Assignment 5 Q2

Wrote: Jingbo Yang / Read: Muzi Zhao

(a)

(i) Array A is sorted, so when executing a `Search(x)` we can use binary search to determine if x can be found. Each time we cut the array in half, and compare x to the pivot to decide which half x should go to for the next search. There are at most $O(\log n)$ comparisons if n is the length of A. In the worst case, we can't find x, so we still need $\Omega(\log n)$ comparisons. Therefore, the worst-case time complexity is $\Theta(\log n)$.
(ii) Finding the position of x may take $\Theta(\log n)$ as described in (i). But we need to move all elements after x if x is inserted, which would take $\Theta(n)$. So the total time complexity is $\Theta(n)$.
(iii) Inserting n elements into an empty array to make it a sorted array would take $\Theta(n^2)$, since according to (ii) we have: $1 + 2 + 3 + \cdots + n = (n^2)$. So the amortized insertion time:

$$\frac{\Theta(n^2)}{n} = \Theta(n).$$

(b)

For $S = \{4, 6, 2, 11, 7\}$ with $n = 5 = \langle 101 \rangle$, we have $A_0 = \{4\}$ and $A_2 = \{2, 6, 7, 11\}$. Then L: $\{4\} \leftrightarrow \{2, 6, 7, 11\}$; for $S = \{16, 7, 2, 9, 0, 11, 5\}$ with $n = 7 = \langle 111 \rangle$, we have $A_0 = \{16\}$, $A_1 = \{2, 7\}$, $A_3 = \{0, 5, 9, 11\}$. Then L: $\{16\} \leftrightarrow \{2, 7\} \leftrightarrow \{0, 5, 9, 11\}$.

(c)

We traverse the linked list L. For each A_i in L, perform binary search. If x is found, return True; else return False. For each array in L with 2^i length, it would take $O(\log 2^i)$, $0 \le i \le k-1$. Then, in the worst case we loop over all arrays:

$$\sum_{i=0}^{k-1} \log 2^i = \log 2^0 + \log 2^1 + \cdots + \log 2^{k-1} \le (k-1)\log 2^{k-1} \le \log n \cdot \log n = O(\log^2 n).$$

(d)

We first use a new array to store x, denoted $A' = \{x\}$ and put this array at the head of L. Traverse L from head; while there are two arrays of the same size, merge them using the merge part of mergesort. In the worst case, $n = b_{k-1}b_{k-2}\ldots b_0 = \langle 111\ldots 1 \rangle$. So we need to merge k arrays. Using the merge part from mergesort:

$$2 \cdot 2^0 + 2 \cdot 2^1 + \cdots + 2 \cdot 2^{k-1} = 2(2^k - 2) = 2(2^{\log n} - 2) = O(n).$$

(e) Aggregate Analysis

Let k be the position of the first 0 in the binary representation. Merging two arrays will take time of the length of the longest array. Starting with an empty set S and inserting n elements, the process follows this pattern:

- When $k = 0$, it means the insertion only needs to affect array A_0. Since A_0 has a size of 1, each new element is simply placed into A_0. When A_0 becomes full, it merges with the next array (like A_1), similar to the carry operation in binary counting. Thus, over the course of inserting n elements, position $k = 0$ will trigger approximately $n/2$ merges.
- When $k = 1$, i.e., the second position where the binary representation has a 0, approximately every 4 insertions will produce a carry to A_1, leading to about $n/4$ such merges over the entire sequence.
- Similarly, when $k = j$, the j-th position has a 0 for the first time, resulting in approximately $n/2^{j+1}$ merges.

In general, the total merging cost can be expressed as:

$$2^0 \cdot \frac{n}{2} + 2^1 \cdot \frac{n}{4} + 2^2 \cdot \frac{n}{8} + \cdots$$

This is a geometric series, and its sum can be simplified to:

$$\sum_{j=0}^{\log n} 2^j \cdot \frac{n}{2^{j+1}} = \frac{n}{2} \sum_{j=0}^{\log n} 1 = O(n \log n)$$

Therefore, the amortized cost per insertion is:

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

(e) Accounting Method

Assume each time we insert an element, we charge it with $\log n$ credit. When we insert elements one by one, elements already in L might be moved to other arrays, which costs 1 for each movement per element. Since each time an element is moved to an array that is larger than the original (e.g., if the element is in 2^i, it would be moved to 2^{i+1}), therefore the total number of movements $\leq \log n$. Thus, the credit for each element - cost of moving $\geq \log n - \log n = 0$.

Amortized cost:

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

(f) Delete(x)

Let A_m be the smallest array in the doubly-linked list L. Use Search(x) to find its position. There are two situations: if x is in A_m, we remove x from A_m. At this point, there are $2^m - 1$ elements left in A_m. Notice that $2^m - 1 = 2^0 + 2^1 + \cdots + 2^{m-1}$. Since A_m is the smallest array, we know $b_{m-1}, b_{m-2}, \ldots, b_0$ are all zero. So we can put 2^0 elements to A_0, 2^1 elements to A_1, $\ldots$, 2^{m-1} elements to A_{m-1}, and remember to maintain the order in each new array. Then insert these new arrays $(A_0, \ldots, A_{m-1})$ into the doubly-linked list L, and remove A_m.

If x is not in A_m, let's say x is in A_t and $t > m$. First remove x from A_t, and get the first element in A_m, insert it into A_t using binary search. Thus $|A_t| = 2^t$ still holds. For A_m, there are $2^m - 1$ elements left. Like discussed in the first case, divide it into $A_0 \sim A_{m-1}$ arrays, put them into L and delete A_m.

Time Complexity: search(x) would take $O(\log^2 n)$. Removing x would take $O(2^m)$ or $O(2^t)$, assigning left elements into new arrays would take $O(2^m - 1)$ since we assign them one by one. If we need to insert the first element from A_m into A_t, it would take $O(n)$ according to part (d). Updating doubly-linked list would take $O(\log n)$. So in the worst case, the time complexity is:

$$O(\log^2 n) + O(2^m) + O(2^t) + O(2^m - 1) + O(n) + O(\log n) = O(n) \quad \text{(Since } 2^m \leq 2^t \leq \log n\text{)}$$