CSC263H1 Assignment 5

Jiatao Xiang, Xu Wang, Huakun Shen

March 14th, 2019

Question 1

a. *Written by Huakun Shen, checked by Xu Wang, Jiatao Xiang*

$\boldsymbol{I_1} = \{6, 8, 4, 13, 9\}$

$\boldsymbol{L_1} \rightarrow \boldsymbol{A_0} : [6] \leftrightarrow \boldsymbol{A_2} : [4, 8, 9, 13]$

$\boldsymbol{I_2} = \{21, 12, 7, 14, 5, 16, 10\}$

$\boldsymbol{L_2} \rightarrow \boldsymbol{A_0} : [21] \leftrightarrow \boldsymbol{A_1} : [7, 12] \leftrightarrow \boldsymbol{A_2} : [5, 10, 14, 16]$

b. *Written by Huakun Shen, checked by Xu Wang, Jiatao Xiang*

Traverse through L, for each array A, if the first element of $A > x$, then x is not in A, because A is in ascending order. If the first element of $A \leq x$, then x may be in A. In this case, binary search x in A. If x is found, return $\boldsymbol{TRUE}$. If x is not found, continue to next array.

```
def Search(x):
    for i = 0...len(L)-1:
        if Ai[0]<=x:
            BinarySearch x in Ai:
                if x is found:
                    return True
    return False
```

Suppose L has n elements, $len(L)$ is the length of the linked list, $len(L)$ is at most $(\lfloor \log_2 n \rfloor + 1)$.

$$\begin{aligned} RT &= \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} i \\ &= \frac{\lfloor \log_2 n \rfloor \cdot (\lfloor \log_2 n \rfloor + 1)}{2} \\ &= \mathcal{O}(\log(n)^2) \end{aligned}$$

The worst-case time complexity of $Search(x)$ is $\mathcal{O}(\log(n)^2)$.

c. *Written by Huakun Shen, checked by Xu Wang, Jiatao Xiang*

Insert x as an array of size 1, and merge this array with L. Call the array A.

Consider L as a binary representation of n, where n is the total number of element in L. 0 means there does not exist an array A_i at index i, 1 means there exists an array A_i at index i with a size of 2^i.

We want to insert the array A in the position of the first occurence of 0 in binary representation. Before that, whenever a 1 is encountered (i.e. array A and the array represented by the "1" have the same size) merge the two arrays and carry on with the merged array until a 0 is encountered.

```
def Insert(x):
    for i = 0...len(L) - 1:
        if Ai is not in L:          # Binary representation of bit 0
            Ai <- A                 # Put A in i th position
        else:
            A <- Merge(A, Ai)
```

Suppose L has n elements, $len(L)$ is the length of the linked list, $len(L)$ is at most $(\lfloor \log_2 n \rfloor + 1)$.

$$\begin{aligned} RT &= \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} 2^{i+1} \\ &= 2 \cdot \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} 2^i \\ &= 2 \cdot \frac{1 - 2^{\lfloor \log_2 n \rfloor + 1}}{1 - 2} \\ &= 2^{\lfloor \log_2 n \rfloor} \cdot 2^2 - 2 \\ &\leq 4n - 2 \\ &= \mathcal{O}(n) \end{aligned}$$

The worst-case time complexity of $Insert(x)$ is $\mathcal{O}(n)$.

d. *Written by Huakun Shen, Jiatao Xiang, checked by Xu Wang*

The Amortized time per insert in a sequence of n inserts is $\mathcal{O}(\log(n))$

Aggregate Method:

Let $T(n)$ denote the total runtime of a sequence of n inserts.

For each merge of 2 A_i, i.e. merging 2 arrays of size 2^i costs at most $(2^{i+1} - 1)$ comparisons.

In a sequence of n inserts, the number of merges occuring between 2 A_i arrays, i.e. 2 arrays of size 2^i is $\lfloor \frac{n}{2^{i+1}} \rfloor$.

For example, in a sequence of 16 inserts, the number of merges occuring between 2 A_0 arrays is $\lfloor \frac{16}{2^{0+1}} \rfloor = 8$, and 4 merges for A_1, 2 merges for A_2, 1 merge for A_3.

$$\begin{aligned} T(n) &= \left\lfloor \frac{n}{2^1} \right\rfloor \cdot (2^1 - 1) + \left\lfloor \frac{n}{2^2} \right\rfloor \cdot (2^2 - 1) + \left\lfloor \frac{n}{2^{i+1}} \right\rfloor \cdot (2^{i+1} - 1) \\ &= \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^{i+1}} \right\rfloor \cdot (2^{i+1} - 1) \\ &= \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^{i+1}} \right\rfloor \cdot 2^{i+1} \\ &= \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} \frac{n}{2^{i+1}} \cdot 2^{i+1} \\ &= \Sigma_{i=0}^{\lfloor \log_2 n \rfloor} n \\ &= n \cdot \lfloor \log_2 n \rfloor \\ &= \mathcal{O}(n \cdot \log(n)) \end{aligned}$$

$$\begin{aligned} \text{Amortized runtime per insert} &= \frac{T(n)}{n} \\ &= \frac{n \cdot \log(n)}{n} \\ &= \mathcal{O}(\log(n)) \end{aligned}$$

Accounting Method:

Given a sequence of n inserts, we know that the length of L is at most $(\lfloor \log_2 n \rfloor + 1)$.

For each of the insert, we assign $(\lceil 1 + 4 \cdot \log_2 i + \log_2 n \rceil)$ credits, where i is the length of L at the moment the element is inserted. The 1 is the cost of inserting an element into an array of size 1. $2 \cdot \log_2 i$ is for unlinking and linking arrays while merging. $\log_2 n$ is the number of comparisons an element could experience. Detailed explanation is as follows.

Facts:

- For a merge between 2 arrays of size x, the number of comparisons is at most $2x$, which means every element in the 2 arrays could experience at most 1 comparison.
- A newly inserted element starts in an array (A_0) of size 1 and merge as long as there's a new array of the same size.
- After each merge, a larger array (size doubles) would form and all elements will be in this array.

After n inserts, there will be at most $(\lfloor \log_2 n \rfloor + 1)$ arrays in L, and the number of elements in the largest array is $2^{\lfloor \log_2 n \rfloor}$. Since every merge between 2 arrays forms a new array of double the size, if we think backwards, an array of size $2^{\lfloor \log_2 n \rfloor}$ must have experienced $\lfloor \log_2 n \rfloor$ merges, and every elements in the largest array must have experienced $\lfloor \log_2 n \rfloor$ comparisons.
For elements in the smaller arrays (if exists), they must have experienced less than $\lfloor \log_2 n \rfloor$ merges (or $\lfloor \log_2 n \rfloor$ comparisons).
In brief, after a sequence of n inserts, every element in L must have experienced at most $\lfloor \log_2 n \rfloor$ comparisons. Thus, assigning $\log_2 n$ credits for merging to each inserted element is sufficient.

While merging, the links on the arrays to be merged would be broken (since we need to form a new array), and relink the the new array to L. We need to consider the cost of linking.
Fact:

- Breaking or building a link between 2 nodes in a doubly linked list takes 2 steps (as long as we keep track of the position of the nodes).
- While L has a size of i (i elements in L), the length (number of arrays) in L is at most $\lfloor \log_2 i \rfloor + 1$, with at most $\lfloor \log_2 i \rfloor$ links.
- Each merge breaks a link, form a new array, and link the new array to L.

For each insert, there could be at most $\lfloor \log_2 i \rfloor$ unlinkings and at most $\lfloor \log_2 i \rfloor$ linkings, which takes at most $2 \cdot 2 \cdot \log_2 i = 4 \cdot \log_2 i$ steps.

In conclusion, assigning $(\lceil 1 + 4 \cdot \log_2 i + \log_2 n \rceil)$ credits to each insert is sufficient. Since i is the real-time size of L while inserting which $i \leq n$ all the time, $\log_2 i \leq \log_2 n$ and $\lceil 1 + 4 \cdot \log_2 i + \log_2 n \rceil = \mathcal{O}(\log_2 n)$. We can say that the amortized runtime per insert in a sequence of n inserts is $\mathcal{O}(\log_2 n)$.

e. *Written by Jiatao Xiang, checked by Xu Wang, Huakun Shen*

```
def Delete(x):
    Let Aj be the samllest list in L.
    if x in the Aj:
        Aj.remove(x)
        if len(Aj) != 0: #then it must be (power of 2) - 1
            Then use a loop to split Aj into Aj-1...A0 and connect to L.
        else: unlink Aj from L
    else if x is not in Aj:
        say x is in Aj+i
        switch x with y, where y is the the first element of Aj
        sort Aj+i   #we only need to move the y into proper place
        Aj.remove(x)
        if len(Aj) != 0:    #then it must be (power of 2) - 1
            Then use a loop to split Aj to Aj-1...A0 and connect to L.
        else: unlink Aj from L
end algorithm
```

The worst case is that we need to swap x with the first element y of the smallest list A_j, and then sort the list A_{j+i} with $\mathcal{O}(n)$ (because only 1 element is not in the correct position in the array, swapping it to the right position doesn't cost higher than traversing the entire array), then remove x with constant time, finally use $\mathcal{O}(n)$ to split the smallest set A_j to $A_{j-1}...A_0$ and use $\mathcal{O}(log(n))$ to connect to the linked list L (There will be at most log(n) newly-generated arrays of integers resultnig from split). Thus, the total cost of delete is $\mathcal{O}(n + n + \log n) = \mathcal{O}(n)$.

Question 2

a. *Written by Huakun Shen, checked by Xu Wang, Jiatao Xiang*
For each house, use BFS to construct a BFS Tree. Then in each BFS Tree, each hospital node contains

the length of the shortest path to the house. Then, we find the min of them, which is $\mathcal{O}(|V|)$.
Each BFS costs $\mathcal{O}(|V| + |E|)$. Since the number of house is constant, say $c \in \mathbb{N}$. Then the total runtime of all BFS's is $\mathcal{O}(c(2|V| + |E|)) = \boldsymbol{\mathcal{O}(|V| + |E|)}$

b. *Written by Huakun Shen, checked by Xu Wang, Jiatao Xiang*
Do BFS for each hospital. Given a house, in each BFS tree, the house node contains the the length of the shortest path to the root hospital. Since there are k hospitals, there are k shortest path from a house to k different root hospitals. Then find the minimum length of path among all these shortest paths.
Since each BFS costs $\mathcal{O}(|V|+|E|)$ of time, we do BFS for each hospital, then the runtime for constructing k BFS trees is $\mathcal{O}(k(|V| + |E|))$. Min operation costs $\mathcal{O}(k)$ of time.
Thus, solving problem P costs $\mathcal{O}(k(|V| + |E|) + k) = \boldsymbol{\mathcal{O}(k(|V| + |E|))}$ of time.

c. *Written by Jiatao Xiang, checked by Xu Wang, Huakun Shen* We will solve this problem with adjacency list of graph G. We loop through the head of adjacency list to find all the hospitals, assign distance to 0 and add all the hospitals to the Queue which cost $\mathcal{O}(|V|)$, then we use a similar algorithm as BFS does. Use a while loop to loop over each one hospital at a time and only explore the node adjacent to them. For example, the first hospital find all the its adjacent house(not including hospital), if the color of house is white, add it to the queue, then assign the house's distance with distance of parent + 1, and change its color to grey. If the color is already grey, which means it has already been discovered by its closest hospital, then we will not discover it any more. Note: We find that when a node is dequeued from the queue and explore its adjacent nodes, it becomes fully explored and we change it color to black. Similarly, we use while loop do same thing on nodes from the queue. And the while loop will ends when all the node become black which means they all finds the closest hospital and this will cost the same as BFS does which is $\mathcal{O}(|V| + |E|)$, thus the total cost is $\mathcal{O}(2|V| + |E|)$ which is also $\mathcal{O}(|V| + |E|)$.