Quiz 1

- Do not open this quiz booklet until directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- You have 120 minutes to earn 120 points. Do not spend too much time on any one problem. Read them all first, and attack them in the order that allows you to make the most progress.
- **You are allowed a 1-page cheat sheet.** No calculators or programmable devices are permitted. No cell phones or other communications devices are permitted.
- Write your solutions in the space provided. If you need more space, write on the back of the sheet containing the problem. Pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. Simply cite them.
- When writing an algorithm, a **clear** description in English will suffice. Pseudo-code is not required unless asked for.
- **Pay close attention to the instructions for each problem.** Depending on the problem, partial credit may be awarded for incomplete answers.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Parts</th>
<th>Points</th>
<th>Grade</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>36</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bonus</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>120</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Name: __________________________

Circle your recitation:

- R01,2 Alin Tomescu 10,11AM
- R03 Deepak Narayanan 12PM
- R04 Joseph Henke 12PM
- R05 Casey O’Brien 1PM
- R06,7 Ilia Lebedev 1,2PM
- R08 Andreea Bodnari 2PM
- R09 Kevin Zatloukal 3PM
- R10 Deniz Oktay 4PM
Problem 1. Short Answer Questions [35 points] (7 parts)

(a) [5 points] If \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \), then \( h(n) = \Omega(f(n)) \). True or False? Justify your answer.

Solution: True. \( O \) is transitive, and \( h(n) = \Omega(f(n)) \) is the same as \( f(n) = O(h(n)) \)

(b) [5 points] \( f(n) \) is defined to be the running time of the program \( A(n) \):

```python
def A(n):
    # a tuple is an immutable version of a
    # list, so we can hash it
    atuple = tuple(range(0, n))

    # A set is a dictionary without values
    S = set()
    for i in range(0, n):
        for j in range(i+1, n):
            # add tuple (i,...,j-1) to set S
            S.add(atuple[i:j])
```

Give a \( \Theta \) bound for \( f(n) \).

Solution: \( \Theta(n^3) \): Inside the two for loops, both slicing and hashing take linear time.
(c) [5 points] In an effort to make MERGE-SORT faster, you decide to divide the array into $k$ equal sized, disjoint subarrays, where $k > 2$. This means that you have to merge $k$ lists. Write the recurrence for this algorithm assuming merge can be accomplished in $O(n \log(k))$ time and solve it, i.e., give a $\Theta$ bound.

**Solution:** The recurrence is:

$$T(n) = k \cdot T(n/k) + n \cdot \log(k)$$

Sketching the recursion tree, we have:

- $O(n \cdot \log(k))$ work at each level.
- $O(\log_k(n))$ levels.

For a total of $O(n \cdot \log_k(n) \cdot \log(k)) = O(n \cdot \log(n))$ work, as $\log_k(n) = \log(n) / \log(k)$.

---

(d) [5 points] A min-heap is an implementation of the Priority Queue abstract data type. Which of the following operations are supported *efficiently* by min-heaps: `FIND(value)`, `FIND-MIN()`, `INSERT(value)`, `DELETE(value)`, `EXTRACT-MIN()`?

**Solution:** `FIND-MIN`, `INSERT`, `EXTRACT-MIN`.

`FIND` takes $O(n)$ time in a heap. Before you run `DELETE`, you have to first find if the value is in the heap, so that takes $O(n)$ time too.

---

(e) [5 points] Suppose that you have inserted 15 distinct items into a min-heap. The minimum element will be in the first position of the array. How many positions in the underlying array might contain the maximum element? *Hint: Draw a picture of the tree.*

**Solution:** The minimum is in the first position in the array, so only one position may contain the minimum. However, the maximum could be in any of the leaves, of which there are 8.
(f) [5 points] Recall our representation of a heap as a binary tree embedded into an array. Suppose the same representation is used to store a binary search tree. How big of an array would be needed to store a best-case BST of \( n \) elements? How big of an array would be needed to store a worst-case BST of \( n \) elements? Assume \( n \) is \( 2^k - 1 \) for some \( k \).

**Solution:** The BST is perfectly balanced, requiring an array of \( n \) elements. No indexes in the array are unused. The BST has only one child at each node, and it is always the right child, whose index is given by \( i_{\text{right}} = 2i + 1 \). The index of the \( n^{th} \) child is then given by \( 2^n - 1 \) (counting from 1 for the first insertion).

(g) [5 points] Ben Bitdiddle modifies RADIX-SORT to use INSERTION-SORT to sort by digits, instead of COUNTING-SORT. Would the resulting algorithm still work correctly? What is the complexity of this new algorithm? The complexity of conventional RADIX-SORT on \( n \) numbers in base \( b \) with at most \( d \) digits is \( O((n + b) \cdot d) \).

**Solution:** Insertion sort is a stable sort, so the new Radix Sort algorithm would still work correctly. The new complexity becomes \( O(n^2 \cdot d) \), because now we are using insertion sort, an \( O(n^2) \) algorithm, \( d \) times in our Radix Sort.
Problem 2. Binary Search Trees [24 points] (3 parts)

This problem explores several aspects of binary search trees. In all the search trees we consider here, you may assume that the keys are taken from a totally ordered key space $K$, and all the keys stored in the tree are distinct. Totally ordered means that any pair of keys can be compared, and if they are different, one is bigger than the other.

(a) [8 points] Describe (in clear English or pseudocode) an algorithm that does the following: It takes as input an arbitrary (not necessarily balanced) binary tree with unique keys from set $K$ at the nodes, and returns a correct answer saying whether or not the tree is an actual binary search tree. Your algorithm should take time no worse than $O(n)$, where $n$ is the current number of elements in the tree. Include an analysis of the time complexity of your algorithm.

Solution: We need to check that for every node $x$ in the tree, $y.Key < x.Key$ where $y \in x.LeftTree$ and $z.Key > x.Key$ where $z \in x.RightTree$.

An easy way to check this invariant involves observing that an in-order traversal of a BST traverses through the nodes in order of increasing key value.

With this observation, we can use a recursive in-order traversal algorithm to output a list of the keys at the visited nodes. Once we have the list of keys, it is easy to check in $\Theta(n)$ time if the list is sorted or not.

```python
def inorder(B, elements):
    if B == Null: return elements
    inorder(node.left, elements)
    elements.append(node.value)
    inorder(node.right, elements)

def checkBST(B):
    elements = inorder(B, [])
    for index in range(1, length(elements)):
        if elements[index-1] >= elements[index]:
            return False
    return True
```

**Time complexity analysis:** The time complexity for the inorder traversal is $\Theta(n)$, since we traverse all the tree’s nodes exactly once. Checking if the obtained list is sorted is also a $\Theta(n)$ operation. Hence, the overall complexity is $\Theta(n)$.

Common Mistakes: An extremely common mistake we saw was that students proposed the following algorithm,

```python
def checkBST(tree):
    if tree == Null: return True
    if tree.left exists and tree.left.value > tree.value:
        return False
    if tree.right exists and tree.right.value < tree.value:
```
return False
return checkBST(B.left) && checkBST(B.right)

Consider the following simple counter example to see why this algorithm doesn’t work.

```
    3
   / \ 
  2   5
 /   / \
1   4   
```
(b) [8 points] Describe an algorithm that begins with an arbitrary binary tree and produces a count of “how far” the tree is from a real binary search tree. Our measure of how far the tree is from being a BST is the number of distinct pairs of items in the tree that represent violations of the BST property transitively applied. For example in the tree below, (6, 3) is a violation because 6 is to the left of 3. The total number of violations is six: the pairs (6, 3), (6, 2), (6, 4), (3, 2), (6, 5), and (8, 7).

Your algorithm should run in time $O(n + k)$, where $k$ is the number of violations. Include a time complexity analysis.

**Solution:** Our solution involves first using an in-order traversal to produce a list $L$ of elements in the tree. We now claim that with the list $L$, the problem of finding the number of violations in the tree is equivalent to finding the number of inversions in a list. As shown in class, the problem of finding the number of inversions in a list can be solved in $O(n + k)$ time using an augmented version of insertion sort.

**Time complexity analysis:** Since insertion sort never performs any unnecessary interchanges, the total running time of this algorithm can be expressed as $O(\sum_{i=1}^{n}(1 + s_i))$ where $s_i$ is the number of inversions between the element originally at index $i$ in the list, and all elements originally in the subarray $[A[1], A[2], \ldots, A[i-1]]$. (Convince yourself that this is true) Now, since $k$ is the total number of inversions in the list $k = \sum_{i=1}^{n} s_i$, which implies that total running time of the algorithm is $O(n + k)$.

**Common Mistakes:** Most students presented solutions that would not count the violation between 6 and 5 in the original example presented in the problem statement.
(c) [8 points] Describe (in clear English or pseudocode) an $O(n)$ algorithm for balancing an arbitrary binary search tree, that is, for producing a new binary search tree with the same elements as the original one but with height $O(\log(n))$. Include a time complexity analysis.

**Solution:** We first use an in-order traversal to generate a sorted list of the elements in the given BST. Once we have all the elements in sorted order, we can use the following recursive algorithm to construct a well balanced BST:

```python
def construct(elements):
    n = length(elements)
    if n == 0:
        return Null
    middle = n/2
    head = new Node
    head.value = elements[middle]
    head.left = construct(elements[0..middle-1])
    head.right = construct(elements[middle+1, n-1])
    return head

def balanceBST(B):
    elements = inorder(B, [])
    return construct(elements)
```

**Time complexity analysis:** The algorithm has time complexity of $O(n)$ for the in-order traversal and $O(n)$ for the construction of the balanced BST - this follows from the recurrence $T(n) = 2T(n/2) + O(1)$.

**Common Mistakes:** Some students tried to balance the BST bottom up, using rotations to reduce potential height imbalances between left and right subtrees. However, rotating about some node may make sub-trees of that node unbalanced. To make this clearer, consider the following example –

Let $x$ be a node with sub-trees of height $h$ and $h - k$ (where $k \in Z^+$). WLOG, let us assume that the right sub-tree of $x$ has height $h$. If $y$ is the right child of $x$, then let the two sub-trees of $y$ have heights $h - 1$.

Observe that simple rebalancing as explained in lecture would result in node $x$ having sub-trees of height $h - 1$ and $h - k$ – an imbalance further down the tree!
Problem 3.  Treehash [12 points] (4 parts)

Suppose we store $n$ elements in an $m$-slot hash table using chaining, but we store each chain (set of elements hashing to the same slot) using a binary search tree (BST) instead of a linked list. Each element corresponds to a key/value pair and there are no duplicates. Also suppose that $m = n$, so the load factor $\alpha = n/m = 1$. Assume that there will be no resizing of the table.

(a) [3 points] What is the expected running time of insert in this hash table? Why?
   Assume simple uniform hashing.
   **Solution:** The expected running time of insert will be $O(1 + \log \alpha)$. Since $\alpha = n/m = 1$, the expected running time is $O(1)$.

   **Common Mistakes:** The most common mistake was giving the worst case run time instead of the expected runtime.

(b) [3 points] What is the worst-case running time of insert in this hash table? Why? (Do not assume simple uniform hashing.)
   **Solution:** In the worst case, all $n$ items will hash to the same slot and result in a tree of height $O(n)$. Thus, insertion into the tree will take $O(n)$.

   **Common Mistakes:** The most common mistake was assuming that the tree was balanced and giving a worst case time of $O(\log n)$.

(c) [3 points] Suppose that the BST is replaced with an AVL tree. What is the worst-case running time of insert in this new hash table? Why? (Do not assume simple uniform hashing.)
   **Solution:** In the worst case, all $n$ items will hash to the same slot. However, because we have an AVL tree the height of the tree is at most $O(\log n)$, and thus insertion takes $O(\log n)$.

(d) [3 points] Suppose that the AVL tree is replaced with a heap. What is the worst-case running time of insert in this new hash table? Why? (Do not assume simple uniform hashing.)
   **Solution:** In the worst case, all $n$ items will hash to the same slot. To insert an item into a hash table we have to first search for the item (in case we have already inserted that key) and then insert the item into the heap. While insertion into the heap is $O(\log n)$, searching for the item takes $O(n)$. Thus the overall time for insertion is $O(n)$ in the worst case.

   **Common Mistakes:** The most common mistake was to forget to search for the key before inserting it, and thus almost all students gave a runtime of $O(\log n)$.
Problem 4. Hash Table Analysis [12 points] (2 parts)

You are given a hash table with \( n \) keys and \( m \) slots, with the simple uniform hashing assumption (each key is equally likely to be hashed into each slot). Collisions are resolved by chaining.

(a) [4 points] What is the probability that the first slot ends up empty?

Solution: Independently, each key has a \( 1/m \) probability of hashing into the first slot, or \( (m - 1)/m \) probability of not hashing into the first slot. More formally, let \( E_i \) denote the event that the \( i^{th} \) key does not hash into the first slot. Then,

\[
Pr[E_i] = \frac{m - 1}{m}
\]

Thus, the probability that no key hashes into the first slot is

\[
Pr[E_1 \land E_2 \land \ldots \land E_n] = Pr[E_1] \cdot Pr[E_2] \cdot \ldots \cdot Pr[E_n] = \left( \frac{m - 1}{m} \right)^n.
\]

where the first equality is because of the independence of the events \( E_i \), which follows from the simple uniform hashing assumption.

Alternatively, one can derive the same answer through counting. The number of ways in which \( n \) keys hash into \( m \) slots is \( m^n \). The number of ways in which \( n \) keys hash into \( m \) slots, avoiding the first slot, is \( (m - 1)^n \). Thus, the probability that all \( n \) keys avoid the first slot is

\[
\frac{(m - 1)^n}{m^n} = \left( \frac{m - 1}{m} \right)^n
\]

Common Mistakes: The first step in the solution is to realize that the event in question is the intersection of \( n \) events, whose probability is the product of the probabilities that each of the \( n \) events occurs (by simple uniform hashing and consequently, independence of the events.) A common incorrect answer was \( (m - n)/m \).
(b) [8 points] What is the expected number of slots that end up not being empty?

Solution: Let $X_i$ be the event that slot $i$ is nonempty. Since $X_i = 1$ when slot $i$ is nonempty and is 0 otherwise, $E[X_i] = \Pr(X_i = 1)$ is the probability of slot $i$ being nonempty.

The number of nonempty slots is $\sum X_i$. By linearity of expectation, the expected number of nonempty slots is

$$E[\sum X_i] = \sum E[X_i]$$

(Important Note: To apply linearity of expectation, you do not need the random variables $X_i$ to be independent. Indeed, here they are not.)

From part (a), the probability that the first slot is nonempty, or $E[X_1]$, is then $1 - \left(\frac{m-1}{m}\right)^n$, and is the same for all slots. Thus, the expected number of nonempty slots is

$$m \left(1 - \left(\frac{m-1}{m}\right)^n\right).$$

Alternatively, this solution can also be derived (in a rather more complicated way) from first principles and the definition of expected values of random variables, and avoiding linearity of expectation. Linearity of expectation is your friend. Use it!

Common Mistakes:

1. The mathematical definition of expected values: Let $X$ be a random variable which takes values over a universe $\mathcal{U}$ (in this question, the universe $\mathcal{U} = \{0, 1, 2, \ldots, m\}$ representing the number of slots that end up not being empty.) Then, the expected value of $X$ is defined as

$$\sum_{u \in \mathcal{U}} u \cdot \Pr[X = u]$$

Make sure you understand this. If you wrote down the definition of the expected value, you got significant part credit.

2. Some students used the number

$$\left(\frac{m-1}{m}\right)^n$$

computed in part (a) as the probability that exactly one slot is empty. This is not the case. This number is the probability of the event that a given slot, say the first, is empty. This includes other events, e.g., that the first and second are empty, the first and third are empty, and so on...
Problem 5. Snootheby’s [36 points] (6 parts)

Snootheby’s is an auction house that sells expensive art to very rich people. The prices at which they sell their art vary wildly, depending on the current economic situation and the whims of their customers. Consequently, Snootheby’s finds it important to keep careful track of the typical prices at which they have recently sold their art.

Snootheby’s has hired you to design a system (a data structure and algorithms) to keep track of their typical sales prices, specifically, of the $k$ middle sales prices for the current year so far, for some fairly small value $k$ that they will provide as a parameter. For example, if the number of sales and sales prices so far is $n$ and $n - k$ is an odd number, then the number of prices higher than the “middle” ones should be $\left\lfloor \frac{n-k}{2} \right\rfloor$ or $\left\lceil \frac{n-k}{2} \right\rceil$—we don’t care which.

Their accounting starts anew each year, on January 1. Sales data gets recorded in a buffer, from which we will process the information, one sale at a time.

(a) [12 points] Design a data structure to keep track of the $k$ middle sales prices for the current year so far. Your data structure must support two operations:

- **QUERY:** Return the middle $k$ sales prices for the year so far. (If there are fewer than $k$ sales so far then return all of their prices.) This should run in time no worse than $O(k)$.
- **ADD($x$):** Add information about the next sale, with sales price $x$, into the data structure. This should run in time no worse than $O(\log(n) + k)$, where $n$ is the current number of recorded sales.

Describe clearly the data you would maintain and how it would be organized (e.g., “Store a list of such and such sorted in such order”).

Solution: We use a max-heap $MX$, a min-heap $MN$, and a sorted doubly-linked list $L$. $L$ keeps track of the middle $k$ sales prices, with the largest one at the front of the list. $MX$ stores all the sales prices smaller than the middle ones, and $MN$ stores all the sales prices greater than the middle ones. We keep track of the size of each structure, i.e., the number of elements in the structure. We maintain the $MN$ and $MX$ heaps “balanced” in size, i.e., $|size(MN) - size(MX)| \leq 1$.

Alternative solution: use an AVL tree to store all the $n$ sales prices ($LARGE AVL$) together with a sorted list/AVL that stores the $k$ middle sales prices ($MIDDLE AVL$). Together with this data structure we maintain the minimum element and the maximum element of the $MIDDLE AVL$ as well as the number of elements in $LARGE AVL$ smaller than min ($MIDDLE AVL$) as $count_{left}$, and the number of elements in $LARGE AVL$ larger than max ($MIDDLE AVL$) as $count_{right}$. An invariant that needs to be maintained is that the difference between $count_{left}$ and $count_{right}$ is no more than 1.

The **QUERY()** operation would return the elements in the $MIDDLE AVL$ of size $k$, which takes $O(k)$.

The **ADD($x$)** operation would add $x$ to the $LARGE AVL$. Update the number of elements larger/smaller for the $MIDDLE AVL$ min/max elements. Check where $x$ needs to be added: if added inside the $MIDDLE AVL$, remove either the minimum or the maximum, depending which one of $count_{left}$ or $count_{right}$ is larger. Now check whether
the data structure invariant still holds, and update accordingly (as in above main solution) otherwise.

**Common Mistakes:**

- the root of an AVL tree is not guaranteed to be the median of the elements stored inside the AVL. Also, the middle \( k \) elements are not necessarily on the top levels of the AVL. See the example below, where the median is 4 and is one the lowest level inside the AVL.

- obtaining the first \( k \) elements from a heap of length \( n \) takes \( O(k \log n) \) time (you have to \( \text{ExtractMin} \) \( k \) times).

- using hash tables for this type of problem does not provide any benefit

- you cannot store an AVL tree in array efficiently: you will waste array space because an AVL tree is not necessarily a complete binary tree (heaps are always complete binary trees)

- you cannot binary search a linked-list because you do not have \( O(1) \) random access to \( A[i] \) like you do in an array

- while you can add to an array in amortized \( O(1) \) time using table-doubling, you cannot add to sorted array in \( O(1) \) time (amortized or not) because you have to shift elements in the sorted array in order to place the new value at the right position
(b) [8 points] State (but do not prove) two invariants that are always preserved by your algorithm, that is, a collection of properties that are always true after any number of ADD operations have been performed, and that imply that QUERY operations always return correct answers. The two invariants should correspond to an invariant about the prices (keys in your data structure) and an invariant about the sizes of your data structures.

**Solution:** Here is one version. This can be proved by induction on the number of *Add* operations.

1. \( L \) is a sorted doubly-linked list.
2. \( MN \) is a min-heap.
3. \( MX \) is a max-heap.
4. If \( n \leq k \), then \( L \) contains all the elements and \( MX \) and \( MN \) are empty heaps.
5. If \( n > k \), then: (a) Every element of \( MN \) is greater than or equal to every element of \( L \). (b) Every element of \( L \) is greater than or equal to every element of \( MX \). (c) \( L \) contains exactly \( k \) elements. (d) \(|\text{size}(MX) - \text{size}(MN)| \leq 1\).

With this invariant, it is clear that Query returns the \( k \) middle elements as required, since it returns the elements of \( L \). Moreover, the various cases in the definition of *Add* ensure that this invariant is preserved at each step, which yields an inductive proof.
(c) [3 points] Describe (in clear English or pseudocode) how your QUERY operation works. Recall that your goal is an $O(k)$ operation.

**Solution:** Traverse the list $L$ and return all its elements. Alternatively, we could return a pointer to $L$.

(d) [3 points] Analyze the time complexity of your QUERY operation. The goal is an accurate analysis of the complexity of your operation and you will receive full credit for a correct analysis regardless of time complexity.

**Solution:** Traversing and returning the elements of the list $L$ is $O(k)$. Returning just a pointer is $O(1)$.
(e) [6 points] Describe (in clear English or pseudocode) how your ADD\( (x) \) operation works. Recall that your goal is an \( O(\log(n) + k) \) operation.

**Solution:** If \( \text{size}(L) < k \), then just insert \( x \) into the linked list \( L \) at the right location. Otherwise, there are three cases:

1. \( x < \text{min}(L) \).
   Insert \( x \) into \( MX \); if this results in \( \text{size}(MX) > \text{size}(MN) + 1 \), then rebalance: Remove the top element of \( MX \) and insert it at the end of list \( L \) (where the smallest element of \( L \) is stored). Then remove the element at the front of \( L \) and insert it into \( MN \).

2. \( x > \text{max}(L) \).
   This is symmetric with the previous case. Insert \( x \) into \( MN \); if this results in \( \text{size}(MN) > \text{size}(MN) + 1 \), then rebalance: Remove the top element of \( MN \) and insert it at the front of \( L \). Then remove the element at the end of \( L \) and insert it into \( MX \).

3. \( \text{min}(L) \leq x \leq \text{max}(L) \).
   Then insert \( x \) into \( L \) at the right location, and remove one element from \( L \) to maintain the \( k \) size: If \( \text{size}(MN) \leq \text{size}(MX) \) then remove the front element from \( L \) and insert it in \( MN \); otherwise remove the last element from \( L \) and insert it in \( MX \).

(f) [4 points] Analyze the time complexity of your ADD\( (x) \) operation. The goal is an accurate analysis of the complexity of your operation and you will receive full credit for a correct analysis regardless of time complexity.

**Solution:** If \( \text{size}(L) < k \) before the operation is performed, then \( x \) is simply inserted into the list \( L \) and the cost is \( O(k) \). Otherwise, consider the three cases in the operation description above.

1. Finding \( \text{min}(L) \) is \( O(1) \) since \( L \) is sorted. The comparison with \( \text{min}(L) \) is \( O(1) \). Inserting the element into \( MX \) is \( O(\log(n)) \). Removing the top element of \( MX \) and restoring the heap property is \( O(\log(n)) \). Inserting the removed element into \( L \) takes \( O(k) \), and removing the other element takes \( O(1) \). Inserting the other element into \( MN \) is \( O(\log(n)) \). Overall, in this case the Add operation takes \( O(\log(n) + k) \).

2. Symmetric with the previous case, \( O(\log(n) + k) \).

3. Finding \( \text{min}(L) \) and \( \text{max}(L) \) is \( O(1) \). Comparisons are \( O(1) \). Inserting the element into \( L \) is \( O(k) \). Inserting the extra element into one of the heaps is \( O(\log(n)) \). Overall, since \( k \leq n \), the cost is \( O(\log(n) + k) \).
Problem 6.  [5 points] (1 part)

BONUS QUESTION: ONLY ATTEMPT AFTER YOU HAVE COMPLETED THE OTHER QUESTIONS

Give an English word included in the Merriam Webster online dictionary that has 6 or more of the same letter or argue that such a word does not exist. Points will be given for creative and humorous arguments. (Stresslessness is not an included word!)

Solution:

• antidisestablishmentarianism
• supercalifragilisticexpialidocious
• possessiveness (6 s’s)
• senselessness (6 s’s)
• indivisibilities (8 i’s)