Ivy Truong

3/17/20

CSCI 3412

Homework 4

1. a) My algorithm is a modified version of a merge sort. Below is the pseudocode, given that n is the total length of the given array, and k is the number of sorted lists to be merged. n is represented as len(array), while k is the value that will be received from the user. Please look at part 1b for explanation for why the algorithm meets O(nlogk) time efficiency.

#calculate the length of the sublist that mergesort will stop dividing

kLen =

MergeSort(array, kLen):

#if it reaches the set array length, return so it can sort the array

If len(array) <= kLen:

Return array

If len(array) > kLen:

#keep splitting the array in half into two sublists

Mid = int(len(array)/2)

Left = array[0:mid]

Right = array[mid:len(array)]

Left = MergeSort(Left, kLen)

Right = MergeSort(Right, kLen)

#after splitting the lists, sort the sublists using any sorting method

#after dividing the list accordingly, merge the lists together in a new array

#make a new array

newArr = []

#check the smallest values between the two lists and move the smaller value into the new array

i = j = 0

while True:

if Left[i] < Right[j]:

newArr.append(Left[i])

i += 1

elif Left[i] ≥ Right[j]:

newArr.append(Right[j])

j += 1

#if there are any left over elements in the two sublists, then append the rest of the elements to the new array

if i < len(Left):

#append rest of elements in left sublist

#break out of loop

elif j < len(Right):

#append rest of elements in right sublist

#break out of loop

Return newArr #from merging the sublists together into one

1. b) My algorithm meets O(n log k). Given that there are k nearly sorted lists, it would take log(k) because the sort continually divides the original list in half until there are k sublists, which means that the sort would only have to sort k lists rather than n lists. To merge the sorted lists together in a single sorted array, it would take the number of lists (k) multiplied by the length of each list, which is the total number of values (n) divided by the number of sublists (k), to get O(k \* ) = O(n). The overall time efficiency of the algorithm would be O(logk) \* O(n) = O(nlogk).

1. c) Python code on a different file

-algorithm took 142.5 seconds to sort 1000000 random integers.

2. Comparison among lists

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Unsorted, singly linked list | Sorted, singly linked list | Unsorted, doubly linked list | Sorted, doubly linked list |
| Search (L, k) | O(n) | O(n) | O(n) | O(n) |
| Insert (L, x) | O(1) | O(n) | O(1) | O(n) |
| Delete (L, x) | O(n) | O(n) | O(1) | O(1) |
| Successor (L, x) | O(1) | O(1) | O(1) | O(1) |
| Predecessor (L, x) | O(n) | O(n) | O(1) | O(1) |
| Minimum (L) | O(n) | O(1) | O(n) | O(1) |
| Maximum (L) | O(n) | O(n), should be O(1) | O(n) | O(n), should be O(1) |

-k = key

-x = value/pointer to object

-if it is not O(n), explain why (ones highlighted in yellow do not run in O(n))

-red means wrong

a) Insert: The time complexity of the insert function for both unsorted singly linked lists and doubly linked lists is O(1) because the value can be inserted at the front of the list. Since the list is not sorted, the position of the newly inserted value does not matter, so the quickest way to insert a value in the unsorted list is to add it to the front of the list.

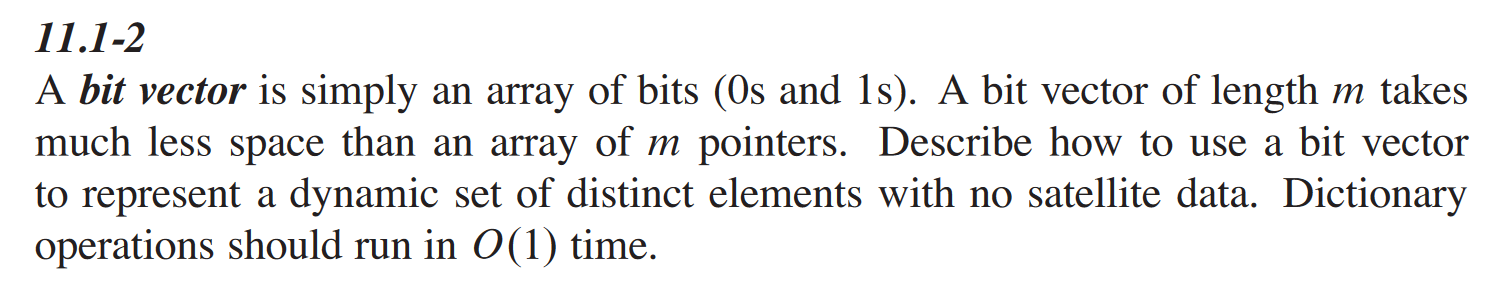
b) Delete: The time complexity of the delete function for both unsorted and sorted doubly linked lists is O(1) because the value can be deleted right away after it is found. It only takes O(1) to delete the element right away, but if it wanted to delete an object with a specific key, then it would take O(n) to search for that object. The main reason is that doubly linked lists have a pointer that points to the next element and the previous element in the list, so there is no need to keep track of the previous element. For example, say that the doubly linked list is A,B,C and we want to delete B. Since it is a doubly linked list, we can let the next pointer of A point to C and the previous pointer of C point to A automatically and then delete B. In singly linked lists, it takes O(n) at most to search for the element (with a specific key) and up to O(n) to look for the previous element so that when deleting the chosen element, part of the linked list is not lost. We have to look for the previous element of the chosen element to delete in order to connect the two parts of the list after deleting the chosen element. For example, lets say the list A,B,C is a singly linked list and we want to delete B. We would need another tracker pointer to find the previous element of B (which is A) since B doesn’t have a previous pointer that points to A, only a pointer that points to C. We would need that tracker pointer to point the next pointer of A to element C so that part of the singly linked list is not lost.

c) Successor: Both singly linked lists and doubly linked lists can access the next element at O(1) because they have a designated pointer that points to the next element in order to connect the list.

d) Predecessor: Both unsorted and sorted doubly linked lists can access the previous element at O(1) because they have a designated pointer that points to the previous element in the list.

e) Minimum: Both sorted singly linked lists and doubly linked lists can access the minimum element at O(1) because since it is sorted in ascending order, the smallest element would be at the beginning of the list and the minimum can be accessed right away.

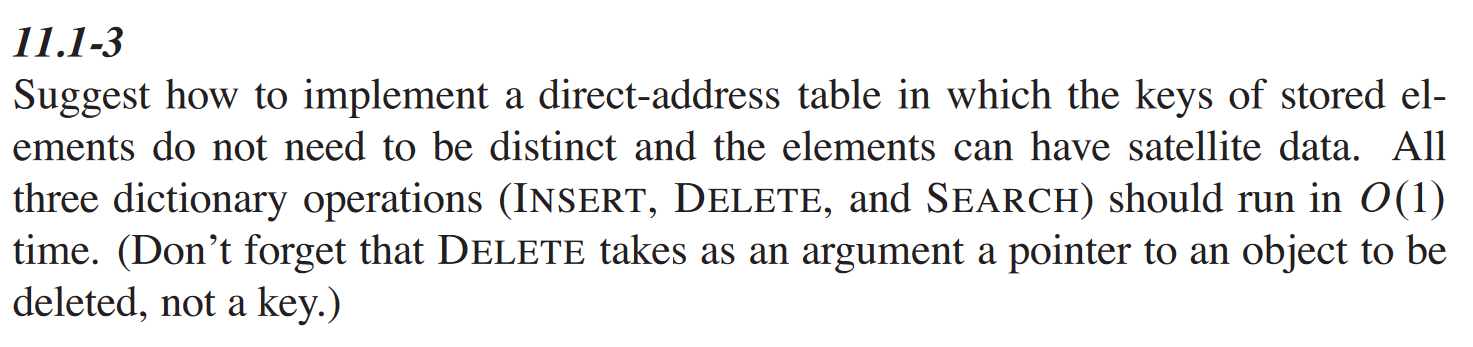
3.



A bit vector can be used to represent a set of data that fit in two options, that it fits the characteristic or it doesn’t (like True/False scenario). For example, the bits 0 and 1 can represent a set of values where a number is prime or not, where 0 means it is not prime and 1 means it is prime.

To represent a distinct set of elements with no satellite data, we can use bit vector to map the values in the set. If the value is in the set, mark as a 1, and 0 if it is not in a set. To represent a set in bit vector format, we would need to create a bit vector of size n+1, where n is the largest element we need store in the set. If given 8 bits (1 byte) in the bit vector and the set is {1, 2, 3, 6, 7}, then the bit vector would be 0111 0011, where bits 1, 2, 3, 6 and 7 are marked as a 1 to show that they are values that are in the set (first bit represents 0 because counting starts at 0). In this case, the bit vector would act similarly to an array, where we use the placement of the bits to represent the values in the set. All dictionary operations would be able to run in O(1) time because access to the set values are similar to that of an array, where you access by index in an array, and the bit placement in the bit vector.

4.



To implement a direct-address table with non-distinct keys and elements with satellite data, we would create a hash table with separate chaining. In the hash table, there will be all the keys and if the key is not used, it will be set to NULL. Since the keys don’t have to be unique, the keys can have a doubly linked list of elements that share the same key connecting to the hash table. For example, if two elements hash to key 1, then there can be a linked list of the two elements, where the head pointer that points to the start of the list is the key 1.

The insert operation will run in O(1) because if another element hashes to the same key, the element can be added to the front of the linked list for the key it hashes to. The delete operation will run O(1) because it can delete right away after it has found the element in the list. Since the list is a doubly linked list, where an element in the list would have a previous pointer pointing to the previous element in the list and a next pointer pointing to the next element in the list, it is easier to connect the list after deleting the chosen element without having to traverse through the list again to find the previous element. The search operation will run in O(1) time because it would return the list of elements that correspond to the key in the hash table.

Extra Credit: Web scraping and data science

1. Yes

2.

Source 1: https://www.youtube.com/watch?v=XQgXKtPSzUI

Review 1: This video taught me how to web scrape using a package called Beautiful Soup. I thought it was interesting to see that we can use this application to grab different parts of the webpage that we want and make a database out of it. In the tutorial he uses Beautiful Soup to grab specific HTML tags or a set of HTML tags to obtain either an image, a title, or a description. The one thing that might be difficult for new people learning about web scraping using this tutorial is that one must know how to read HTML to know what kinds of information they want to grab, so one must have prior knowledge of HTML before watching this tutorial. Overall, this tutorial was really helpful because it went step by step on the different functions that can be used with Beautiful Soup to grab certain sets of information from a webpage.

Source 2: https://realpython.com/web-scraping-and-crawling-with-scrapy-and-mongodb/

Review 2: This article talked about how to web scraping in a larger scale using scrapy to web scrape questions and links from a webpage and store the results. It was cool to note that there were other applications that supported a larger scale of web scraping data.

3. Python code and JSON file on separate files