CS101 Algorithms and Data Structures

Fall 2019

Homework 9

Due date: 23:59, November 24, 2019

1. Please write your solutions in English.
2. Submit your solutions to gradescope.com.
3. Set your FULL Name to your Chinese name and your STUDENT ID correctly in Account Settings.
4. If you want to submit a handwritten version, scan it clearly. Camscanner is recommended.
5. When submitting, match your solutions to the according problem numbers correctly.
6. No late submission will be accepted.
7. Violations to any of above may result in zero score.

Problem 1: Hash Collisions - Direct Addressing

Consider a hash table consisting of $m = 11$ slots, and suppose we want to insert integer keys $A = [43, 23, 1, 0, 15, 31, 4, 7, 11, 3]$ orderly into the table.

First, let's suppose the hash function h_1 is:

$$h_1(k) = (11k + 4) \mod 10$$

Collisions should be resolved via chaining. If there exists collision when inserting a key, the inserted key(collision) is stored at the end of a chain.

(1) In the table below is the hash table implemented by array. Draw a picture of the hash table after all keys have been inserted. If there exists a chain, please use $\rightarrow$ to represent a link between two keys.

Index	**Keys**
0	
1	7
2	
3	
4	0
5	1→31→11
6	
7	43→23→3
8	4
9	15
10	

(2) What is the load factor λ?

$\lambda = \frac{10}{11}$

(3) Suppose the hash function is modified into

$$h_2(k) = ((11k + 4) \mod c) \mod 10$$

for some positive integer c. Find the smallest value of c such that no collisions occur when inserting the keys from A. Show you steps in detail and draw a picture of the hash table after all keys have been inserted in the table below. If there exists a chain, please use $\rightarrow$ to represent a link between two keys.

Index	**Keys**
0	23
1	15
2	7
3	43
4	0
5	1
6	11
7	3
8	4
9	31
10	

Steps of finding c:

```
A = [43,23,1,0,15,31,4,7,11,3]
for c in range(10,1000):
    H = [[] for _ in range (11)]
    for k in A:
        h = ((11*k+4)%c)%10
        H[h].append(k)
    if max([len(L) for L in H]) == 1:
        print(c)
        print(H)
        break
```

Therefore, $c = 79$.

From the previous questions, we can easily find that there often exists collisions in hash table. In real world, collisions is everywhere. What we need to do is to reduce the case of collisions.

Suppose we want to store n items into a hash table with size m. Collisions resolved by chaining.

(4) In lecture, Prof. Zhao has mentioned that we often want a hash table which can support `Search()` in $O(1)$ time complexity. Assume simple uniform hashing, that is $E[h(k) = l] = \frac{n}{m}$. Therefore, is `Search()` always has expected running time $O(1)$ as the lecture says? If it always, please explain your reason. If it is not always, under what case can `Search()` for a hash table has expected running time $O(n)$?

NO. It requires finding the position of the element in the hash table ($O(1)$), and then looking through the length of the chain at that location. This yields an overall runtime for search of $O\left(1 + \frac{n}{m}\right)$ if you know the input distribution.

The previous expression only reduces to $O(1)$ if $n = \Theta(m)$. If $n = \Theta(m^2)$ or $m = \Theta(1)$ or $n >> m$, then `Search()` is expected-time $O(n)$.

Common mistakes: You may assume $m = 1$. This is wrong and meaningless since first we don't talk about a certain number and second if you can assume this, I can also assume n is a very large specified number, i.e. 1000000.

After lecture, TA Yuan and TA Xu are all motivated by Prof.Zhao. For this time both of them have two interesting ideas on dealing with collisions to reduce the worst time complexity for each operation.

(5) TA Xu wants to deal with collision via **resizing the array**. He lets $m \to 2m$, but he is so lazy that he doesn't want to change the hash function. Is this method can reduce collisions? Why or Why not?

No. Because this method will insert the keys into original position, which make more space wasted.

(6) TA Yuan wants to store each chain using a data structure S instead of a linked list because he think this method can reduce the worst time complexity for insertion. Assume $\Theta(m) = \Theta(n)$, so the load factor $\lambda = \frac{\Theta(n)}{\Theta(m)} = 1$. Remember that in the hash table, there is no duplications. Suppose the data structure S can be **Binary Search Tree**, **Binary Heap** or **AVL Tree**. What is the worst-case running time of `Insert()` for each data structure in new hash table? Explain your reason briefly.

Before Insertion, we need to search whether there is duplication. This is very important! Please notice it in Lecture and Discussion. In worst case, all n items will hash to the same slot.

BST: Search needs $O(n)$, Insert needs $O(n)$. The total is $O(n)$.

Heap: Search needs $O(n)$, since you don't know where the key is. Insert needs $O(\log n)$. The total is $O(n)$.

AVL: Search needs $O(\log n)$, Insert needs $O(\log n)$. The total is $O(\log n)$.

Therefore, only AVL is the best choice among them. This is why it is the only optimized data strcture written in Lecture.

Problem 2: Hash Collisions - Open Addressing

ShanghaiTech has recently instituted a policy of renaming students from the conventional "First-Name Last-Name" to "Student k" where k refers to the student's ID number. Keyi Yuan is a CS101 TA. Unfortunately, he isn't a very friendly TA, so the number of students in his section has dwindled to 7.

Keyi wants to maintain his students' records by hashing the student numbers in his recitation into a hash table of size 10. He is using the hash function $h(k) = k \mod 10$ to insert each "Student k" and is using linear probing to resolve collisions.

Professor Zhao has handed Keyi an ordered list of the seven students in his section. After inserting these students into his hash table one at a time, Keyi's completed hash table looks like this:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys	24				14	35	54	55	98	17

(1) Keyi is supposed to return the list of students to the professor, but he accidentally spills *A Little Little Tea Milk* on it, rendering it illegible. To cover his incompetence, Keyi wants to hide his mistake by recreating the list of student numbers in the order they were given, which is the same as the order they were inserted; however, he only remembers two facts about this order:

1. 98 was the first student number to be inserted.
2. 35 was inserted before 14.

Help Keyi by figuring out what the order must have been. Write the order directly.

The order: 98, 35, 14, 54, 55, 17, 24

Reason: 98 is inserted first by the first fact.

Note that other than 14 and 35, all the other student numbers are not inserted into the location the hash function would return for them, suggesting that their location must be a result of linear probing after a collision. Due to fact 2, we can determine that 35 was inserted second.

The only student number that would cause a collision is now 55. If 55 were inserted next, it would end up at cell 6. Therefore, 14 must be inserted third.

Student numbers 24, 54, and 55 would cause collisions. However, insertion of 24 or 55 would end up at cell 6, which is incorrect. Therefore, 54 must be inserted fourth.

We are left with 24 and 55 causing collisions. Insertion of 24 would end up at cell 7, which is incorrect. Therefore, 55 must be inserted fifth.

All that is left are 17 and 24. Insertion of 24 would put it at cell 9, which is incorrect. Therefore, 17 must be inserted sixth.

Finally, 24 must be inserted last.

(2) Student 98 leaves Keyi's section, so Keyi deletes that record from the hash table. The next day, Keyi sees Student 15 yawning during the lecture, so he decides to give Student 15 a zero in class participation. However, Keyi can't remember whether Student 15 is actually in his section, so he decides to look up Student 15 in his hash table. **Assume we use erasing methods in Slides 149-151.** Which cells does Keyi need to inspect to determine whether Student 15 is in the table? Write down the probe sequence directly.

5→6→7→8→9→0

Reason: After 98 is removed from the table, we must remember that our linear probing collision resolution requires us to skip over deleted cells. Therefore, first cells 5, 6, 7, 8 and 9 are probed. Finally, after examining

cell 0 and realizing that it was always empty, we can stop inspecting cells and declare that 15 is not in the hash table.

Common mistakes: You may not check the cell 0. This is uncorrect since the computer doesn't know which value stores in the whole hash table.

(3) Student 98 rejoins Keyi's section, but Keyi is forgetful. So he uses a new hash function to insert all students into a new empty hash table:

$$h(k) = ((k+7) * (k+6)/16 + k) \mod 10$$

Collisions are resolved by using quadratic probing, with the probe function:

$$(i^2 + i)/2$$

Fill in the final contents of the hash table after the key values have been inserted in the order which is the same as question (1). We need to specify that the a/b opertaion means $a/b = \lfloor \frac{a}{b} \rfloor$.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Keys	98	14	35	54	55	24		17		

Problem 3: Secret Love In ShanghaiTech

Bob and Alice are sophomores in ShanghaiTech. Bob loves Alice. But Alice doesn't know he loves her. And Bob is too shy to express his love. Therefore, he wants to send Alice a secret message of characters via a specially encoded sequence of integers. Each character of Bob's message corresponds to a **satisfying** triple of distinct integers from the sequence satisfying the following property: every permutation of the triple occurs consecutively within the sequence.

For example, if Bob sends the sequence $A = (4, 10, 3, 4, 10, 9, 3, 10, 4, 3, 10, 4, 8, 3)$, then the triple $t = \{3, 4, 10\}$ is satisfying because every permutation of t appears consecutively in the sequence, i.e. $(3, 4, 10)$, $(3, 10, 4)$, $(4, 3, 10)$, $(4, 10, 3)$, $(10, 3, 4)$ and $(10, 4, 3)$ all appear as consecutive sub-sequences of A.

For any satisfying triple, its **initial occurrence** is the smallest index i such that all permutations of t is shown from index 0 to i in the sequence. For example, the initial occurrence of t in A is 10.

Bob's secret message will be formed by characters corresponding to each satisfying triple in the encoded sequence, increasingly ordered by initial occurrence. To convert a satisfying triple (a, b, c) into a character, Bob sends Alice an arithmatic function $f(a, b, c) = (a + b + c) \mod 27$ to decode the message. $f(\cdot) = 0$ to 25 correspond to the lower-case letters 'a' to 'z', while $f(\cdot) = 26$ corresponds to a space character. For example, the triple $\{3, 4, 10\}$ corresponds to the letter 'r'.

(1) Suppose Bob sends Alice the following sequence:

$$(10, 13, 9, 10, 13, 5, 2, 13, 5, 10, 9, 13, 10, 9, 13, 2, 5, 13, 2, 67, 23, 1, 2, 10, 1, 2, 1, 10, 2, 1)$$

Write down the list of satisfying triples contained in the sequence, as well as their associated characters, ordered by initial occurrence. What is the secret message? Fill in the table below.

You may not fill in all rows of the table, or you may need to add rows of the table.

Triple	$(a + b + c) \mod 27$	**Letter**	**Initial Occurrence**
$(13, 10, 9)$	5	f	13
$(5, 13, 2)$	20	u	18
$(10, 2, 1)$	13	n	29

Therefore, the secret message is 'fun'.

(2) Now Bob has the courage to express his love to Alice. But Alice refuses him immediately, because she is tired of decoding his messages by hand. After that, Bob becomes very sad, knowing that Alice is very angry about the process finding all satisfying triples, which needs $O(n^3)$ time complexity. In order to help him cheer up, please design an expected $O(n)$ time algorithm to decode a sequence of n integers from Bob to output his secret message. Hint: Hash Table.

Scan each consecutive triple from Bob's sequence, and use a hash table to keep track of the triples seen so far. When processing a triple of integers (a, b, c), check whether a, b and c are unique. If they are not unique, then (a, b, c) cannot be a satisfying triple. It they are unique, check whether (a, b, c) is stored in the hash table.

If so, we've seen this triple before, so this triple will not complete an initial occurrence of a satisfying triple. Otherwise, if (a, b, c) is not in the hash table, add the newly seen ordered triple to the hash table. Then check whether the five permutations of the triple, (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a), all exist in the hash table. If all five permutations exist in the hash table, this means that we have previously encountered all the other permutations, followed by the first occurrence of (a, b, c). This is the definition of a satisfying triple, so append the triple's decoded character to an output string.

Each hash table insert and find operation takes expected $O(1)$ time; each triple requires at most six finds and one insertion, so can be processed in expected $O(1)$ time. Thus, to process all $O(n)$ consecutive triples of Bob's sequence can be done in expected $O(n)$ time.

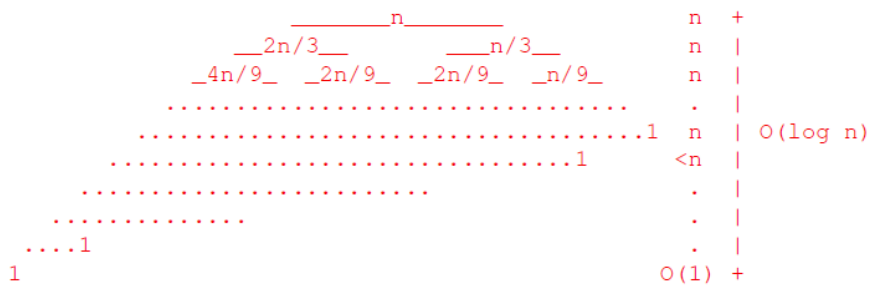
Notice: We will not focus on the hash function.

Problem 4: Inequal Divide

In this problem, we will analyze an alternative to divide step of merge sort. Consider an algorithm `InequalSort()`, identical to `MergeSort()` except that, instead of dividing an array of size n into two arrays of size $\frac{n}{2}$ to recursively sort, we divide into two arrays with inequal size. For simplicity, assume that n is always divisible by vivisors (i.e. you may ignore floors and ceilings).

(1) **Analysis:** If we divide into two arrays with sizes roughly $\frac{n}{3}$ and $\frac{2n}{3}$. Write down a recurrence relation for `InequalSort()`. Assume that merging takes $\Theta(n)$ time. Show that the solution to your recurrence relation is $O(n \log n)$ by drawing out a recursion tree, assuming $T(1) = O(1)$. Note, you need to prove both upper and lower bounds.

$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + \Theta(n)$.



The root does no more than cn work for some positive constant c. Then the second level does no more than $c(n/3) + c(2n/3) = cn(1/3 + 2/3) = cn$ work, the third level no more than $c(n/9) + c(2n/9) + c(2n/9) + c(4n/9) = cn(1/3 + 2/3)^2 = cn$, and so on. In fact, each level requires at most cn work, though may do significantly less work near the bottom of the unbalanced tree. The longest root to leaf path has height $\log_{3/2} n = O(\log n)$, leading to a $O(n \log n)$ upper bound.

The work is also lower bounded by $\Omega(n \log n)$, because the subtree above and including level $h = \log_{3/1} n$ is a complete binary tree with height h, for which each level does at least $c'n$ for some positive constant c'.

(2) **Generalization:** If we divide array into two arrays of size $\frac{n}{a}$ and $\frac{(a-1)n}{a}$ for arbitrary constant $1 < a$ recursively, what is the asymptotic runtime of the algorithm? Is there any change on time complexity?

In recursive tree, each level requires at most $cn(1/a + (a-1)/a)^i = O(n)$ work, while the height of the tree is $\log_{a/(a-1)} n$ which is $O(\log n)$ for any constant $a > 0$.

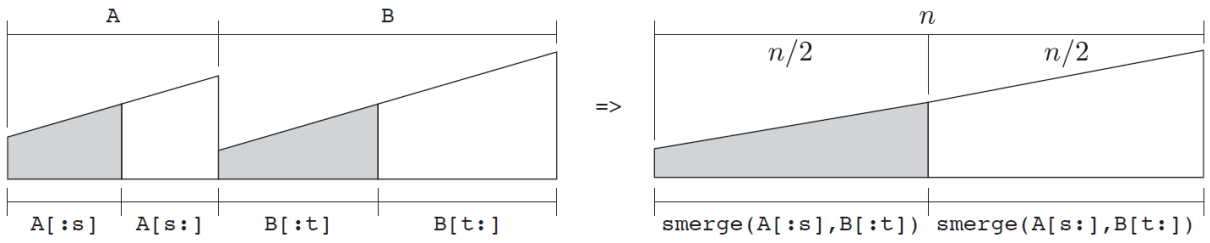
(3) **Limitation:** If we divide array into two arrays of size a and $n - a$ for some positive constant integer a recursively, what is the asymptotic runtime of the algorithm? Is there any change on time complexity? Assume that merging still takes $\Theta(n)$ time, $T(a) = O(a)$. It may help to draw a new recursion tree.

Here, the amount of work done at the root (level 0) is cn, while the work done at the next level is $ca + c(n-a) = cn$. Then the work done at level $i+1$ is $c(n - ia)$. Then the total runtime of the algorithm is:

$$\begin{aligned} cn + \sum_{i=0}^{n/a} c(n - ia) =& cn + cn\left(\frac{n}{a} + 1\right) - ca\frac{\frac{n}{a} \cdot \left(\frac{n}{a} + 1\right)}{2} \\ =& \frac{c}{a}\left(n^2 - \frac{n^2}{2a} + 2an - \frac{a^2}{2}\right) \\ =& \Theta(n^2) \end{aligned}$$

Problem 5: Slice Merge

In this problem, we will analyze an alternative to merge step of merge sort. Suppose A and B are sorted arrays with possibly different lengths, and let $n =$ `len(A)+len(B)`. You may assume n is a power of two and all n items have distinct keys. The slice merge algorithm, `smerge(A,B)`, merges A and B into a single sorted arrays as follows:



Step 1: Find indices s and t such that $s+t = \frac{n}{2}$ and prefix subarrays `A[:s]` and `B[:t]` together contain the smallest $\frac{n}{2}$ keys from A and B combined.

Step 2: Recursively compute `X = smerge(A[:s], B[:t])` and `Y = smerge(A[s:], B[t:])`, and return their concatenation `X + Y`, a sorted array consisting of all items from A and B.

For example, if $A = [1,3,4,6,8]$ and $B = [2,5,7]$, we find $s = 3$ and $t = 1$ and then recursively compute:

```
smerge([1,3,4], [2]) + smerge([6,8], [5,7]) = [1,2,3,4] + [5,6,7,8]
```

(1) Describe an algorithm to find indices s and t satisfying step (1) in $O(n)$ time, using only $O(1)$ additional space beyond array A and B themselves. Remember to argue the correctness and running time of your algorithm.

To find indices s and t, begin with $s = 0$ and $t = 0$. If $A[s] < B[t]$, add one to s, otherwise, add one to t, and repeat until $s + t = n/2$. Claim: at the end of each iteration, $A[:s]$ and $B[:t]$ contains the $s + t$ smallest keys from A and B combined. Proof by induction. At that start, $s + t = 0$ so the claim is vacuously true. Now assume the claim holds at end of iteration $k = s + t$ so that $A[:s]$ and $B[:t]$ contain the smallest $s + t$ elements from A and B. Arrays A and B are sorted, so the minimum of elements from $A[s:]$ and $B[t:]$ is either $A[s]$ or $B[t]$. The procedure adds one to whichever index contains the smaller of the two, reinstating the inductive hypothesis. This algorithm does constant work in each of $n/2$ iterations, using only constant external storage to keep track of s and t, so this algorithm takes $O(n)$ time.

(2) Write and solve a recurrence for $T(n)$, the running time of `smerge(A,B)` when A and B contain a total of n items. Please show your steps. How does this running time compare to the `merge` step of `MergeSort()`?

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$, so $T(n) = \Theta(n \log n)$ by Case Two of the master theorem, since $f(n) = \Theta(n) = \Theta(n^{\log_2 2})$.

$\Theta(n)$ is the time complexity of (1).

(3) Let `smerge_sort(A)` be a variant of `MergeSort(A)` that uses `smerge` in place of `merge`. Write and solve a recurrence for the running time of `smerge_sort(A)`. Please show your steps.

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \log n)$, so $T(n) = \Theta(n \log^2 n)$ by Case Two of the master theorem, since $f(n) = \Theta(n \log n) = \Theta(n^{\log_2 2} \log^1 n)$.

$\Theta(n \log n)$ is the time complexity of (2).