CIS350/3501 Summer 2021 Midterm EXAM Part II

(80 points)

Student Name: _Demetrius Johnson____Date:__6/17/2021__

Student ID: _____02870614_____

Due Date/Time:

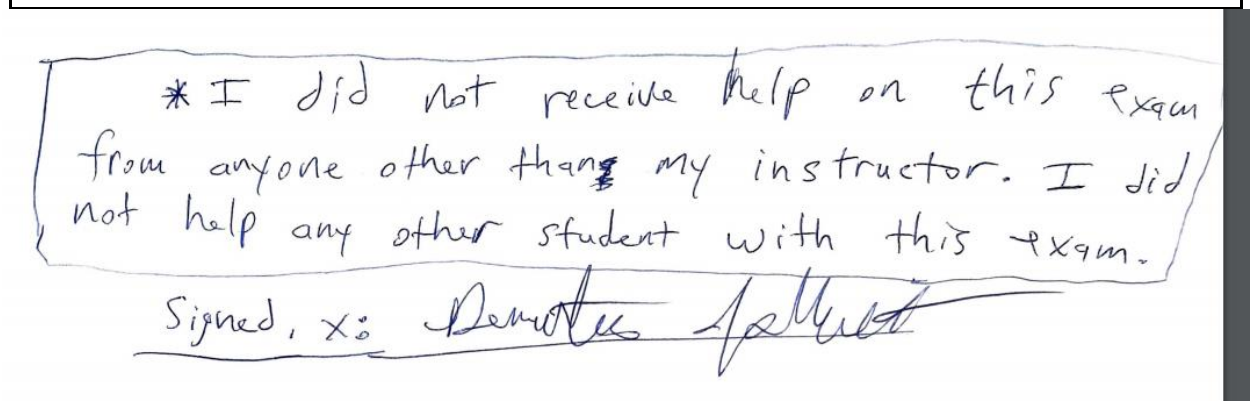
The finished work (as a single pdf or word file) must be submitted on Canvas by **Friday, June 18, 2021, 6:00 PM**.

1. Please provide necessary steps to gain the partial credit if your final solution is not correct.
2. Make an effort to write in a readable fashion. We will skip over (and therefore not grade) non-readable portions.

Note, there are 9 problems on 8 pages.

After you have completed the test, please copy the following statement on the bottom of the last page and then **SIGN and DATE** it to attest that you abided by it:

I did not receive help on this exam from anyone other than my instructor. I did not help any other student with this exam.



1. (15 points) Time Complexity

Give a tightest possible bound for the Big-Oh runtime complexity of the following program fragments or functions in terms of the value of the input parameter `n`:

(a)
```
for (i = 0; i < n; i++)      //n
   a[i] = 0;
for (i = 0; i < n; i++)      //n
   for (j = 0; j < n; j++) //n*n = n^2
      a[i] += a[j] + i + j;
```

- First for-loop: executes n times.
- Second for-loop: executes n times.
- Third for-loop nested inside second for-loop; executes n^2 times.
- Analysis: $n + n + n^2 = n^2 + 2n$.
- Big O() = $O(n^2)$ since n^2 is the highest degree term.

(b)
```
sum = 0
for (i  = 0; i < n*n; i++)  //n*n = n^2
   for (j  = 0; j < i; j++) //n^2
      sum++;
```

- first loop (i) iterates up to n*n times.
- second loop (j) iterates up to i times for each i-loop iteration; thus as n approaches infinity, and the i-loop iterates up to n*n times, the j-loop will essentially simplify to approximately iterating up n*n times for each iteration of the i-loop (the initial smaller-than-n*n j loop iterations for a given i-loop value will become irrelevant as n->infinity.)
- Thus; since the j-loop is nested inside of the i-loop, we do $n^2 * n^2 = n^4$ iterations for the j-loop.
- So a simplified equation for the code segment is $n^2 + n^4$.
- Big O() = $O(n^4)$ since n^4 is the highest degree term.

(c)
```
for (int i = 0; i < n; i++)          //n
    for (int j = 0; j < i*i; j++)    //n^2
       x++;
```

- The first loop executes up to n times.
- The second loop executes up to i*i times which is ~ n*n times for very large n. As n->infinity, the smaller j-loop iterations from the smaller i-loop iteration values will become irrelevant similar to the last question; the j-loop will execute approximately n^2 times for each i-loop iteration.
- Thus; since j is nested inside of i, we multiply i-loop approximation times j-loop approximation for very large n, and say the total number of j-loop iterations is: $n*n^2 = n^3$.
- A simplified equation for the code is $n + n^3$.
- Big O() = $O(n^3)$ since n^3 is the highest degree.

(d)
```
int simple (int n)
{
   if (n < 1000 )
      return 1;
   else
      return n + simple (n - 1);
}
```

- Here we hae a recursive function; as n->infinity, the if statement becomes irrelevant since the value 1000 will theoretically never be reached if we are decrementing from n at infinity, or even for very large n, it won't be reached for a long time before serving as a base case to end the function; thus it is almost no different than if it said n < 0; it will not change the function much; the important statement in the function that will bear weight on the complexity is the else statement.
- So I will focus on the else statement for very large n; the recursive definition here essentially will tell the function to keep doing recursive calls, and as a result the function will add 1 less than n to the current n; so we have a sum like this:
  - n + (n-1) + (n-2) + (n-3) + (n-4) +...+ (n-i), where i increments until n - i == 1000.
  - if you add up all n and all -i, you get C*n - j, where c is some constant > or = 0, and j is the sum of all i values. This means the recursive function will simplifiy to n for very large n.
- Big O() = O(n).

(e)
```
int verysimple (int n)
{
   if (n <= 1 )
      return 1;
   else
      return n * verysimple (n/2);
}
```

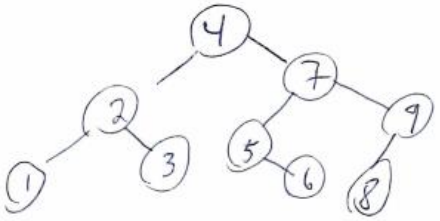
- As in that last case, this is a recursive function, so we can ignore the base case statement in this function and focus on the recursive case.
- This one is rather simple; for each recursive call, we multiply times the n from the previous call, but each call will reduce n by a factor of ½.
- Thus: $n * n/2 * n/4 *... * n/2^j$, where $n/2^j$ is >= 1, with j starting at 0 and incrementing by 1 for each call; we see that n is repeatedly divded by 2 until its value is reduced to <= 1.
- Finally, since recursive calls will stop after reaching the base case where $n/2^j <= 1$, we can rearrange the equation and say $n <= 2^j$; now take $\log_2()$ of both sides, we get $j = \log_2 n = j$. Thus; $\log_2 n$ will be approximately the number of recursive calls that will occur with respect to a given input n.
- Big O() = $O(\log_2 n)$

2 (10 points) Binary Search Trees

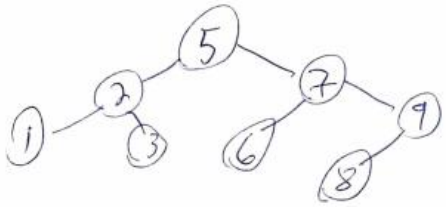
(a) Show a result of inserting 4, 2, 3, 1, 7, 5, 6, 9, 8 into an initially empty binary search tree.

(b) Show the result of deleting the root.

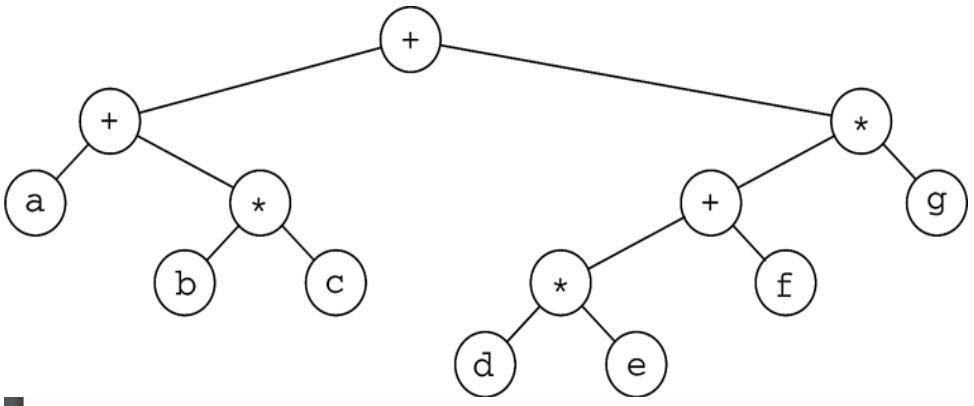
2. a)



b) delete root: overwrite it with largest value in left subtree or smallest in the right subtree. We will the latter since it has more nodes.



3 (6 points) Give the prefix, infix, and postfix expressions corresponding to the following tree:



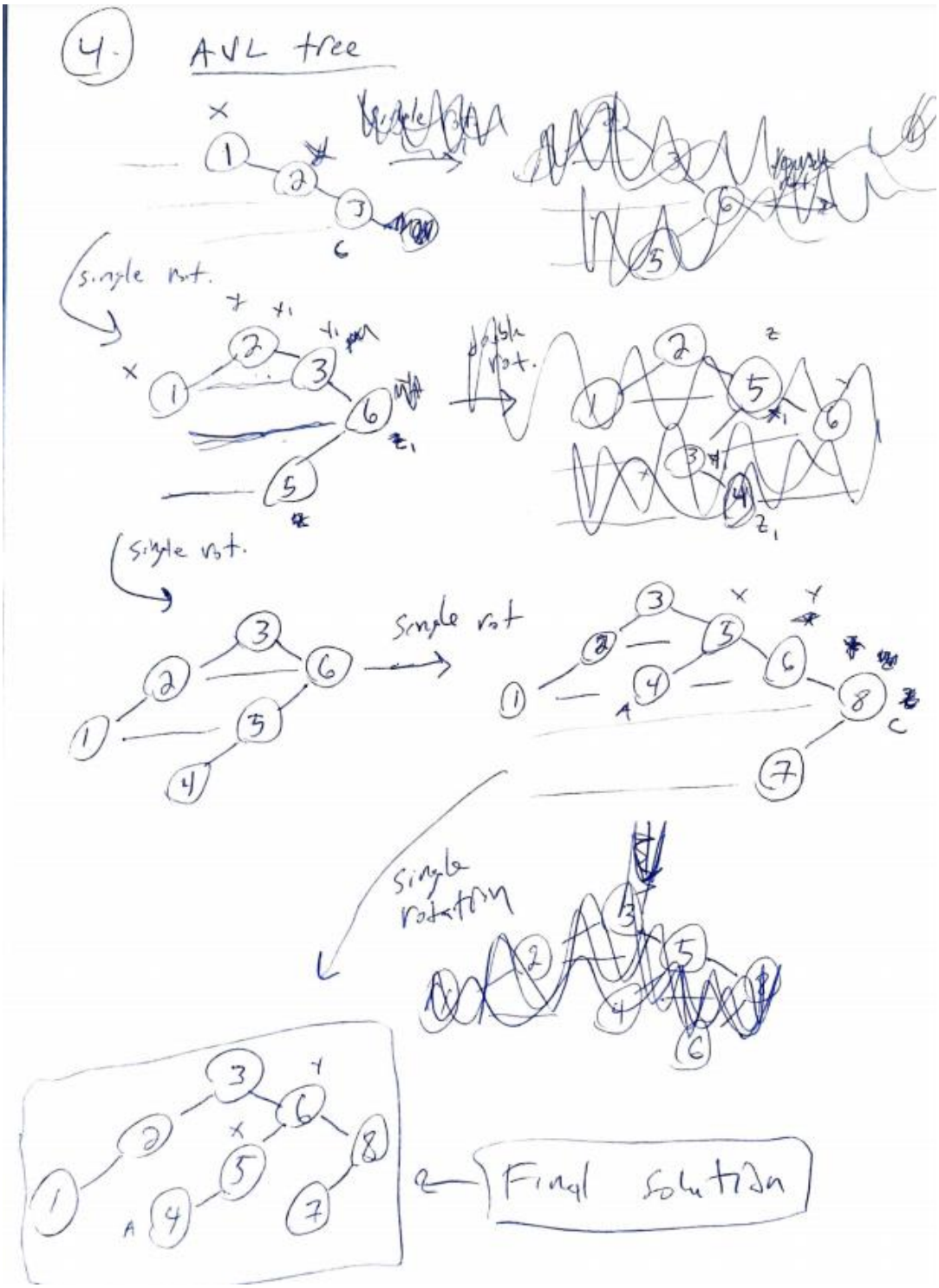
(3.) (preorder) Prefix: + + a * b c * + * d e f g

Infix (in order): a + b * c + d * e + f * g

Postfix (post order): a b c * + d e * f + g * +

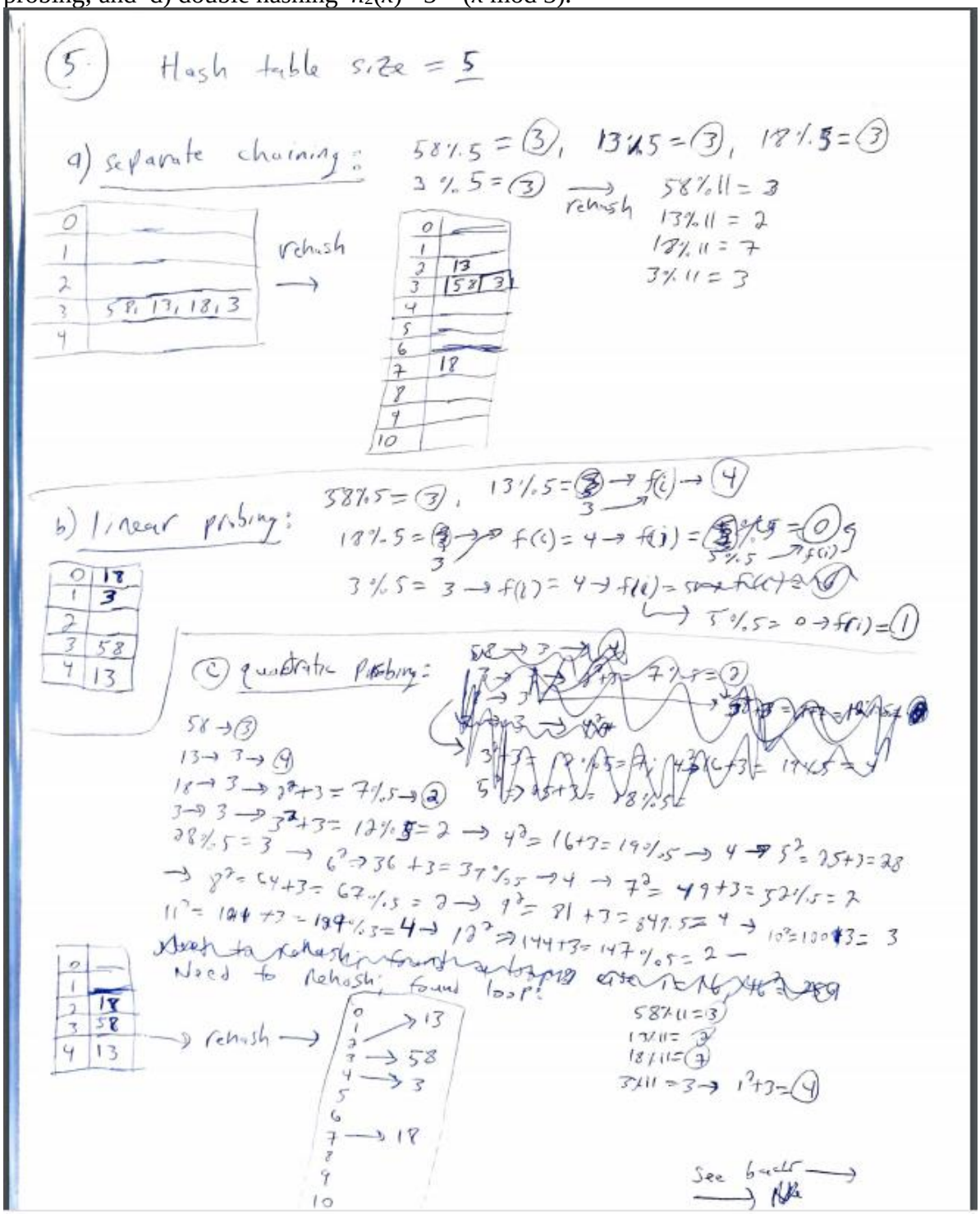
4 (8 points) AVL Trees

Show a result of inserting 1, 2, 3, 6, 5, 4, 8, 7 into an initially empty AVL tree.



5 (12 points) Hashing

Draw the result of hashing 58, 13, 18, 3 into a table if the $h(x) = x \bmod 5$. The initial hash table size is 5. If you cannot fit everything into the table (when a collision chain is longer than 4), then you have to rehash (reset the table size to 11 and update $h(x) = x \bmod 11$), but show both tables. For collisions use: a) separate chaining; b) linear probing; c) quadratic probing; and d) double hashing $h_2(x) = 3 - (x \bmod 3)$.



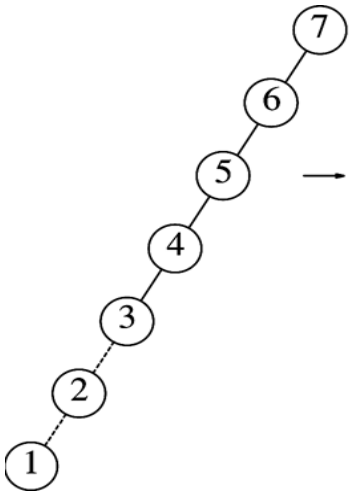
d) double hash:

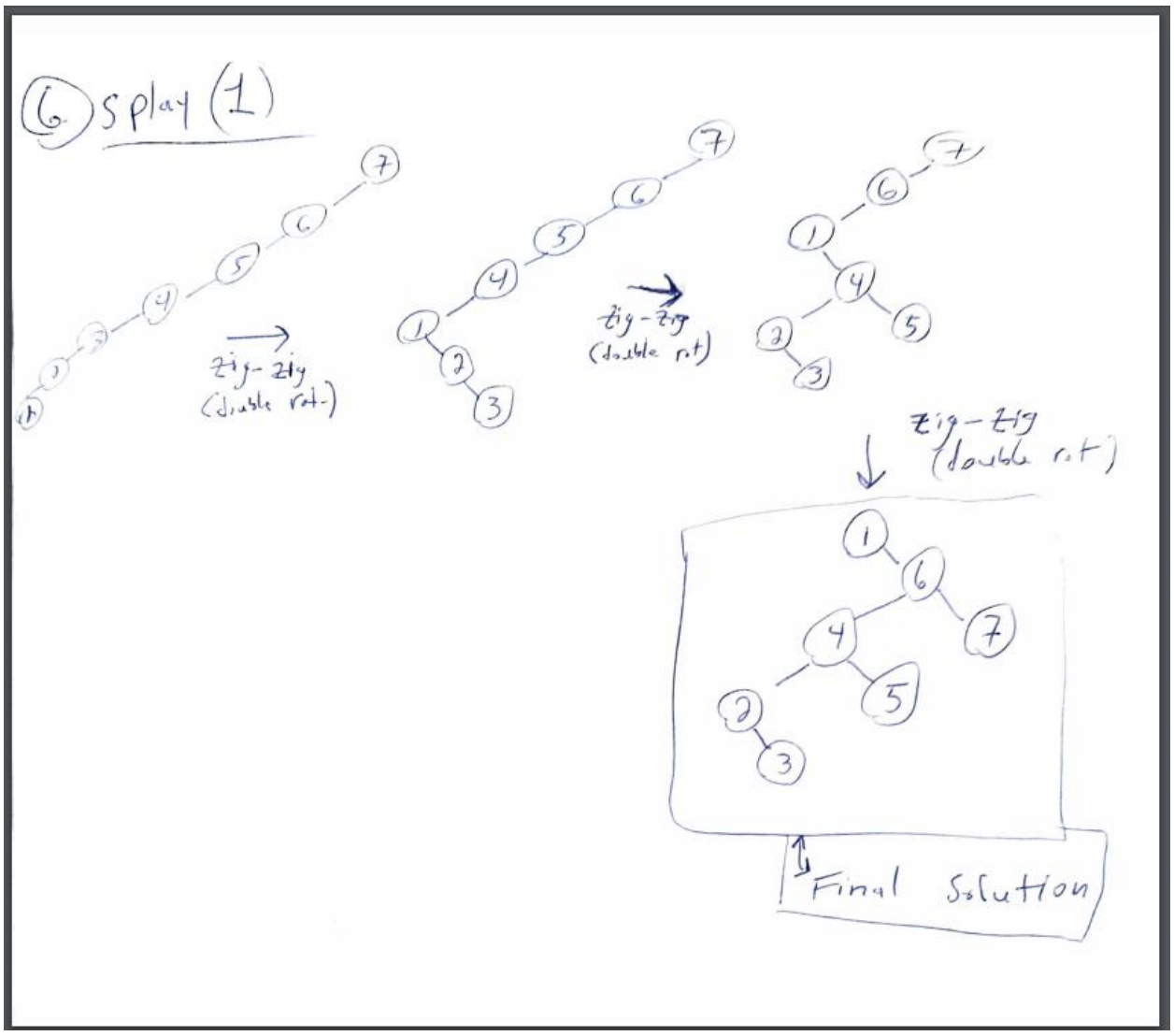
58 → (3) →
13 → 3 → 3 + (3 − (13 % 3)) = 3 + 2 = 5 → 5 − 1 = 5 → (0)
18 → 3 → 3 + 2(3 − (18 % 3)) = 7 % 5 = (2)
3 → 3 → 3 + 3(3 − (3 % 3)) = 3 + 6 = 9 % 5 = (4)

0	13
1	
2	18
3	58
4	3
...	

6 (6 points) Splay Tree

Show the tree after the search for key 1 in the following splay tree:





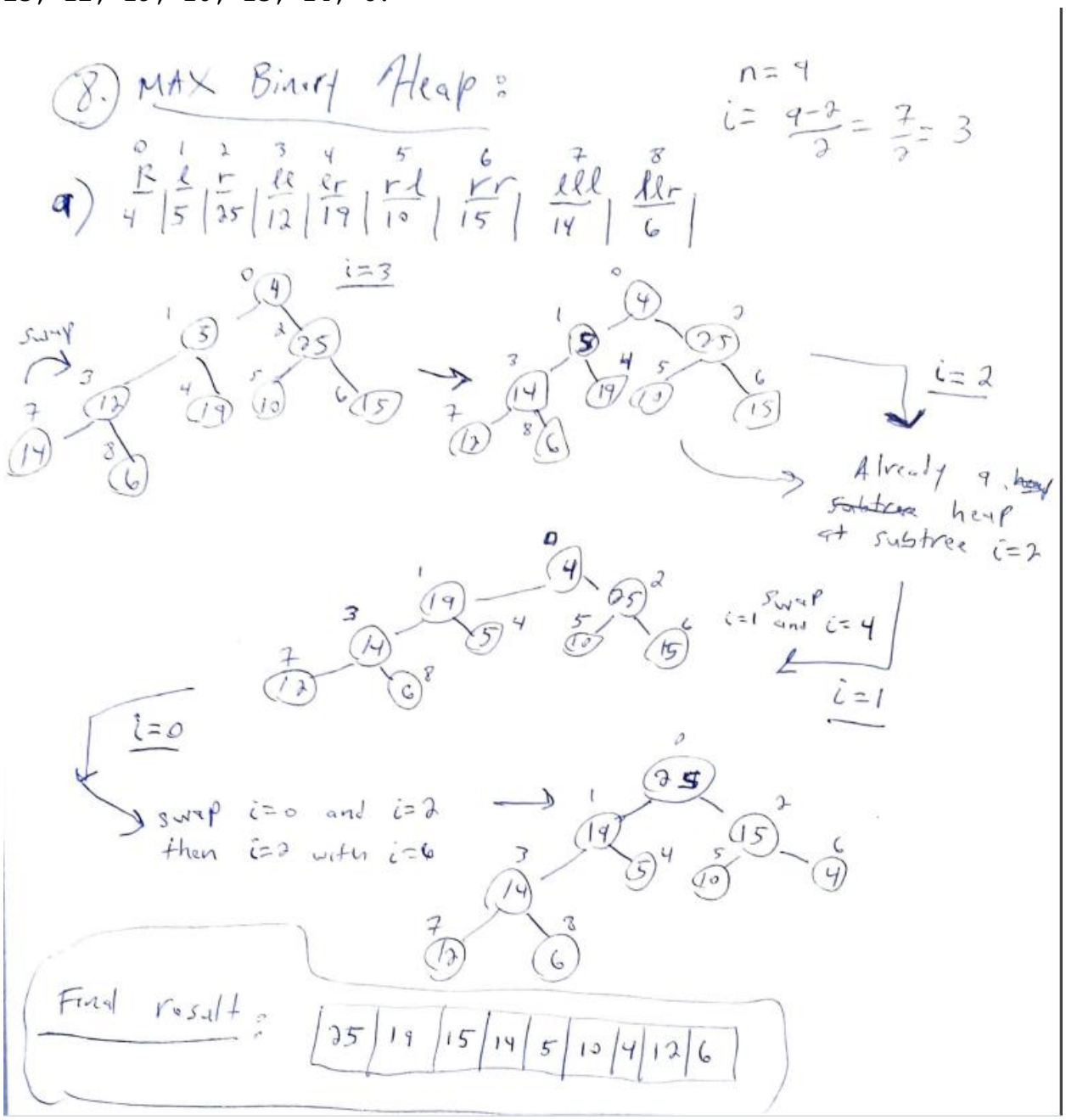
7 **(3 points)** What is the main difference between a B-tree and a Binary Search Tree? What is the main difference between a B-Tree and a B+ Tree? What are the main applications of B+ tree?

The primary difference between B-tree and a Binary Search Tree is the number of children aloud per parent node; binary search trees only allow a maximum of two children per node and allow for 1 or 0 children (a leaf!), while B-trees allow for a predefined (based on implementation methods) maximum and minimum number of children per parent node, thus, they are not limited in their width at any given level (other than the root of course).

The primary difference between B-trees and B+ Trees is that of a slight implementation difference; they are both sorted trees with sorted search keys and function exactly the with the exception of B+ Trees do not store any data in their internal nodes but rather contain only the pointers to an array of search keys; the leaves are the array which contain the search keys and which also do not store data but also pointers to data records. The main application of a B+ tree is data base management systems in order to minimize disk reads and be able to search a large range of disk space since no data is stored in the search tree, but only pointers to the contiguous blocks of memory that may be searched for.

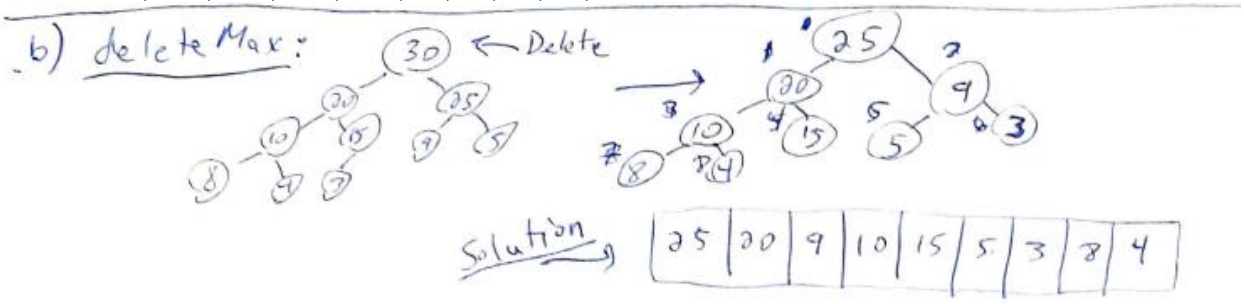
8. (12 points) Max Binary Heap

a) Show the results of using the **linear-time algorithm** to build a max binary-heap using the input: **4, 5, 25, 12, 19, 10, 15, 14, 6.**



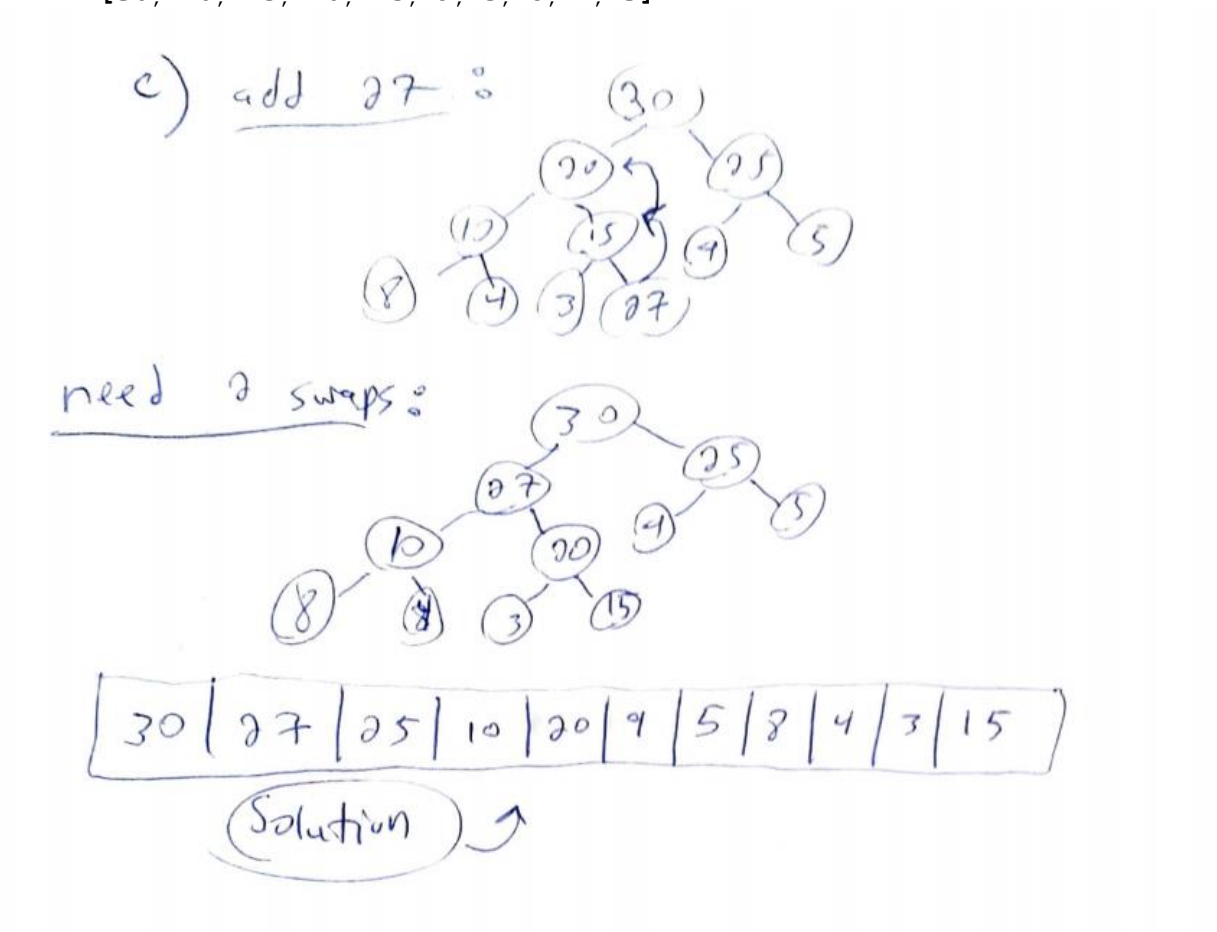
b) Show the result of **deleteMax** to the following ***max*** binary heap.

`[30, 20, 25, 10, 15, 9, 5, 8, 4, 3]`



c) Show the result of adding 27 to the following ***max*** binary heap.

`[30, 20, 25, 10, 15, 9, 5, 8, 4, 3]`



9. (8 points)

a) Write a recursive function that take only a pointer to the root of a binary search tree, BST, and print all the items in BST in the **decreasing** order.

//I wrote it in c++ just so it is easier for me to see the loops and functions…etc…:

```
struct BinaryNode
{
  int element;
  BinaryNode *left;
  BinaryNode *right;
};
//NOTE: I CHANGED RETURN TO VOID BECAUSE I SAW NO REASON TO USE A RETURN VALUE.
void printTreeDecreasingOrder(BinaryNode* node, ostream& out)
{
	//first we handle right sub trees (larger values)

	if(node->right != NULL) { //use this loop to move to the right-most (largest
element) node in a given sub tree

		printTreeDecreasingOrder(node->right, out);

	}
	out << node->element; //right is NULL; now we can print current element.


	//now we need to handle left sub trees (smaller values):

	if(node->left != NULL){

		printTreeDecreasingOrder(node->left, out); //this will automatically check
right sub trees of the left subtrees when the function is called again and starts from
first if-statement.

	}



}
```

b) What is the running time of your routine?

Run time of my routine is **O(N)** since every node must be visited in order to print out all elements; however since it is a BST it is what reduces the function to only O(N) since elements are already sorted, and I take advantage of the binary characteristics of a BST for the recursive function calls. The space for this function however is very inefficient since if we have a tree with a very large depth, many recursive function calls will need to be called and placed on stack at the same time.