value of null. The previous node, newInsert, is set to point to the new Insert node, both a and b are set to point to the next node in their DLLSet, the size of the new DLLSet is increased, the node newInsert is set to point to the new Insert node and the for counter, I, is increased.

The second condition is if either of the two nodes is equal to the tail of their respective DLLSet. If this happens the for loop will exit, because there are no more numbers to be compared that will be equal to a number in the opposite set.

The third condition is if a.element is less than b.element. If this is true then a is set to point to the next node in its DLLSet.

The last condition is if a.element is greater than b.element. If this is true then b is set to point to the next node in its DLLSet.

This method will have a worst case run time of $O(\mathrm{m})$, where m is the sum of the sizes of the two passed DLLSets. This is because the for loops condition is from 0 to m-1. The additional memery usage is $O(1)$ because there is always only 3 reference nodes used in this method.

<u>FastUnion() Method</u>

The FastUnion() method starts by assigning the value sArray.length/2 to j. Then a while loop is executed as long as this value j is greater than 1, which means that the array has more than 2 elements in it. In the array, a for loop will run and be looped from 0 to the current value of j. In this loop will combine values in the loops in pairs and will keep doing this until there are only two sets in the array. After the while loop is exited, this will return the union of the first half of the array with the second half.

<u>recUnion() Recursive Method</u>

Assuming than the array is of size n, then the base case, where n = 1 has a run time on $T(1) = c$. If $n > 1$, then $T(n) = 2T(n/2) + cn$. This is called every recursive function call, with an *n* value of n/2, so the second time this Is called $T(n/2) = 2T(n/2^2) + cn/2$. Substituting makes, $T(n) = 2^2T(n/2^2) +2cn$. The third time $T(n/2^2) = 2^2T(n/2^3) + 2cn/2$. Substituting makes, $T(n) = 2(2^2T(n/2^3)+cn) +2cn = 2^3T(n/2^3) + 3cn$. If this is done i times, $T(n) = 2^iT(n/2^i) + icn$. If we assume that $2^i = n$, this becomes $T(n) = nT(1) + cn \log_2 n$. We know that $T(1) = c$, which means $T(n) = cn(1 + \log_2 n)$, if we ignore the constant terms we will get that the worst case runtime is $\Theta(n\log_2 n)$.