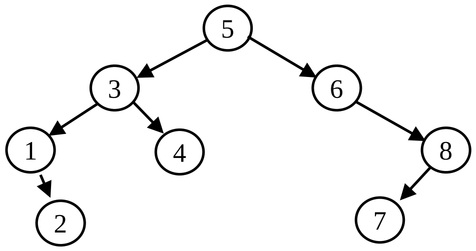
Section 1: True / False (1 point each, 15 pts total)

Circle the word TRUE or the word FALSE. If neither is circled, both are circled, or it impossible to tell which is circled, your answer will be considered wrong. For the *O*() questions, we are looking for tightest bounds.

1. [✓ TRUE] or [✓ FALSE]
   Quadratic probing is a technique employed by aliens to experiment on gullible human beings.
2. TRUE or [✓ FALSE]
   Inserting a value in a binary search tree is worst-case $O(\log n)$.
3. [✓ TRUE] or FALSE
   Searching for a value in a red-black tree is worst-case $O(\log n)$.
4. TRUE or [✓ FALSE]
   Deleting a value in a splay tree is worst-case $O(\log n)$.
5. [✓ TRUE] or FALSE
   Deleting the largest value in a min heap is worst-case $O(n)$.
6. TRUE or [✓ FALSE]
   Insertion into a hash table using chaining is worst-case $O(n)$
7. [✓ TRUE] or FALSE
   A black node in a red-black tree can simultaneously have both a black child node and a red child node
8. TRUE or [✓ FALSE]
   A polynomial hash code will map string anagrams to the same key.
9. TRUE or [✓ FALSE]
   A red-black tree cannot have more red nodes than black nodes, not counting dummy leaf nodes.
10. TRUE or [✓ FALSE]
    A single insertion in a AVL tree might require multiple tree restructurings to restore balance
11. TRUE or [✓ FALSE]
    The height of the left subtree of a leftist heap is always greater than or equal to the height of the right subtree.
12. TRUE or [✓ FALSE]
    A heap is a type of binary search tree
13. [✓ TRUE] or FALSE
    In a min heap, all the values in the left subtree of the root can be smaller than all the values in the right subtree.
14. TRUE or [✓ FALSE]
    Given distinct keys, a hash function must always produce different indices.
15. TRUE or [✓ FALSE]
    A hash table can guarantee $O(1)$ (i.e., constant-time) performance on some operations.

Section 2: Short Answer (varying point values. 70 pts total)

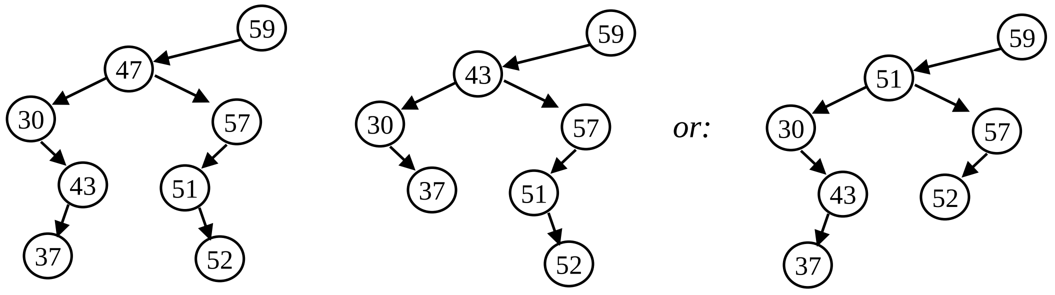
16. (**4 points**) Draw the binary search tree that would result from inserting the following values, in the exact order given (assume you began with an empty tree): **5, 6, 3, 8, 7, 4, 1, 2**



17. (**5 points**) Describe one advantage and one disadvantage of a leftist heap versus a min-heap:

Advantage: can do efficient merge
Disadvantage: space efficiency worse—must be a real tree instead of a simple array

18. **(5 points)** Describe *in words* the process of removing 47 from the following binary search tree; then show what the resulting tree would look like.



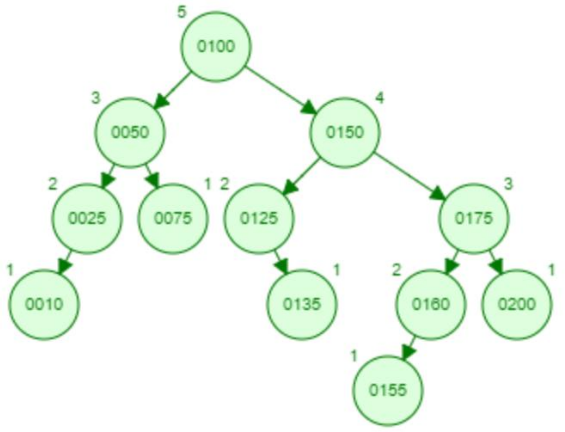
Since 47 has 2 children, we replace it w/either the max of left subtree, or min of right, and delete that other node. So, replace 47 w/43, then delete old 43, making its child 37 a child of its parent 30 (or repl. 47 w/51, del 51, make 52 child of 57.

19. (**3 points**) Given a complete binary search tree of size *n*, how many different ways are there to store the key values from 1 to *n* in the nodes? (You can give your answer as an algebraic expression if desired.)

`1`

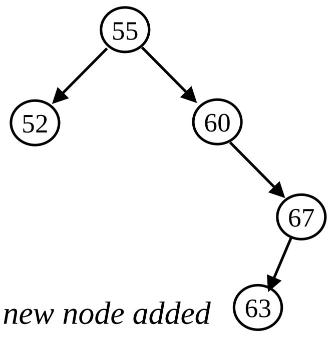
20. (**4 points**) Draw an AVL tree in which two of the leaves differ in depth by at least 2 (make sure to indicate the two leaves you are comparing):

`Leaves 75 and 155 differ in depth by 2`
(this is just one example)

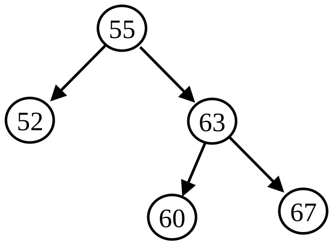


21. (**4 points**) Insert the key 63 into the following AVL tree; first, show where it would be inserted in the original tree (indicate the position on the given tree), then draw in the box to the right the resulting tree after rebalancing.

Show insertion location here: *...and draw the rebalanced tree here:*



new node added



double rotation done

22. **(3 points)** In what sense is a red-black tree considered a "balanced tree"? Give a complete and precise explanation (your explanation must take red as well as black nodes into account):
`All of the leaves must have the same black depth, and because of the "no two reds in a row" rule, this means the deepest leaf can be no more than twice the depth of the shallowest leaf`

23. **(4 points)** List the 4 properties of a red-black tree:

1. **Root of tree is black**

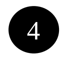
2. **All leaves are black**

3. **Children of a red node are black**

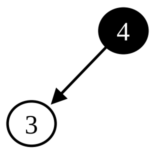
4. **All leaves have the same black depth**

24. **(8 points)** Insert the values 4, 3, 2, and 1 (in that order) into an empty red-black tree. (Make sure it is clear in your trees which nodes are black and which are red.)

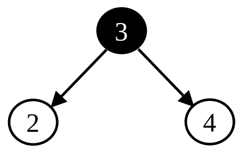
a) after inserting 4:



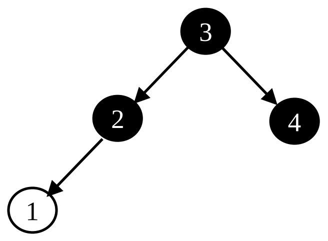
b) then inserting 3:



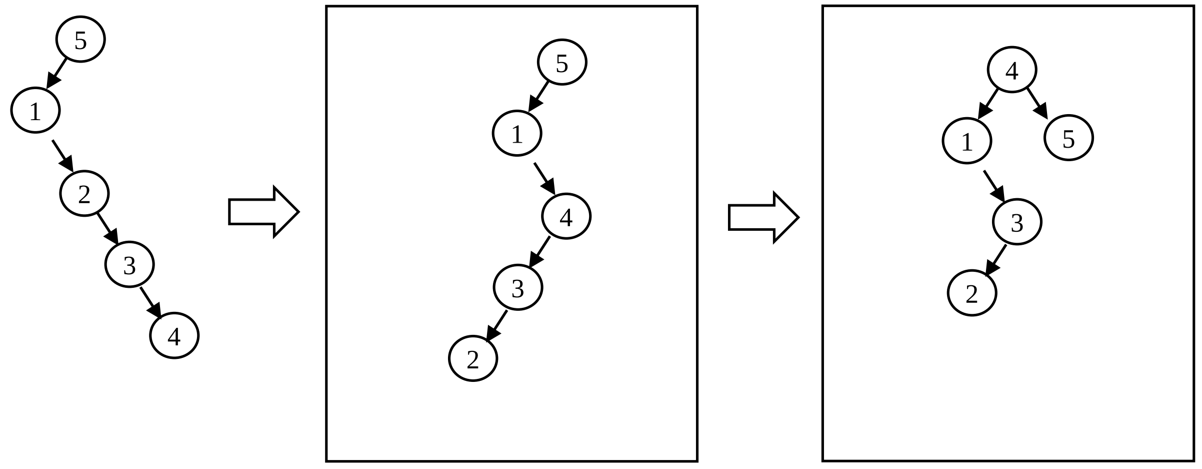
c) then inserting 2:



d) finally, inserting 1:



25. **(6 points)** A user searches for the key 4 in the following splay tree, triggering a splay operation. Draw the results after each zig-zig, zig-zag, or zig operation (a combination of two such operations are necessary, so draw 2 more trees):



26. **(4 points)** Explain what a min-heap is, including its structure, elements, and the relationship among the elements. Be detailed in your explanation.

    `A min-heap is a binary tree where a node is always smaller than its children`

27. **(4 points)** Describe the steps involved in inserting a new value in a min-heap (if you use the term "bubble/trickle up", you must explain it):

    `The new value is added to right of last item in last level` *2 pts*
    `The new node is "bubbled-up"—swapped with its parent—as long as it is smaller than the parent` *2 pts*

28. **(4 points)** How would you find the *maximum* (not minimum!) value in a min-heap? What would the worst-case runtime be (in big-O terms), and why?
    `You would have to search all the leaves looking for the smallest. This is` ***O*(*n*)**, **`because in a binary tree, half the nodes are leaves.`**

29. **(2 points)** How would you find (without removing) the maximum value in a ***min-max*** heap in *O*(1) time? You can assume the tree has at least 2 completely full levels.

    `You would just take larger of the two children of the root`

30. **(6 points)** You are given a hash table of size 7. Your hash function is simple modulo-7 remainder, and you are using linear probing to deal with collisions. Show the hash table that would result from inserting the following values, in the order given: 10, 7, 12, 17, 26, 3

7	3	-	10	17	12	26
0	1	2	3	4	5	6

31. **(4 points)** List two important properties of a good hash function, explaining what/why:

    `- It must spread out the values`
    `- It must be efficient.`

Section 3: Design and Coding (15 points total)

Note: for this section, you are being asked to write more substantial sections of code. Clarity of code structure and efficiency of design count

32. **(15 points)** You are given a hash table, stored in the array `hashTable`, with the size of the table stored in the variable `hashSize`. All inserted keys will be positive (i.e., > 0), and the hash table is initialized to all 0's. Assume the hash function has already been coded up for you, in the function `hashFunc()`. Your hash table must use linear probing to resolve collisions. Given the following implementation of the `delete()` function, write the code for the companion function `insert()`, with the signature and functionality given below. Note that you must correctly handle lazy deletes, in a way compatible with the provided `delete()`

```
int  *hashTable;  // Actually, a pointer to an array, allocated elsewhere
int  hashSize;   // The size of the hashTable array
int  hashFunc(int key);  // code written elsewhere

// Function to remove key, if it exists, from hash table.
// Returns true if key found and deleted; else, returns false.
bool delete(int key) {
    int hashPos = hashFunc(key);
    for (int i = 0; i < hashSize; i++) {
        int pos = (hashPos + i) % hashSize;
        if (hashTable[pos] == key) {
            hashTable[pos] = -1;
            return true;
       }
        else if (hashTable[pos] == 0)
            break;
    }
    return false;
}

// Function to add new value to hash table. Can assume value isn't
// already in the table, and that table is not full.
void insert(int key) {
    // YOUR CODE HERE



    int hashPos = hashFunc(key);

    for (int i = 0; i < hashSize; i++) {
        int pos = (hashPos + i) % hashSize;
        if (hashTable[pos] <= 0) {
            hashTable[pos] = key;
            return true;
       }
        else if (hashTable[pos] == 0)
            break;
    }



}
```

[Continue your answer for question 32 here if you need more space]