Midterm 2020-2021 Sem 1

Brendan Cheong Ern Jie

12/02/2022

1 Question 01

(a) Code 1a)
[1024]

(b) Code 1b)
[here]

(c) Code 1c)
[3.4]

(d) Code 1d)
[This is just bubble sort Ans: 0024CShiisst]

(e) Code 1e)
[sorting this phrase is just a,b,c,d .. z. So using lower bound is asking us index of f to first letter of sorted phrase which is 0 to 6 which is 7]

2 Question 02

(a) O(N $\log(N)$)

(b) O(1)

(c) O(N)

(d) O(N^2)

(e) O($N \log(N)$)

3 Question 03

(a) 6! = 720 permuations

(b) 1 permutation

(c) No, using merge sort defats the purpose of using radix sort to achieve O($n + \frac{d}{k}$) time as it will always stick to O($N \log(N)$) time instead. Moreover, radix sort is already stable, so the additional merge sort really does nothing at all but makes the algorithm slower.

(d) I am mergeSort. No, other frequently used sorting algorithm like quicksort are used as well and have similar time complexities of O($N \log(N)$. Other sorting algorithms are used as we do not always need a stable sort like merge sort)

4 Question 04

(a) The fastest you can go is O(N). This is because you would have to iterate through the array to find each and every element in the single linked list and see if there are duplicates by adding them to a set. This takes O(N) time. When adding to set, you can check if the set already contains the element using set.count(i) where i is the element in O(log N) time. Thus, the time complexity would be O(N).

(b) Nothing will happen to the SLL implementation, as the element that is deleted has no other vertex pointing to it. Perhaps this will cause heap pollution and an eventual segmentation fault.

(c) The proper data structure would be std::deque, as it already has operations for pushing to the top of the stack in the form of push_front() and removing from the front in the form of pop_front() as well as checking the front with front(). A fixed size array wouldn't work as the size of the stack changes with the number of elements place in it. Thus, std::deque supports the stack's Last In First Out data structure.

(d) The proper data structure would be fixed-size array, as the fixed size Array can fix its size and capacity at C, while the other data structures size would increase the more elements you place inside it

(e) HARD!!!!

[Honestly, I would just implement 2 queues, one for strings and one for integers using std::queue. Then I would keep 2 boolean statements called int_first, string_first, depending on which was the most recently added element, like for say if the element was a int, then int_first would be true and string_first would be false, and vice versa if the added element was a string. Depending on the element type, it would be added to its respective queues based on its type. If dequeueString() is called, it will just call queue.pop_front() from the string type queue, and if dequeueInteger() is called, it will call queue.pop_front() fron the integer queue. If fornt() is called, it will peek() at the front of the queue depending on the most recent type based on the boolean statements.]

5 Question 05

(a) HARD!!!

[-1, 5, 3, 2, 1, 4], so when 4 is immediately promoted to root after extracting 5, it can't go down

(b) I would first use ExtractMax().Remove(v) to check if the element exists or not. If it does not, IncreaseKey will not work. If it does exist, store the extracted max value in a variable call max_variable. Now, insert the key using Insert(v + delta_v). After that, add back the previously extracted max value with Insert(max_variable). Note that we can't just use Remove(v) as it does not report back NULL if v does not exist.

6 Question 06

(a) HARD!!!

1. create binary min heap from a[1..K] → O(K)
2. for i = K+1, ..., N, extract min from that heap and insert a[i] to the heap → O((N-K)logK)
3. the heap now contains the K largest elements, just perform heapsort → O(K log K) to sort the elements in the heap

This algorithm's time complexity is in $O(N \log(K))$

(b) HARD!!! For each iteration of N, from i = 0 to N,
1. create a binary min heap from i = 0 to j = i + k, where j can loop back round if j is ¿ N where j =

N - j, do this from [i..N]. This takes up O(K) times to create the heap
2. extract the minimum from this heap of size K in $O(\log(K))$ time and store it in an int called max_int
3. If the minimum from the heap is bigger than max_int, replace max_int with the new number. repeat this K times until the heap is empty, this takes up $O(\log(K))$
4. repeat this is N times, thus the overall time complexity is $O(NK + N\log(K))$ which translates to $\rightarrow O(N\log(K))$