Ali Abdullah Ahmad
20031246
Assignment 4
CS 600

Q1. No. 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.
Ans.

In a list-based union-find structure, each set is represented as a linked list, where all elements in a set are linked together. Additionally, each set maintains a leader (or representative) node, which keeps track of the head of the list and possibly other metadata such as the size of the set.
Listing Elements in a Set Efficiently
To list all elements of a set in time proportional to its size:

1. Access the Representative Node:
    - Each element maintains a reference to its set representative.
    - To retrieve all elements of a given set, we first identify its representative.
2. Traverse the Linked List:
    - Since all elements of a set are stored in a linked list, starting from the representative node, we can iterate through the list.
    - Each element can be accessed in constant time, and the entire list is traversed in O(s), where s is the size of the set.

Advantages of This Approach

- Efficient Union Operation: When merging two sets, the smaller list can be appended to the larger one, ensuring that subsequent operations remain efficient.
- Direct Access: The linked structure enables direct listing of set members without additional searches.

Thus, using a linked list representation, the list operation runs in O(s), proportional to the number of elements in the set, making it efficient for retrieving all members.

Q2. No. 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.

Ans.

we can implement a union-find structure using extendable arrays, where each set is represented as a dynamic array that grows when elements are added.

Implementation Steps

1. Initialization:
    - Start with n singleton sets, where each element is its own set.
    - Maintain an array of pointers to dynamically allocated arrays, each representing a set.
    - Maintain a leader array to track which set an element belongs to.
2. Find Operation (O(1)):
    - Each element directly points to the array that represents its set.
    - Simply return the representative of the set in O(1).
3. Union Operation (O(logn)):
    - To merge two sets, identify the larger and smaller set.
    - Append the smaller set's elements to the larger set's array.
    - Update the leader array for all moved elements.
    - If arrays need resizing, this takes **amortized O(1) ** per operation.

Time Complexity Analysis

- Initialization: Setting up n singleton sets takes O(n).
- Find Operation: Each find is O(1).
- Union Operation:
    - The union-by-size heuristic ensures that an element moves at most O(logn) times across sets.
    - Across mmm union operations, this results in a total complexity of O(nlogn).

Overall Complexity

O(n logn+m)

where:

- O(nlogn) accounts for the set merging steps.
- O(m) accounts for find operations and individual union operations.

Q3. No. 7.5.21

Consider the game of Hex, as in the previous exercise, but now with a twist. Suppose some number, k, of the cells in the game board are colored gold and if the set of stones that connect the two sides of a winning player's board are also connected to $k' \leq k$ of the gold cells, then that player gets k' bonus points. Describe an efficient way to detect when a player wins and also, at that same moment, determine how many bonus points they get. What is the running time of your method over the course of a game consisting of n moves?

Ans.

Efficient Strategy for Detecting a Winning Move and Bonus Points

We initialize the board with:

- Black pieces along the top (Set BU) and bottom (Set BB) edges.
- White pieces along the left (Set WL) and right (Set WR) edges.
- Gold cells (Set G) scattered across the board.

Each move involves:

1. Placing a Piece: Create a new set for the piece using MakeSet (O(1)).
2. Merging Adjacent Pieces: If adjacent cells have the same color, perform Union operations (O(1)).
3. Win Detection: Check if the newly formed component connects BU to BB (for black) or WL to WR (for white) using FindSet (O(1)).
4. Bonus Calculation: If a winning move is found, count the number of gold pieces in the winning component using additional FindSet operations (O(k')).

Time Complexity Analysis

- Board Initialization: O(n)
- Per Move:
    - Union Operations: O(1)
    - Win Detection (FindSet calls): O(1)
- After Winning Move:
    - Counting Gold Pieces: O(k')
- Total Complexity Over n Moves: O(n + k')

This ensures efficient game management while maintaining quick win detection and bonus point calculation.

Q4. No. 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?

Ans.

Two-Color (Red and Blue) Sorting (In-Place)

We can solve this problem using the two-pointer technique in O(n) time with O(1) space.

Algorithm (Two-Color Sorting)

1. Initialize Two Pointers:
    - Let left = 0 (tracks the position for blue elements).
    - Let right = n-1 (scans the array).
2. Partition the Array:
    - Traverse the array from left to right.
    - If a blue element is found, swap it with S[left] and increment left.
    - Continue until right is reached.

Three-Color Sorting (Red, Blue, and White)

We can extend the approach to three colors using Dutch National Flag Algorithm.

Algorithm (Three-Color Sorting)

1. Initialize Three Pointers:
    - low = 0 (tracks blue elements).
    - mid = 0 (traverses the array).
    - high = n-1 (tracks red elements).
2. Partition the Array:
    - If S[mid] is blue, swap with S[low] and increment both low and mid.
    - If S[mid] is white, just increment mid.
    - If S[mid] is red, swap with S[high] and decrement high (without incrementing mid).

Q5. No. 8.5.22

Suppose we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an O(n log n)-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

Ans.

We need to determine the candidate with the most votes in a sequence S of n elements, where each vote is an integer representing a candidate ID. Since we do not know the number of candidates in advance, we will use sorting and counting to find the winner in O(nlogn) time.

Algorithm Steps:

1. Sort the Sequence:
    - Sorting takes O(nlogn) using a standard comparison-based sorting algorithm (e.g., Merge Sort or QuickSort).
2. Count Votes:
    - Iterate through the sorted sequence and count consecutive occurrences of each candidate.
    - Maintain a variable to track the candidate with the highest count.
    - This counting step takes O(n).

Pseudocode:

```
def find_winner(S):
    S.sort()  # O(n log n) sorting

    max_votes = 0
    current_votes = 1
    winner = S[0]

    for i in range(1, len(S)):
        if S[i] == S[i - 1]:  # Same candidate as previous
            current_votes += 1
        else:
            if current_votes > max_votes:
                max_votes = current_votes
                winner = S[i - 1]
            current_votes = 1  # Reset count for new candidate

    # Final check for last candidate
    if current_votes > max_votes:
        winner = S[-1]

    return winner
```

Time Complexity Analysis:

- Sorting O(nlogn) dominates the complexity.
- Linear pass O(n) for counting votes.
- Overall Complexity: O(nlogn).

Q6. No. 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.

Ans.

O(n log k)-Time Algorithm for Determining the Election Winner

Since we now know that there are only k candidates (where k<n), we can improve the previous O(nlogn) sorting-based solution using a heap-based counting approach that runs in O(nlogk) time.

Algorithm Steps:

1. Use a Hash Map to Count Votes:
    - Traverse the sequence SSS and use a hash map (dictionary) to store vote counts for each candidate.
    - This step takes O(n) since each insertion and lookup in a hash table is O(1).
2. Use a Min-Heap of Size k to Track the Top Candidate:
    - Insert candidate counts into a min-heap of size k.
    - Each insertion/removal in the heap takes O(logk), leading to an overall O(klogk) time complexity.
3. Extract the Candidate with Maximum Votes:
    - Once all votes are counted, extract the candidate with the highest count from the heap.
    - This takes O(logk).

Time Complexity Analysis:

- Vote Counting (Hash Map): O(n)
- Heap Insertions (for k candidates): O(klogk)
- Heap Extract to Find Winner: O(logk)
- Total Complexity: O(n+klogk)=O(nlogk)

Q7. No. 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.

Ans.

We are given 2 sorted lists, A and B of n elements each, all of which are distinct.

Method to calculate Median in the set defined by Union of A and B:

a) We start by calculating the medians medA and medB of sorted list A and B respectively.
b) If medA and medB are equal, then return either medA or medB.
c) Else if medA is greater than medB then median is present in one of the two subarrays defined below:
   • From first element of A to medA (A[0] …...A[n/2]) or
   • From medB to last element of B (B[n/2] to B[n-1])
d) Else if medA is smaller than medB them median is present in one of the two subarrays defined below:
   • From medA to last element of list A (A[n/2] …...A[n-1]) or
   • From first element of list B to medB (B[0] …...B[n/2)
e) Recursively call above process until the size of both the subarrays becomes 2.
f) Using formula to calculate the median when size of the lists become 2 is:

Median = (max(A[0], B[0]) + min (A[1], B[1])) / 2

The above algorithm will run in O(log n) time.

Example:

A = {3, 7, 13, 21, 34}
B = {4, 8, 15, 25, 35}
medA = 13
medB = 15
medA < medB (apply step d)
[13, 21, 34] and [4, 8, 15]
In above two medA and medB is 21 and 8 respectively
Now medA > medB (apply step c)
Subarray becomes [13, 21] and [8, 15]

Now, we use the formula to calculate median which we have given in Step F:

Median = (max(13, 8) + min(21, 15)) / 2
= (13 + 15) /2
= 14

Q8. No. 9.5.24

Suppose University High School (UHS) is electing its student-body president. Suppose further that everyone at UHS is a candidate and voters write down the student number of the person they are voting for, rather than checking a box. Let A be an array containing n such votes, that is, student numbers for candidates receiving votes, listed in no particular order. Your job is to determine if one of the candidates got a majority of the votes, that is, more than n/2votes. Describe an O(n)-time algorithm for determining if there is a student number that appears more than n/2 times in A

Ans.

Algorithm: Using a Hash Table (Dictionary)

1. Initialize a Hash Table H:
    - Keys: Student numbers (candidates).
    - Values: Count of votes each student received.
2. Iterate Through Array A:
    - If the student number exists in H, increment their vote count.
    - If the updated count exceeds n/2, return True.
    - Otherwise, add the student number to H with an initial count of 1.
3. If No Majority Found, Return False.

Time Complexity Analysis

- Hash Table Operations (Insert & Update): O(1) on average.
- Iterating Through n Votes: O(n).
- Overall Complexity: O(n)