Homework Assignment 2

CSC 263H

Out: May 29, 2016
Due: June 17, 2016 Midnight

Be sure to include your name and student number with your assignment. If your handwriting is possibly illegible, be sure to hand in your assignment in some typed form. Partners should only submit one copy to Markus.

When using results seen in class or the book do not rewrite the code. Instead just talk about how you use or modify these items to answer the questions.

1. For this question you will design algorithms that utilizes data structures we have seen in class to solve a problem.

Consider a set of classes \( C \) and items \( X \). Each item \( x_i \in X \) has a class \( c_j \in C \). For ease of notation we let \( c \in C \) be the set of items that belong to its class, that is \( x \in c \iff x.class = c \). Each item has a value \( x_i.key \in \mathbb{R} \), we can assume the keys are unique.

We want to implement the following operations:

\( \text{Insert}(x,c) : X = X \cup \{x\}, \text{and } x.class = c. \) Informally this adds element \( x \) with class \( c \) to the set (you may assume that \( c \in C \)).

\( \text{DeleteMax}(c) : X = X \setminus \{x\} \text{ where } x = \text{argmax}_{x_i \in c}(x_i.key). \) Informally this removes the largest element from class \( c \).

\( \text{CreateClass}(c,N) : \text{Creates a new class } C = C \cup \{c\} \text{ with the elements in } N. \) Before we assume that the elements of \( N \) are all new so \( (N \cap X = \emptyset) \), afterwards \( X = N \cup X \). Informally creates a new class comprised of (and only of) the elements in \( N \).

\( \text{SumLargest}(L) = \sum_{c \in L} \text{max}_{x \in c}(x.key). \) Informally finds the largest element in each of the classes specified by \( L \subseteq C \) and sums their largest values.

\( \text{MergeClasses}((c_1, c_2), c_{\text{new}}). \) Remove classes \( c_1 \) and \( c_2 \) from \( C \) and create a new class \( c_{\text{new}} \) which contains (and only contains) the items that were in \( c_1 \) and \( c_2 \). Informally merge the elements of the two classes into a new class and destroy the old ones.

\( \text{Insert}(x,c) \) should run in \( O(\log(|c|) + |C|) \), that is logarithmic in the number of elements in \( c \).

\( \text{DeleteMax}(c) \) should run in \( O(\log(|c|) + |C|) \), that is logarithmic in the number of elements in \( c \).

\( \text{CreateClass}(c,N) \) should run in \( O(|N|) \), that is linear in the number of items in \( N \).

\( \text{SumLargest}(L) \) should run in time \( O(|C|) \) that is linear in the number of classes.

\( \text{MergeClasses}((c_1, c_2), c_{\text{new}}) \) should run in time \( O(|c_1| + |c_2| + |C|) \) that is linear in the total number of elements in the merged classes.
Describe what data structure(s) you will use and how you will implement each method. Provide pseudo code and a brief justification for the runtimes and correctness.

Implementations that do not meet these bounds will not get any marks. The reason for the $+|C|$ in the bounds is you’ll need to iterate through and find different classes, the number of classes could be very large relative to the other items in the bounds.

**Update:** For the classes specified by $L$ you may assume that determining if class $c$ is a member of $L$ ($c \in L$) can be done in constant time.
Solution:

We will use a series of max heaps (one for each class) linked via a linked list. We will use the array representation of heaps that we saw in class.

At a high level our list $C$ will be a doubly linked list of nodes where each node in addition to the two pointers contains a pointer to it’s member heap and the name of the class that heap represents.

It is easy to see that to search for a class we only require scanning the link list until we match the name of that class. Assuming we are tracking $|C|$ classes this operation only take $O(|C|)$ time.

Now we describe how to do each operation and why it fits in our required times.

$Insert(x, c):$

Scan the list for class $c$. When you find it perform a heap insert of $x$ into the array we found. Heap insert takes $O(\log(|c|))$ time and scanning takes $O(|C|)$ time. Thus our total time is $O(\log(|c|) + |C|)$.

$DeleteMax(c)$ should run in $O(\log(|c|) + |C|)$, that is logarithmic in the number of elements in $c$.

Scan the list for class $c$. When you find it perform a extract max on the array we found. Extract max takes $O(\log(|c|))$ time and scanning takes $O(|C|)$ time. Thus our total time is $O(\log(|c|) + |C|)$.

$CreateClass(c, N):$

Insert at the head of $C$ a new node which contains $c$ as the name and attach an array to the pointer with the elements of $N$. Call make heap on this array (takes time $O(|N|))$ it is now a proper max heap. The only real work was transforming $N$ into a heap, inserting at the head of a doubly liked list takes constant time) this took time $O(|N|)$.

$SumLargest(L):$

Walk through $C$ and for each node check if it’s class is in $L$ (we said this only took $O(1)$ time). If a class is in $L$ add it’s maximum value (can be found in $O(1)$ time by looking at the first element of its array) add it to a running sum. Since at each node we only do constant work we only need to account for the time it takes to traverse the list which is $O(|C|)$.

$MergeClasses((c_1, c_2), c_{\text{new}}):$

Walk through $C$ and when you find $c_1$ or $c_2$ copy their array into a new array (whichever one you find second append it to the end of the new array) and remove the node. Removing a node only takes constant time since we are just moving some pointers around. Copying the contents of an array take time linear in the size of the array, that is $O(|c_1|)$ and $O(|c_2|)$. After finding both classes (and removing their nodes) create a new node with the name $c_{\text{new}}$ insert it anywhere in the list and attach it’s pointer to the new array. We can simply transform the new array into a max heap operation in time $O(|c_1| + |c_2|)$ inserting the new node is a constant time operation since we just move a few pointers around.

Thus our total time is $O(|c_1| + |c_2|) + O(|c_1|) + O(|c_2|) + O(|C|) \in O(|c_1| + |c_2| + |C|)$. 

3
2. This question involves augmenting a data structure we have seen.

**Hint:** You will need to augment an AVL tree to do this question.

Consider the following ADT. We have a set $X = x_1, \ldots, x_n$ of unique numbers.

The operations are:

$Insert(x) : X = X \cup \{x\}$

$GreaterThan(x) : \text{return the sum of all elements in } X \text{ which are greater than or equal to } x$. Note $x$ may or may not be in $X$.

Both of these should run in $O(\log n)$ time.

What data structure will you augment to implement this ADT? How will you augment your data structure?

Show the pseudocode for your operations. Note you may reuse any operations we have seen for your chosen data structure. If you are simply modifying an operation don’t rewrite the code, instead describe how you would modify it. Briefly justify why your operations work and run in the required time.

All operations should run in time $O(\log n)$. Briefly justify why your method works, that is why is it correct and why does it run in the appropriate amount of time.

**Solution:**

We will augment an AVL tree to solve this problem. We will simply map each element of $X$ to a node in an AVL tree using the value of $x$ as the key in the tree. We will assume that we have a pointer to the root node of the tree, we call this $X.root$.

As we saw in class an AVL tree is a BST that in addition to the key value stores the balance factor at each node. We further augment this by storing at each node $i$ the sum of all values under it at that node (that is the sum of all values contained in the subtree rooted at node $i$).

It is easy to see if each node contains this augmented value we can efficiently calculate $i.sum$ as $i.key + i.LeftChild.sum + i.RightChild.sum$. For nodes without a left (right) child we simply assume $i.LeftChild.sum (i.RightChild.sum)$ is zero. This means for leaf nodes $i.sum = i.key$.

$Insert(x)$:

We are just going to copy the insert from the sample solutions I posted awhile back.

$GreaterThan(x)$:

We simply search for $x$ as we do in a normal BST search. During the search we will keep a running sum $S$. There are three cases to consider at an internal node $k$:

If $k.key > x$:

We add to $S$ the value at $k.key$ and $k.right.sum$. We continue the traversal.

If $k.key < x$:

We continue the traversal.

if $k.key = x$:

We add to $S$ the value in $k.right.sum$ and $k.key$, we then return $S$.

For a leaf node $l$ there are again three cases to consider:
If $l.key > x$:
We add to $S$ the value at $l.key$ and return $S$.

If $l.key < x$:
We return $S$
if $l.key = x$:
We add to $S$ the value at $l.key$ and return $S$.

Since at each node we only do a constant amount of work our run time is that of an AVL search, $O(\log n)$.

To show correctness we need to argue that if a node $j$ has $j.key \geq x$ we add $j.key$ to $S$ one and only one time.

At some internal node $k$ if we decide to go left (case 1) that means $k.key > x.key$, thus we need to add it to our sum. Furthermore each node to the right of $k$ has a key larger than $x$ (by the BST property), thus we add each of their values to the sum using $k.right.sum$. Since we go down a level and left we will never count any of the values in $k$ or the subtree rooted at $k.right$ again. So all of these values got added one and only one time.

At some internal node $k$ if we decide to go right (case 2) that means $k.key < x.key$, thus $k$ did not belong in the sum and neither did anything in its left subtree (again by the BST property). We never consider these elements again and continue on to the right subtree.

At some internal node $k$ if we decide to stop (case 3) that means $k.key = x.key$, thus we need to add it to our sum. Furthermore each node to the right of $k$ has a key larger than $x$ (by the BST property), thus we add each of their values to the sum using $k.right.sum$. We can ignore nodes in the subtree at $k.left$ since by the BST property $x.key$ is larger than anything in the left subtree. We have considered all of the nodes in the tree at this point (the ones under us in this case and the ones parallel and above us in previous cases).

The justification for leaf nodes is similar to the internal nodes.
3. Show the resulting AVL tree from performing the following insertions and deletions (in the order presented).

Insert: 8,9,7,11,10,3,2,1 Delete 9,11

Only show the final tree (with each node and its balance factor). Intermediate work will not be graded.

While you can easily do this with one of many tools online understanding how to do it via hand may be useful for a test...

Solution:

```
03
=_________
/        /
02       08
      /-1-
     01 07 10 0
```
4. Given a collection $X$ of $n$ words devise a method that can perform the following operations:

- $\text{Anagrams}(x)$: return a list of all anagrams of word $x$ contained in $X$.
- $\text{Insert}(x)$: Add word $x$ to set $X$.

For this you may assume that all words are in english (so they are only comprised of some combination of 26 characters) and no word is longer than a constant length $c$. The set $X$ may contain duplicates. For example if $X = \{\text{dog, cat, dog, god}\}$ and we perform $\text{Anagrams(ogd)}$ we get back $[\text{dog, dog, god}]$ (order here is not important).

You may make certain probabilistic assumptions we have discussed in class.

Before performing any operations you have $O(n)$ time to preprocess $X$ into whatever format you need. For time bounds $\text{Insert}(x)$ must run in $O(1)$ and $\text{Anagrams}(x)$ must run in $O(1)$. All three times given are in expectation and not the worst case.

**Hint1:** If you do your analysis carefully / are careful with your technique you can actually show these run times are all worst case runtimes. Showing worst case runtimes is sufficient to show expected case runtimes.

For returning the list you may simply return a pointer to that list, since it may take non constant time to even print out all the anagrams of a word in $X$.

What assumptions will you make? Describe your method in both english and pseudo code (implementation of the two functions and preprocessing technique). Briefly justify why they are correct and run in the required bounds.
Solution:

Note: because of the constant length restriction on a word this question turned out to be easier than I wanted. I will answer here as if we didn’t have the constant and show how that simplifies our final analysis.

We will solve this via hashing (The probabilistic assumption part was a big hint towards this).

As with hashing we must first describe what we are hashing, that is what is our set of keys.

Unlike the hashing questions we have seen before we won’t be hashing the raw elements, instead we must come up with something more clever. We associate with a word \( w \) an array of length 26, one slot for each letter. In the \( i \)\(^{th} \) slot we will insert the number of times the \( i \)\(^{th} \) letter appears in \( w \). Call this array \( a_w \). Now our set of keys is simply the set of all length 26 arrays of non negative integers. The main observation here is that if \( w_1 \) and \( w_2 \) are anagrams they must contain the exact same amount of letters for each kind of letter, thus \( a_{w_1} \) is the same as \( a_{w_2} \). Perhaps more importantly when hashing \( a_{w_1} \) and \( a_{w_2} \) we end up with the same bucket (under any hash function) since the arrays are identical, now we have a way of linking anagrams to the same bucket. We can easily construct the described array in \( O(c) \) time by iterating through each letter of the word (there are at most \( c \) of them) and incrementing the appropriate array value (the array is of length 26 so it only requires \( O(1) \) lookup time).

Now we will pick the number of buckets to use. While we normally pick the number to be some fraction of the size of the key set it will suffice to pick it to be some fraction of \( n \). Let \( m = \frac{n}{100} \).

We will assume our hash function \( h() \) satisfies SUHA as always.

Now we describe the hash table. For a word \( w \) and its array \( a_w \) we store it’s information in the bucket \( h(a_w) \). The nodes in that chain will store two values, a key to index it (simply the array \( a_w \)) and a pointer to a linked list \( L_{a_w} \). \( L_{a_w} \) will contain all of the words that can generate \( a_w \), that is all the words that are anagrams of each other.

It is easy to see now if we want the anagrams of a word \( w \) we simply need to look at the linked list attached to the node indexed by \( a_w \) which is in the bucket \( h(a_w) \).

Now our procedure for inserting a word \( w \) is as follows:

Calculate \( a_w \), and examine the chain at \( h(a_w) \).

If we find a node with the key \( a_w \) insert \( w \) to the beginning of the list \( L_{a_w} \).

If we don’t find a node with the key \( a_w \), create a new node with key \( a_w \) attach it to the start of the chain at this bucket. For it’s list \( L_{a_w} \) let \( w \) be it’s only element.

To preprocess our input we just call the insert operation on each word in our set.

Now to analyze the runtime.

Since we have \( n \) words we can have at most \( n \) word arrays (it is very likely that we have far less but we are simply trying to upper bound our expected runtime). We have at most \( n \) arrays to distribute and by SUHA we will distribute them uniformly over \( m = \frac{n}{100} \) in expectation. Thus we expect each bucket to have at most \( \alpha = \frac{n}{m} = 100 \) keys assigned to it. Since our nodes in a chain are simply the keys (without duplicate) to scan a chain takes constant time in expectation.

When actually inserting a word into a node in a chain this can again be done in constant time since we are just inserting at the head of a list at that node. When returning a list of anagrams we are simply
returning a pointer to a list at a node which is again a constant time operation. Both of these are worst case times, the expected case time comes from the expectation of scanning a bucket chain in constant time.

So now to preprocess we need:

\( \mathcal{O}(c) \) to transform a word to a list (worst case), since we do this for \( n \) words this is a total of \( \mathcal{O}(cn) \).

To scan a to find the key \( \mathcal{O}(1) \) time (in expectation) and \( \mathcal{O}(1) \) time to do the insertion (worst case). Thus our total time is \( \mathcal{O}(cn) \) (in expectation). But since \( c \) is a constant we get \( \mathcal{O}(n) \) time.

As we just argued above insertion and searching just require transforming a word into its array form which takes \( \mathcal{O}(c) \) time in the worst case and scanning a chain which takes \( \mathcal{O}(1) \) time in expectation. Thus the total time is just \( \mathcal{O}(c) \) in expectation. But since \( c \) is a constant we get \( \mathcal{O}(1) \) time.
5. Consider the following procedure:

We have a sequence of \( n \) rounds. In each round we flip a pair of fair coins. If in any round both coins turn up heads we say “Coin based Yahtzee” and stop the procedure.

What is the probability in round \( i \) we say “Coin based Yahtzee”, express this in terms of \( i \). What is the probability we say “Coin based Yahtzee” at any time, express this in terms of \( n \) and \( i \) you do not need to simplify this.

What if we don’t stop after saying “Coin based Yahtzee”, that is we repeat the procedure \( n \) times. How many times do you expect to say “Coin based Yahtzee”, express this in terms of \( n \).

Show your work for all these parts.

**Solution:**

For the first part, probability in round \( i \) we say “Coin based Yahtzee”:

If we say “Coin based Yahtzee” in round \( i \) that means we must have flipped a pair of heads, the probability of this happening is \( \frac{1}{4} \). Furthermore it means in each of the previous rounds we must have not flipped a pair of heads (for if it were not the case we would have stopped). In a fixed round the probability of not getting a pair of heads is \( \frac{3}{4} \). Since each round is independent the probability of getting \( i - 1 \) non pairs of heads is \( \left( \frac{3}{4} \right)^{i-1} \). Thus the final probability is the probability we get a pair of heads in round \( i \) and the probability we didn’t get any pairs of heads in the previous rounds, by independence between rounds, this is \( \left( \frac{3}{4} \right)^{i-1} \cdot \frac{1}{4} \).

For the second part, the probability we say “Coin based Yahtzee” at any time:

What we calculated above was \( \Pr(“Coin based Yahtzee” \text{ in round } i) \). Now we want:

\[
\Pr(“Coin based Yahtzee” \text{ in round } 1 \text{ or “Coin based Yahtzee” in round } 2 \text{ or ... or “Coin based Yahtzee” in round } n) = \sum_{i=1}^{n} \left( \frac{3}{4} \right)^{i-1} \cdot \frac{1}{4}.
\]

For the third part, how many times do you expect to say “Coin based Yahtzee”:

We can simply treat this as a binomial distribution where our random variable \( X \) represents the number of successes and \( X \sim B(n, \frac{1}{4}) \). The mean of \( X \) is simply \( E[X] = \frac{n}{4} \).
6. Given a collection of words (possibly with duplicates) $X = x_1, \ldots, x_n$ (each of which contains at most $c_1$ characters) we want to determine if the letters can arranged into a palindrome.

That is using all of the characters (and only the characters) in $X$ determine if they can be rearranged into a palindrome. For example if $X = \text{(run, urn, appa)}$ you would determine yes since we can rearrange the letters to form $\text{apurnrupa}$ (which is technically not an english word but let’s ignore that...).

Assume we are working in english which has a finite alphabet (26 characters). Your method must run in $O(n \cdot c_1)$ worst case time. You may not make any assumptions beyond what we have stated.

Provide pseudo code and a brief justification for why your method works and fits the runtime bounds.

**Solution:**

We will use a direct access table (also known as an array) with 26 slots each containing an integer. Our procedure will be as follows:

```
1: T = Direct Access Table (with 26 slots)
2: for $x_i \in [x_1, x_2, \ldots, x_n]$ do
3:     for letter $\in x_i$ do
4:         if letter is the $j^{th}$ letter in the alphabet then
5:             Increment $T[j]$ by 1
6:         end if
7:     end for
8: end for
9: NumOdd = 0
10: for bucket $\in T$ do
11:     if bucket mod 2 == 1 then
12:         NumOdd = NumOdd + 1
13:     end if
14: end for
15: Return NumOdd $\leq 1$
```

To see why this algorithm is correct we note a palindrome must have an even number of each kind of letter (except for at most one letter). Whatever our palindrome may be wherever we place a letter we must offset it on the “opposite” side of the word with another of the same letter. The one exception is in an odd length palindrome the letter directly in the middle does not have an offset since it is its own “opposite” letter. This also tells us how to build our palindrome assuming we did not reject that means at most one word had an odd number of letters. If this were the case that letter goes in the middle and we sandwich it between its remaining (an even amount) letters. We can repeat this sandwiching of the middle using the rest of the letters (which there is an even amount). If we had no letters with an odd amount then we just do the same as above without the one odd letter in the middle. Thus we can form a palindrome iff for each kind of letter we have an even number of letters and at most one with an odd amount, so our algorithm is correct.

Now we will show our runtime is correct. The first for loop simply iterates through each word and the inner loop iterates through each letter in that word, thus we go through at most $n \cdot c_1$ iterations. Since $T$ contains 26 slots we can index into it quickly (also since we only have 26 letters in english we can easily transform from letter to integer), that is in constant time. At the actual slot we are only doing
some quick math so this is again constant work. Thus the work done in this double for loop can be
done in $O(1)$ time.

When iterating through each bucket (again there are only 26 of them) we are simply doing a mod 2
operation so in total this loop only requires $O(1)$ time.

Thus our runtime is simply $O(n \cdot c_1)$. 
7. **bonus**

Consider the same question (4) from above. Say we change the question so that each word no longer has a constant upper-bound in length. That is \( c \) (the max length of any word) could be arbitrarily large.

You have time \( \mathcal{O}(n \cdot c) \) to preprocess the input. Insert and Anagrams must run in time \( \mathcal{O}(c) \). This time we can’t get a worse case bounds this good, these all must be expected times now. Describe how your analysis differs from the one above. You may not change your technique from above, you may only show how your analysis changes (if at all) and how that gets us the new bounds. We will grade this question very strictly so it will be pretty much all or nothing mark wise.

**Solution:**

Simply go back to the solution for (4) and omit the 2 lines which say “But since \( c \) is a constant we get … time.”