Vanessa Nguyen

Homework 2 (Due 3/21/18, 11:55PM)

100 total pts. Please justify your answers with detailed explanations, correct answers with no explanations will receive 0 pts. Submit all answers as one PDF file on Moodle. The name of your answer file should follow this format: FIRSTNAME_LASTNAME.pdf.

1. In class, we learned about comparison sorting algorithms implemented for arrays. Describe how the worst case time complexities of the following sorting algorithms change if they were implement for doubly linked lists? Assume that references to the middle of the linked list exist. Explain in detail, with big-O reductions.

A. (5 pts) Bubble sort?
Worst case: list in reverse order.
Time complexity: O(7 * n * (n – 1) / 2 + n * (n + 1) / 2) ≈ O(n^2); for every switch, we would need to store the value of one of the pair (O(1)). We then need to set the next value of the node previous to the first pair to the second pair value (O(1)) and the previous value of the node next of the second pair to the first pair value (O(1)). Then we need to set a new previous and next for the first and second pair (O(4)). We do this for every pair the first time, and decrease the number of pairs we switch each time we go through the list (because the list is reversed), and we go through the list O(n) times because that is the max possible number of times we can go through the list, leading to O(7 * n * (n – 1) / 2) for all of the sorting, and O(n * (n + 1) / 2) for traversing the sorted parts of the list (we traverse the fully sorted list too). This leads to a total of O(7 * n * (n – 1) / 2 + n * (n + 1) / 2), which reduces to O(n^2).

B. (5 pts) Selection sort?
Worst case: max is first number
Time complexity: O(n * (n – 1) / 2 + 11 * (n – 1)) ≈ O(n^2) ; we need to transverse the rest of the list from the current node to the end to find the minimum of the list for the whole list by comparing the current node to the minimum node (O(n * (n – 1) / 2), when current is at the end of the list, we stop). We need to store the minimum node and set the previous and next nodes for the node previous and next of the minimum node to the current node and set previous and next nodes for the node previous and next of the current node to the minimum node (O(5)). Then we need to set the previous and next of current and minimum to a total of O(9) for a switch. In the worst case, we would need to switch current and minimum each time we go through the list with a total of n – 1 switches, and we would also need to set the values for minimum and current each time we traverse the list (O(2)), leading to O(11 * (n – 1)) for switches. This leads to a total of O(n * (n – 1) / 2 + 11 * (n – 1)) for searching through the array and switching, which reduces to O(n^2).

C. (5 pts) Insertion sort?
Worst case: reverse order
Time complexity: O(n * (n – 1) / 2 + 4 * (n – 1)) ≈ O(n^2) ; we need to transverse the sorted side to the current node to find where the current node belongs (O(n * (n – 1) / 2), when current is at the end of the list, we stop). We then need to insert the node into the correct spot (O(3), change prev/next for current and prev for the node in front of it; in reverse order, the current node would become the head). In the worst case, we would need to move the current each time we go through the list with a total of n – 1 switches, and we would also need to set the values for current each time we traverse the list (O(1)), leading to O(4 * (n – 1)) for switches. This leads to a total of O(n * (n – 1) /2 + 4 * (n – 1)) for searching through the array and switching, which reduces to O(n^2).

D. (5 pts) Merge sort? How would the space complexity change?
Worst case: doesn't really matter
Time complexity: $O((\log_2 n) * (5 * (n - 1) + 1) + (4 * \log_2 n)) \approx O(\log n * n)$. Since there are references to the middle of the linked list, we do not have to traverse the list to find it. We just go to the middle point and make a new list by setting the head of the new linked list as the node after the middle (all the connections should be intact) (O(3), 1 for making the new linked list, another to set the head, and another to set the prev of the first node in this linked list as null), and then we can end the current list at the middle point by setting the next of the last node as null (O(1)). We divide $\log_2 n$ times, so dividing takes a time complexity of $O(4 * \log_2 n)$.
Each time we merge, we need to make a new linked list (O(1)). Since we are making a new linked list for merging, we need to set the next of the head as null, so we can add to it in order (the prev should already be null from the divisions) (O(1)). We need to set a new prev for the node and null for the next each time, and reset the head reference for the divided lists (O(3)), and we do this for all nodes after the new head ($O(n - 1)$). This leads to $O(5 * (n - 1) + 1)$ to merge a list, and we do that $\log_2 n$ times, leading to $O((\log_2 n) * (5 * (n - 1) + 1))$ for merging the list back. Combining the time complexity for dividing and merging gives $O((\log_2 n) * (5 * (n - 1) + 1) + (4 * \log_2 n))$, which reduces down to $O(\log n * n)$.
Space complexity: The space complexity wouldn't really change unless you want to include the values of the node too since we have to store that. It's $O(3 * n) \approx O(n)$. Each node stores the values of itself, the pointer to the next node, and the pointer to the previous node as well, so the space complexity of a node is O(3). You need to make new linked lists consisting of one linked list in merge sort though, leading to $O(3 * n)$ for your space complexity, which reduces down to O(n).

E. (5 pts) Quicksort? How would the space complexity change?
Worst case: Pivot creates subsets of $1/n - 1$
Time complexity: $O(n * (n - 2) + 9 * (n - 1)) \approx O(n^2)$.
In the worst case, you would need to divide $n - 2$ times if all the pivots end up next to each other and start at one end. In a given division, there are n steps taken total from the left and right pointers, leading to $O(n * (n - 2))$ for just the pointers moving in each division. At most, there will be $n - 1$ switches, and each switch will require a value to be store (O(1)), the switching of the prev/next of the prev and next nodes of the pivot and left/right nodes (O(4)), and the switching of the prev and next of the pivot and left/right nodes (O(4)), leading to a time complexity of $O(9 * (n - 1))$. Combining these gets $O(n * (n - 2) + 9 * (n - 1))$, which reduces down to $O(n^2)$.
Space complexity: The space complexity doesn't really change. Space complexity would include the pivots ($O(n - 1)$), divisions ($O(n - 2)$), and left/right pointers ($O(2 * (n - 1))$), leading to $O(3 * (n - 1) + (n - 2))$, which reduces down to O(n).

2. Lucky Charms is an arguably popular American breakfast cereal. It consists of expensive/delicious/delectable marshmallow pieces, and cheap/abhorrent oat pieces. In the year 2049, General Mills (the creator of Lucky Charms), is suspected of reducing the number of expensive marshmallow pieces to cut costs. As a newly hired Federal Fairness Federation investigator, you are tasked with the following:

A. (10 pts) Given M digital boxes of Lucky Charms (with each box represented as an array), and each box containing N total marshmallow and oats pieces, describe an algorithm which would determine the percentage of marshmallows present in O(M*N) time. Assume that M and N are large. Do NOT reduce M and N to "n" in the big-O analysis.

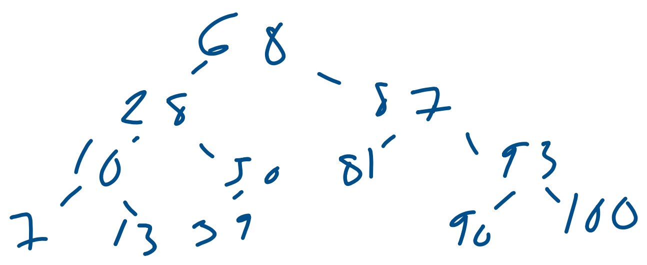
I could go through the data with a counting sort. When I am done counting the pieces, I can record the numbers of marshmallow and oat pieces. A counting sort's time complexity is O(2k + 2n), where k is 2 in this case (marshmallow or oat), so O(4 + 2n), which reduces down to O(n). n in this case is the number of pieces I am going to go through, which is equal to M * N (there are M boxes of N pieces), so the time complexity is O(M * N). For later questions it would be convenient if the marshmallow and oats were separated into two different arrays, so I'll make two different arrays with lengths equal to the count of each pieces and store each piece in their respective array. This won't change the time complexity.

B. (10 pts) Lucky Charms has 8 different marshmallow shapes. You have uncovered Lucky Charms' secret marshmallow recipes, and have discovered that certain marshmallow shapes are more expensive to produce than other marshmallow shapes. Records from 2018 indicate that marshmallow shapes were distributed evenly and fairly into each box of cereal (i.e. each total count of each different marshmallow shape was relatively the same per box). You suspect that General Mills is no longer upholding this standard, and is distributing fewer expensive marshmallows, and more cheap marshmallows. Given the M digital boxes and N total pieces per box in Part A (initially unsorted), describe an algorithm which would determine whether General Mills is fairly distributing expensive and cheap marshmallows in O(M*N) time.
I could do a counting sort like in A to divide the marshmallows and oats, and put them in separate arrays. Counting sorts have a time complexity that reduces to O(n) and since there are M * N pieces to go through, the time complexity is O(M * N) to divide them. Then I can do a counting sort on the marshmallow array and record how many of each marshmallow there are when I finish counting.

C. (10 pts) In the year 3000, Lucky Charms has become the #1 cereal in our Milky Way galaxy. To suit the palates of different species on various planets throughout the galaxy, General Mills executives have developed a new Lucky Charm piece. They claim that each piece produces a random flavor when chewed, supporting up to 1 quadrillion flavors. Given the tasting data of M digital boxes with N total pieces per box from Part A, how would future investigators experimentally determine the veracity of General Mills' claim of random flavors (from a set of 1 quadrillion flavors), without using statistical methods? Assume that computer memory is limited.
They could first do a counting sort like in A to divide the marshmallows and oats, and put them in separate arrays. Counting sorts have a time complexity that reduces to O(n) and since there are M * N pieces to go through, the time complexity is O(M * N) to divide them. Then they can do a quicksort on the marshmallow array and then check the time complexity of the sort. Quicksort works better with random data, so if the time complexity is closer to O(logn * n), then it's more likely that the data was random.

3. Draw the resulting AVL tree, if the following keys are inserted in order into an initially empty AVL tree:

A. (10 pts) 50, 28, 13, 68, 81, 7, 10, 93, 87, 100, 90, 39



Vanessa Nguyen

B. (5 pts) How many of each rotation were used in Part A: single right, single left, double right, and double left?

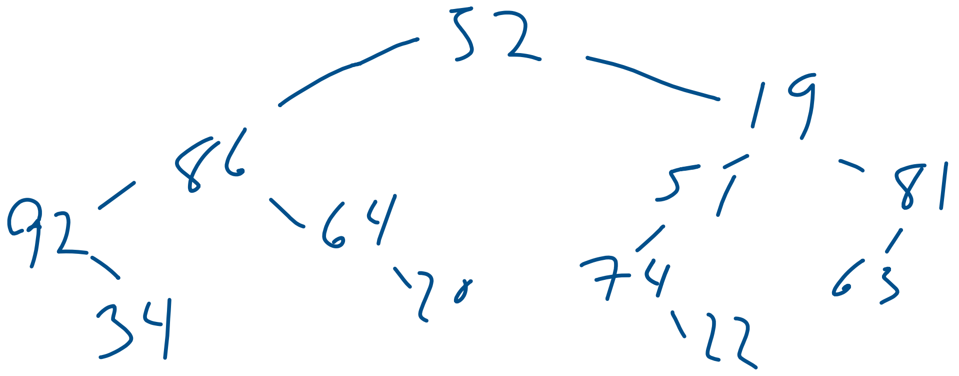
1 single right, 2 single left, 1 double right, and 2 double left.

4. Draw a binary tree which yields the following in-order and post-order traversals:

A. (10 pts) A binary tree which adheres to both of these traversals -

in-order: 92, 34, 86, 64, 20, 52, 74, 22, 51, 19, 63, 81

post-order: 34, 92, 20, 64, 86, 22, 74, 51, 63, 81, 19, 52



B. (5 pts) Write the pre-order traversal for the binary tree in PartA.

52, 86, 92, 34, 64, 20, 19, 51, 74, 22, 81, 63

5. (15 pts) Write code (with comments) for the recursive method, sumBST(BSTnode currentnode). This method performs a post-order traversal of a BST of integers starting from the root, and finds the sum of all integers (data) in the BST.

Assume that the BST is never empty before calling sumBST().

No justification is necessary. Please submit your code (10 out of 15 pts) in writing/typed with the previous 4 questions, i.e. no .java file necessary. Provide comments (5 out of 15 pts) to explain what each part of your code is doing.

```
public class BSTnode {
    BSTnode left;
    BSTnode right;
    int data;
}

public class BST {
    BSTnode root;

    public int sumBST(BSTnode currentnode);
}
```

```
public static void main(String[] args) {
    BST mytree = new BST();
    mytree.add(15);
    mytree.add(9);
    mytree.add(60);
    mytree.add(53);
    mytree.sumBST(mytree.root); // performs (post-order) 9+53+60+15
}


public int sumBST(BSTnode currentnode) {
    int sum = 0; // sum is the sum of the current node and nodes below
it

    // going down the branches

    // checking left brach first, if there is something there, call
function again; will go all the way left
    if (currentnode.left != null) {
        sum += sumBST(currentnode.left); // calling function on left
branch
    }

    // if left branch is empty or all the numbers on that side have
been recorded, go to the right branch next
    // checking right branch next, if there is something there, keep
going
    if (currentnode.right != null) {
        sum += sumBST(currentnode.right); // calling function on right
branch
    }

    // if right and left branch have been recorded/are empty, add the
current node to sum
    sum += currentnode.data;
    return sum; // returns the current sum up the function so that we
can keep adding to it
}
```