CSC263 A5

Nena Harsch and Natalia Tabja

October 2024

Question 1 (written by Nena, verified by Natalia)

Part (a)

We first want to define that \$1 pays for one pairwise comparison. From the question, we know that the actual cost of add is 0 and reduce is αn. We will assign a charge of 2α to the add operation and \$0 to the reduce operation.

Part (b)

We will define two state invariant (SI) as follows:

1. Each element in the list stores \$$2\alpha$.
2. The sum of the charges of ops $\geq$ The sum of costs

Proof:

- Base Case: The size of the list is 0

  1. We know that the list contains 0 elements, therefore 1 holds as there are 0 charges stored.
  2. The sum of charges = the sum of costs = 0 since the list is empty, therefore 2 holds.

- Induction Step: Suppose that the SIs hold for $m-1$ operations [IH]. We want to prove that the SIs hold for the m^{th} operation.
  - The m^{th} operation is add.
    1. For a list of n elements, by the IH and SI1 we have \$$2\alpha n$ charges. When we add another element, by our definition in part a we will charge another \$$2\alpha$. Therefore our list will now have a total of \$$2\alpha(n+1)$ charges, and since we have $n+1$ elements, SI1 holds.
    2. There was an additional charge of 2α when we added an element and no additional costs. Therefore,
       sum of charges of m ops = sum of charges of m-1 ops + 2α > sum of charges of m-1 ops
       sum of costs of m ops = sum of costs of m-1 ops
       and thus by this, the IH, and SI2, we can conclude
       The sum of the charges of m ops > sum of charges of m-1 ops $\geq$ The sum of costs of m -1 ops = The sum of costs of m ops
       The sum of the charges of m ops $\geq$ The sum of costs of m ops
       Thus, SI2 holds.
  - The m^{th} operation is reduce
    1. For a list of n elements, we have $2\alpha n$ charges stored in the list and diminishing the list has an actual cost of αn. Therefore, SI2 holds since the sum of the charges is greater than the sum of the costs.

2. Since we diminish the list in half, there are $\lfloor n/2 \rfloor$ elements and αn charges left. $\alpha n = 2\alpha * n/2 \geq 2\alpha * \lfloor n/2 \rfloor$ charges, therefore SI1 holds.

- Therefore, we can conclude that our state invariants are correct.

Part (c)

T(n) is the worst case of doing any sequence of n operations (over all possible sequences of n operations). If we were to have a sequence of n add operations, by the accounting method and our credit invariants, we would have the maximum charge of $2\alpha n$ in credit (by SI1). For a sequence of n operations (any combination of add or reduce), we can not get a credit higher than this since a reduce would have a charge of 0. We also know that the sum of charges $\geq$ sum of costs (by SI2), so the sum of costs of n operations is bounded above by $2\alpha n$. Therefore, we can say that $\mathrm{T(n)/n} \leq 2\alpha n/\mathrm{n} = 2\alpha$, which is constant. Thus, the amortized cost of an operation is constant.

Question 2 (written by Natalia, verified by Nena)

Part (a)

Let $n \in \mathbb{N}$ be the size of the array (i.e., the number of elements in S).

- **Search(x)**: The worst-case time complexity of `Search(x)` is $O(\log n)$ if x is not in S. We can achieve this by using binary search, which is feasible because A is sorted.
- **Insert(x)**: The worst-case time complexity of `Insert(x)` is $O(n)$. Finding the correct position for x can be achieved in $\log(n)$ time with binary search, but shifting elements in the array takes $O(n)$ time if the element needs to be inserted at the beginning.
- **Amortized Insertion Time**: The amortized insertion time, defined as the worst-case total time to execute a sequence of n `Insert` operations divided by n, is calculated as follows.

  Since S starts off empty, the first insertion takes 1 step. The second insertion takes 2 steps because we shift one element and insert the new one. The third insertion takes 3 steps, and so on. Thus, the total number of steps can be calculated as:

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n^2 + n}{2}$$

  Therefore, the amortized cost is:

$$\frac{1}{n}\left(\frac{n^2 + n}{2}\right) = \frac{n+1}{2}$$

  which is $O(n)$.

Part (b)

For two examples with different sets S:

- **i)** $S = \{4, 6, 2, 11, 7\}$

$$n = 5 = \langle 1, 0, 1 \rangle \Rightarrow \text{Two arrays: one of size } 2^1 = 1 \text{ and one of size } 2^2 = 4:$$

$$A_0 : \{4\}, \quad A_2 : \{2, 6, 7, 11\}$$

- **ii)** $S = \{16, 7, 2, 9, 0, 11, 5\}$

$$n = 7 = \langle 1, 1, 1 \rangle \Rightarrow \text{Three arrays of sizes 1, 2, and 4:}$$

$$A_0 : \{16\}, \quad A_1 : \{2, 7\}, \quad A_2 : \{0, 5, 9, 11\}$$

Part (c)

Search(x):

1. Starting from the first array, check the head of each array in L.
2. If x is smaller than the head of the array A_i, we know $x \notin A_i$, as each array is sorted in ascending order. Otherwise, we perform binary search on A_i.
3. If A_{i+1} doesn't exist, we return `False`.

Time Complexity:
- We check at most $k = \lceil \log_2(n) \rceil$ arrays and perform binary search in each. - The total number of steps is:

$$\sum_{i=0}^{k-1} i = \frac{(k-1)k}{2} = O(\log^2 n)$$

Thus, the complexity of `Search(x)` is $O(\log^2 n)$.

Part (d)

Insert(x):

1. Start by creating an array of length $1 = 2^0$, with x as its only element.
2. **Merging:**
   - If L has no A_0 (i.e., no array of length $2^0 = 1$), then the new array becomes L's A_0.
   - Otherwise, merge it with the existing A_0 to form A_1 (an array of length $2^1 = 2$), and continue similarly for higher orders. Each time two arrays of the same order are merged, they form an array of the next highest order.
3. Continue merging until reaching an order i such that $A_i \notin L$. Add the newly created array as A_i in L, thus updating L with this array.

Sorted Merging:

- Initialize two pointers: one at the beginning of each array being merged.
- Compare the elements at each pointer; the smallest gets added to a new array, and the pointer corresponding to that element is incremented to the next element in the array.
- Continue until all elements are added to the new array.

Time Complexity Analysis

- **Step 1** takes constant time as it simply creates a new array.
- **Steps 2 and 3** involve merging approximately $\log_2(n)$ arrays in the worst case, as this is the maximum number of pre-existing arrays that may need merging.
- Each merge operation between two arrays of size 2^i requires $O(2^i)$ comparisons since each element in both arrays must be moved to the new array.

Thus, the total time for a single insertion is:

$$O\left(\sum_{i=0}^{\log_2(n)-1} 2^{i+1}\right) = 2 \cdot O\left(\sum_{i=0}^{\log_2(n)-1} 2^i\right)$$

Since $\sum_{i=0}^{\log_2(n)-1} 2^i$ is a geometric series, we can simplify:

$$= 2 \cdot O\left(2^{\log_2(n)} - 1\right) = 2 \cdot O(n-1) = O(n)$$

Therefore, the worst-case time complexity for a single insertion is $O(n)$.

Part (e)

Amortized Complexity of Insert(x)
Accounting Method:

- We charge each insertion $O(\log_2(n))$ credits, since each insertion could potentially trigger merges at multiple levels up to $\log_2(n)$. This way, we are preparing for possible costs across all levels.
- Over n insertions, this results in a total of $O(n \log_2(n))$ credits, ensuring that we accumulate sufficient funds to pay for merges at every level when they occur.

- For each level i, arrays of size 2^i are merged at a cost of $O(2^{i+1})$ operations. Since merges at level i happen only once every 2^i insertions (when the current level reaches capacity and needs to merge upward), we distribute the cost of these merges across the necessary number of insertions.
- The $O(\log_2(n))$ charge per insertion ensures that each insertion contributes enough credits to cover merges across all levels. Specifically, by the time we need to merge at level i, we have accumulated exactly the 2^{i+1} credits required to cover the merge cost at that level.

Thus, the total credits accumulated after n insertions is $O(n \log_2(n))$. Dividing by n, we find that the amortized cost per insertion is $O(\log_2(n))$, as required.

Aggregate Analysis:

- When we insert n elements, we'll have gone through each level of merging up to $\log_2(n)$ times.
- Each level/order i costs $O(2^{i+1})$, but it only occurs $O(\frac{n}{2^i})$ times across all n insertions.
- This is because to create an array at level i, we need 2^i elements. Each time we get 2^i elements, a merge is triggered at level i because we reach the capacity of that level. This will occur approximately $\frac{n}{2^i}$ times.

Thus, the total cost for all n insertions can be approximated by:

$$\sum_{i=0}^{\log_2(n)-1} \left(\frac{n}{2^i} \cdot 2^{i+1}\right) = \sum_{i=0}^{\log_2(n)-1} (2 \cdot n) = 2n \cdot \log_2(n) \sim O(n \log_2(n))$$

Thus on average, each insertion takes:

$$\frac{O(n \log_2(n))}{n} = O(\log_2(n))$$

Part (f)

Delete(x)

1. **Locate** x using `Search(x)`:
   - Perform `Search(x)` to find the array A_i containing x.
   - `Search(x)` has a time complexity of $O(\log^2 n)$.
   - If `Search(x)` returns `False`, x is not in L, and the delete operation terminates.
2. **Remove** x **from the Array**:
   - Once x is found, remove it from A_i.
   - After removal, the array size is now $2^i - 1$, which is not a power of 2.
3. **Handle the Empty Slot**:
   - The array no longer satisfies the required power-of-2 size, so we cannot leave an empty slot in place.
   - To restore the structure, we proceed to split this array into smaller arrays of sizes that are powers of 2.
4. **Divide Elements into New Arrays**:
   - Rather than splitting the existing array in place, we distribute its $2^i - 1$ elements into newly created arrays of sizes that are powers of 2.

- For example, an array of size 7 would be divided into new arrays of sizes 4, 2, and 1, containing the elements of the original array.
- The runtime for this step is $O(2^i)$, which is $O(n)$ in the worst case when the largest array is involved.

5. **Handle Duplicate Sizes with Cascading Merges**:
   - For each newly created array, check if an array of the same size already exists in L.
   - If a duplicate array size is found, merge the two arrays to form an array of the next higher power of 2.
   - Repeat this merging process (cascading) until all resulting arrays have unique power-of-2 sizes and can be added back to L without duplicates.
   - The cascading merges take $O(n)$ time in the worst case because each element is involved in at most one merge per level. The total merge cost across all levels is given by:

$$\sum_{i=0}^{\log n} O(2^i) = O(n)$$

6. **Insert Resulting Arrays into** L:
   - Add the final set of unique power-of-2-sized arrays back into L, ensuring that each level in L has only one array of each power-of-2 size.

Overall Time Complexity: The total time complexity for `Delete(x)` is $O(n)$.

Question 3 (written by Nena, verified by Natalia)

A brief description of the algorithm

- We define G as an adjacency list representing the vertices and edges in the graph. The color can be white (not discovered), grey (discovered but not explored), and black (discovered and explored). The distance d is the distance from the nearest hospital. And the parent p is the node that discovered the current node.
- We will use the BFS algorithm to discover and explore all nodes to find the distance from the nearest hospital.

Algorithm 1 BFS(G):

```
1: Q ← empty
2: for each v ∈ V do
3:     if v ∈ hospitals then
4:         color[v] ← grey
5:         d[v] ← 0
6:         p[v] ← NIL
7:         ENQ(Q, h)
8:     else
9:         color[v] ← white
10:        d[v] ← ∞
11:        p[v] ← NIL
12:    end if
13: end for
14: while Q is not empty do
15:    u ← DEQ(Q)
16:    for each (u, v) ∈ E - H do
17:        if color[v] == white then
18:            color[v] ← grey
19:            d[v] ← d[u] + 1
20:            p[v] ← p[u]
21:            ENQ(Q, v)
22:        end if
23:    end for
24: end while
25: color[u] ← black
```

- The reason why this algorithm is correct is because, high level, we are searching for houses that are distance 0, 1, 2... from hospitals in increasing order.
- **Predicate:** After processing all nodes at depth x, the queue will contain only nodes at depth x + 1 (distance of x + 1 from the nearest hospital).
- **Base Case:** Initially, we go through each vertex in the graph. For hospitals, we set their distance to 0, color them grey, and enqueue them, as they are the starting points for the BFS. For houses, we set the distance to infinity (unknown), parent to NIL, and color them white. The predicate holds at the start since the queue only contains hospitals, which are at distance 0.
- **Induction Step:** We assume that all the nodes in the queue have a d = x value that is the closest distance to a hospital [IH]. Note that the shortest path theorem still holds with multiple sources, being the multiple hospitals we enqueued in the beginning. If it is equally close to two hospitals, then it will depend on the order which the nodes are enqueued, but the min distance will not be affected. When

we process these nodes, we explore their neighbors. For each white neighbor, we set its distance to x + 1, mark it grey, and enqueue it. Therefore, once we explore all nodes that have a d = x, we will only have nodes that have a d = x + 1 that are grey nodes, as previously grey and black nodes were not regarded. Therefore, the predicate holds.

- We know that since to initialize each vertex to a color, distance, and parent takes constant time (this is the for loop) and there are $|V|$ vertices, O($\|V\|$). We also know that the while loop at most traverses through all the edges (with BFS it is possible to have unreachable nodes and edges), so this is O($\|E\|$). Therefore, BFS has a runtime of O($\|V\| + \|E\|$).