Yash Avinash Patole

Course : CS600

Homework 1

CWID: 10460520

Chapter 7

Exercise 7.5.5

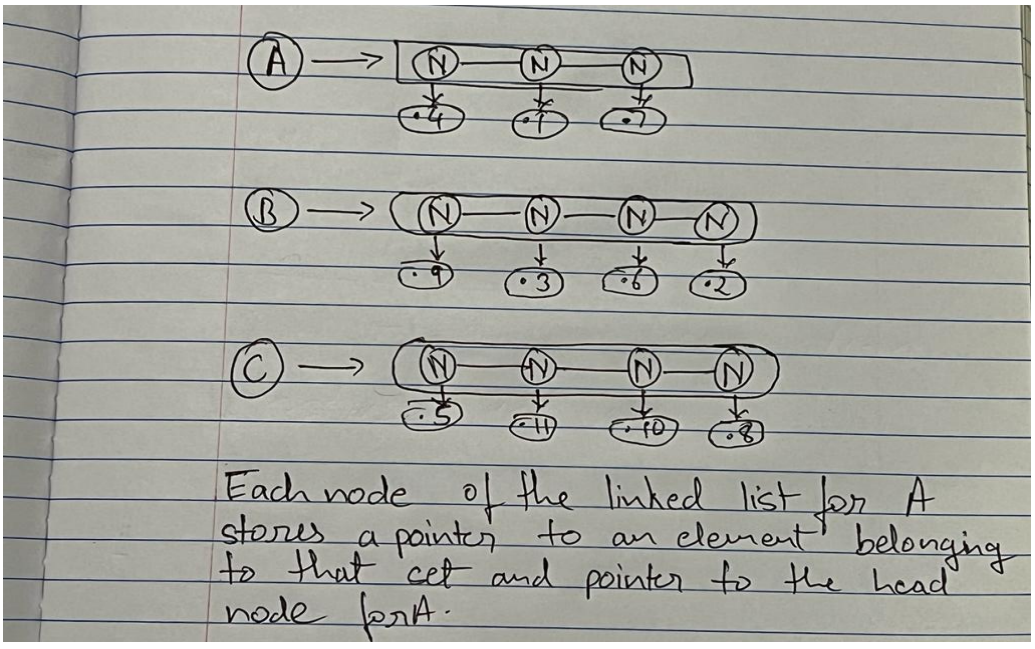
One additional feature of the list-based implementation of a union-find structure is that it allows for the contents of any set in a partition to be listed in time proportional to the size of the set. Describe how this can be done.

Answer:

We can list every element in the set in O(1) time for each element because the head node of the set has links to all the elements in the list, therefore listing n elements will take O(n) time. Consequently, the items can be listed in a time-proportional manner depending on the size of the set.

Consider, we have three disjoint sets A = {1, 4, 7}, B = {2, 3, 6, 9}, and C = {5, 8, 10, 11, 12}.

Assuming as the structure in the following figure:



Algorithm To find all elements of the set-in which x belongs:

Algorithm Find-Set(x):

```
temp= x.head ;
temp = temp->next;  //reach to first node of linked list
while (temp != NULL)
{
print temp->data;
temp=temp->next;
}
```

End

Time complexity: O(n) where n is the number of elements in the set.

Exercise 7.5.9

Describe how to implement a union-find structure using extendable arrays, which each contains the elements in a single set, instead of linked lists. Show how this solution can be used to process a sequence of m union-find operations on an initial collection of n singleton sets in O (n log n + m) time.

Answer:

We can still do the union-find operation in O (n log n + m) time as the find operation will only require O(n) time and all add and remove operations in an extensible array will take O(n) time. Additionally, since an array uses indexing to locate each element, no head node is required.

Following algorithm shows the process to find operations on an initial collection.

Algorithm makeSet():

```
    Create an extendable array S
    for each singleton element, x do
        Add x to S
```

Algorithm find(x):

```
    return S which contains x
```

Algorithm union (u, v):

```
if the set u is smaller than v then
        for each element, x, in the set u do
        remove x from u and add it v
else
        for each element, x, in the set v do
                remove x from v and add it u
```

Exercise 7.5.21

Answer:

A (n + 2) by (n + 2) board is first initialized. Then, in each cell, we arrange black pieces along the top (Set BU) and bottom (Set BB) edges and white pieces along the left (Set WL) and right edges of the board (Set WR). Checking for a winning move is simple thanks to this setup. The number of MakeSet operations required is O(n).

To make a move, we first place a piece at the desired location (i.e., create a set there), and then we look to see if any adjacent cells are occupied by pieces of the same color. In this case, union the newly produced set with the set containing the next cell (there are therefore a maximum constant number of union operations). Next, we determine whether the present piece unions are BU and BB or WL and WR. If it does, this is a winning move, and we continue playing, otherwise.

Once we find a winning move we count the connected gold pieces to this color using FindSet operation and counting all those pieces which are part of Set G (gold pieces) and those are the bonus points.

Initializing the board takes O(n) time, and each move requires one Union and two FindSet operations to determine whether the boundary sets are union. To calculate bonus points after the successful move, we will need k' Find operations. We will therefore have n Union-Find operations and k' Find operations during the duration of a game with n moves in O(n + k) time.

Exercise 8.5.12

Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your approach to three colors?

Answer:

Yes, we can expand this strategy to n colors. Let's imagine we have a third color, green, in addition to blue and red. We want the array to be arranged so that all the blue components appear before the red elements and all the red elements appear before the green elements. First, we'll arrange the items of the array so that those that are blue in color are listed first, followed by those that are not blue. We would have solved half of our difficulty in this method. We will now take the subarray that solely consists of red and green components, and we will apply the same procedure to order all of the red elements prior to the green elements.

To accomplish this, we can utilize Quick-Sort with the left side color serving as the pivot and a custom comparer that directs all components to the left portion if they share the pivot's color or the right part otherwise.

Exercise 8.5.22

Answer:

This problem can be solved by diving the sequence S in two parts L and R while checking for their respective majorities by comparing two candidates since they are integer IDs.

Depending on these following scenarios can happen:

- x is a majority from both L and R, then x is the majority of L + R
- x is a majority from L and y is a majority from R, then we count votes for both x and y which will take O(n) time and decide who won
- x is a majority from L and y is a majority from R, and both have the same votes, then it’s a tie

We can use the following method to determine the winner

Algorithm majorityWinner(S, i, j):

```
Input: A sequence S and index i and j
Output: Winner amongst subsequence S[i:j]
if i == j then
        return S[i]
x ←majorityWinner(S, i, (i + j) / 2])
y ←majorityWinner(S, (i + j) / 2 + 1, j])
if x == y then
        return x
// return winner by counting votes
else if x.votes > y.votes then
        return x
else if x.votes < y.votes then
        return y
else
        return "tie"
```

The above algorithm will take O (n log n) to come up with a winner. Initially it will be called with majorityWinner(S, 1, n) assuming sequence indexing is from 1 to n.

Exercise 8.5.23

Consider the voting problem from the previous exercise, but now suppose that we know the number k < n of candidates running. Describe an O (n log k)-time algorithm for determining who wins the election.

Answer:

Since there can only be n candidates, and since they are integers, they are easily hash able and unique, a lookup table or set of candidate IDs can be created to handle both difficulties. Put (x, x.votes) and get(x) will give us the votes for candidate x. We can allocate a specific number of votes to each contender in the lookup table. We can determine the winner in O(n) time since counting votes will take O(n) time, all hash functions will take O (1) time, and counting the maximum number of votes will take O(n) time.

Exercise 9.5.17

Suppose you are given two sorted lists, A and B, of n elements each, all of which are distinct Describe a method that runs in O (log n) time for finding the median in the set defined by the union of A and B.

Answer:

We can find the median of two sorted arrays by repeatedly finding medians of the two arrays and comparing and finding the median again by breaking them in sub arrays. The time complexity of the algorithm defined below is O(log n) as the general equation which is T(n) = T(n / 2) + O(1).

As, we are breaking into a problem of half size each time and the time for solving problem of size 1 is constant O (1) therefore the time complexity is O(log n).

Algorithm findMedian(A, B, n):

Input: Two sorted arrays A and B containing n elements

Output: Median of union of A and B

```
if n == 1 then
        return A[0] + B[0] / 2
else if n == 2 then
        return (max{A[0], B[0]} + min{A[1], B[1]}) / 2
else
        m1 ←median(A, n)
        m2 ←median(B, n)
        if m1 > m2 then
                return findMedian(A[0: n / 2], B[n / 2: n], n / 2)
        else
                return findMedian(A[n / 2: n], B[0: n / 2], n / 2)
```

Algorithm median(A, n):

Input: Sorted array A containing n elements

Output: Median of A

```
if n is even then
        return (A [n / 2] + A[n / 2 – 1]) / 2
else
        return A[⌊n / 2⌋]
```

Exercise 9.5.24

Answer:

Let A be array of size n, containing the votes.
We need to find if any student has more than n/2 votes.
Algorithm:
Step 1)
To determine the median of the sorted array A, use the Quick-Select. Finding the median using quick select requires O(n) estimated time.

The quickselect algorithm identifies the kth smallest element in the array in O expected time because it accepts an array as input and k. (n). Although Quick Select's worst case is O (n2).

To find median we have to find n/2 th smallest element in the array.

median = Quickselect(A, n/2)

Step 2)

Now count the number of occurrences of median element in the array by linear search. This will take O(n)

If a student number with majority votes (more than n/2 times) exists, it will be the median of the sorted Array A, we just need to verify if that median we found in step 1 has indeed received the majority votes.

```
//count the number of occurrences of the median element
count = 0
for i = 1 to size(A):
   if(A[i] == median)
      count = count+1
//if the number of occurrences is greater than n/2 then it is majority
if count > n/2:
   print ("Yes, student has been found with alternative votes.")
else:
   print ("No student found with majority of votes. ")
```

The algorithm stated above run in O(n) time complexity while looping the array A.