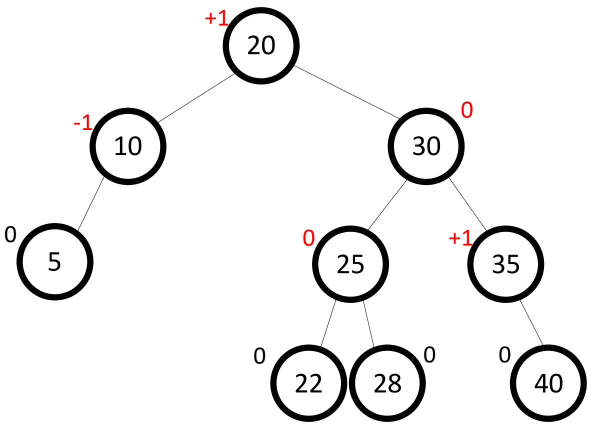
CS104: Data Structures & Object-Oriented Programming

Summer 2021 - Final Exam Sample Problem Solutions

1. **(12 pts.) AVL trees:** Consider the AVL tree shown below.



1.1. Choose the correct balance for the node with key **10**? **[-2 / [✓ -1] / 0 / +1 / +2**].

1.2. Choose the correct balance for the node with key **20**? **[-2 / -1 / 0 / [✓ +1] / +2**].

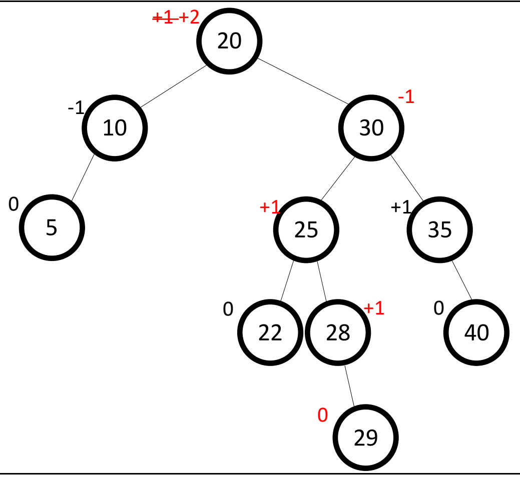
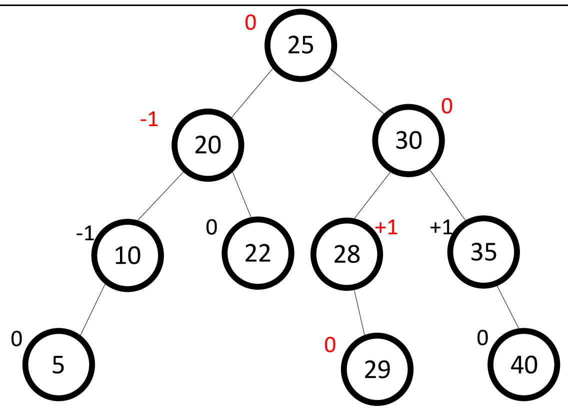
1.3. Choose the correct balance for the node with key **25**? **[-2 / -1 / [✓ 0] / +1 / +2**].

1.4. Choose the correct balance for the node with key **30**? **[-2 / -1 / [✓ 0] / +1 / +2**].

1.5. Choose the correct balance for the node with key **35**? **[-2 / -1 / 0 / [✓ +1] / +2**].

1.6. Which node, if any, when removed would necessitate a rotation(s)? [List the node's key or type "none" if none exists]. **5 or 10**

1.7. Show the process for **insert(29)** including where it is added, updates of any balance values, and the result of any necessary rotations. You can draw the tree at a few points in time during the insert process and need only draw the relevant parts of the tree (though you must show the relevant parts). Upload a PDF or image of your work.

 ~~+1~~ +2 20 -1 10 30 -1 0 5 +1 25 +1 35 0 22 28 +1 0 40 0 29	 0 25 -1 20 30 0 -1 10 0 22 28 +1 +1 35 0 5 0 29 0 40
• Show correct insertion point of 29 • Can earn partial credit if they show intermediate updates to parent balance values up to the root before the rotations (this is not necessary to get full credit, but if the rotation is wrong, you can give 1 point partial credit for these balance values)	• Correct zig-zag rotation on 20, 30, 25 ○ 25 at top ○ 20 to left ○ 30 to right ○ 28 and 29 as left subtree of 30 ○ 22 as right subtree of 20 ○ Correct balance values

2. **(10 pts.) Counting**: **For all 3 problems below,** Julia has **EXACTLY 6 hours** to binge-watch some episodes for various shows. You must show work that supports your final answer to get any credit. You can either upload your work as a scan/picture OR just type it in using a text-based approach (i.e. nCk for n Choose k, nPr for n Permute r, 3^2 for 3 squared, etc.)

   2.1. Assuming that for a given show she always watches episodes in sequential order (never out of order), how many orderings exist for her to watch **8 half-hour episodes from show A** and **2 one-hour long episodes from show B** (assuming she can switch back and forth between shows as she pleases).

- Assuming that for a given show she always watches episodes in order, how many orderings exist for her to watch 8 half-hour episodes from show A and 2 one-hour long episodes from show B.
    - 10 C 2 (or 10 C 8)

   2.2. Now assume she (strangely!) is willing to watch the episodes of the same show in ANY order (not sequentially). How many orderings exist of the ten total episodes described in the previous part of the problem.

- Now assume she (strangely!) is willing to watch the episodes of the same show in ANY order. How many orderings exist of the ten total episodes described in the part i of the problem.
    - 10!

   2.3. Assume she again watches episodes in (sequential) order. There are two unique shows (C and D…forget about A and B for this question) where one has at least **12 half-hour** long episodes and the other has at least **3 two-hour** long episodes which she can watch in any (mixed) order. How many orderings exist of the episodes that she will watch?

- Assume she again watches episodes in (sequential) order. There are two unique shows (C and D) where one has at least 12 half-hour long episodes and the other has at least 3 two-hour long episodes. How many orderings exist of the episodes that she watches?
    - Break into cases:
        - 0 2 hour shows: (12 C 12) = 1 way (all half-hour shows)
        - 1 2 hour show: (9 C 1) = 9 ways (choose where the 2 hour show is watched)
        - 2 2 hours shows: (6 C 2) = 15 ways (choose where the 2 hours shows are watched out of the 6 total episodes)
        - 3 2 hour shows: (3 C 3) 1 way (all 2 hour shows)

        = 1 + 9 + 15 + 1 = 26 orderings

3. **(10 pts.) Number Theory** - Show your work in the same manner and with the same requirements as the previous problem.

3.1. Find the **`gcd(396,168)`** using Euclid's algorithm (showing your work and demonstrating your understanding of the algorithm.

5.1: gcd 12

- 396 = 2*168 + 60
- 168 = 2*60 + 48
- 60 = 1*48 + 12
- 48 = 4*12 + 0

3.2. Find the smallest two integer such that **`396x + 168y = gcd(396,168)`** demonstrating you understand the method taught in class.

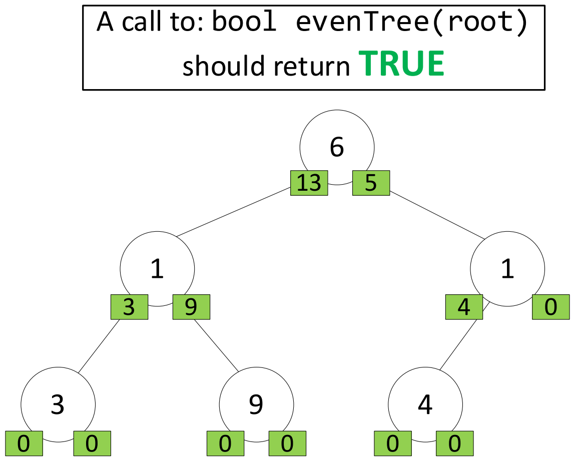
5.2: Answer is x = 3, y = -7 => 3 * 396 - 7 * 168 = 12 = gcd(396,168)

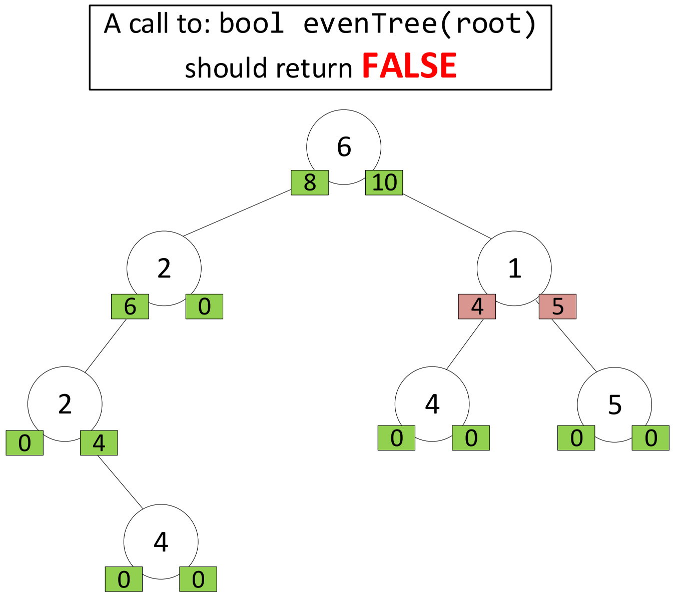
- 60 = 396 - 2*168
- 48 = 168 - 2*60
- 12 = 60 - 1*48

Let r0 = 396 and r1 = 168

- 60 = r0 - 2*r1
- 48 = r1 - 2*60 = r1 - 2*(r0-2*r1) = 5r1 -2r0
- 12 = 60 - 1*48 = r0 - 2r1 - (5r1-2r0) = 3r0 - 7r1 => x=3, y = -7

4. **(10 pts.) Trees and Recursion**. Given a binary tree of **non-negative integers**, write a **recursive** function that returns TRUE if the sum of all nodes in the left and right subtree of **EACH** node is an **even number**. If the sum of nodes in the left and right subtree of even 1 node is odd, return false. Non-existent nodes have an implicit value of 0. Also, an empty tree should return \`true\`. No loops are allowed so you must use recursion.





Complete your code in the linked [eventree.cpp]() and submit that file. A portion is reproduced below.

```
// You may NOT change this struct definition
struct Node {
    int val;
    Node *left, *right;
};

// You may NOT change this function prototype
bool evenTree(Node* root);

// You may add any prototypes of helper functions here


// Now implement the evenTree function and any necessary helpers
// -------------------- YOUR CODE HERE  ------------------
```

```
//-------------- Solution -----------------------

bool evenTree(Node* root)
{
    bool res = true;
    evenTreeHelper(root, res);
    return res;
}

int evenTreeHelper(Node* root, bool& result)
{
    if(root == nullptr) return 0;
    int sum = evenTreeHelper(root->left, result) +
              evenTreeHelper(root->right, result);
    if( sum % 2 == 1)
    {
        result = false;
    }
    return sum + root->val;
}
```

5. **Hashing (10 pts.)**

Consider a hash table that uses **open-addressing** (some form of probing) and the same table structure (vector of HashItem pointers) from homework 6 as well as a similar `HashItem` structure (reproduced below for your benefit). Now write a function that will **compute how many buckets** of the hash table would be **empty** if **chaining / closed-addressing** had been used instead (for the same table size).

You do not necessarily need to convert the table to use chaining, simply calculate how many locations / buckets would be **empty**. Your function must run in **Theta(m)** time where \`m\` is the table size. You may declare other arrays / data structures as needed.

Complete your code in the linked **chains.cpp** and submit that file. A portion is reproduced below.

```
// HashItem definition used by the hash table
template <typename KeyType, typename ValueType>
struct HashItem {
    typedef std::pair<KeyType, ValueType> ItemType;
    ItemType item;
    bool deleted;
};

/**
 * @brief FUNCTION YOU NEED TO WRITE
 * Computes the number of empty buckets that would exist if
 * the hash table used chaining rather than open-addressing (probing).
 *
 * @param [in] table Hash table of pointers to HashItems. An entry
 *  is occupied if the pointer is non-NULL
 * @param [in] h Hash function that supports operator()(Key) and returns
 *   an unsigned integer of arbitrary size
 * @return the number of buckets/chains that would be empty had chaining
 *   been used rather than open-addressing
 */
template<typename HashItem, typename Hash>
size_t numEmptyIfChaining(const std::vector< HashItem* >& table, Hash h);
```

```
template<typename HashItem, typename Hash>
size_t numEmptyIfChaining(const std::vector< HashItem* >& table, Hash h) {
    size_t m = table.size();
    size_t counter = 0; // num chains  not used for method 1 and
                        // chains used for method 2
    // create our own hash table of bools
    std::vector<bool> tracker(table.size(), false);

    // Iterate through hash table - Theta(m)
    for(auto p : table){
        if(p != nullptr  &&  p->deleted == false){
            // hash the key
            unsigned int loc = h(p->item.first) % m;
            // set that location to true to indicate it would be non-empty
            tracker[loc] = true;
        }
    }
    for( auto b : tracker){
        if( b == false) {
            counter++;
        }
    }
    return counter;
}

template<typename HashItem, typename Hash>
size_t numEmptyIfChaining(const std::vector< HashItem* >& table, Hash h) {
    size_t m = table.size();
    size_t counter = 0; // num chains  not used for method 1 and
                        // chains used for method 2
    // create our own hash table of bools
    std::vector<bool> tracker(table.size(), false);

    // Iterate through hash table - Theta(m)
    for(auto p : table){
        if(p != nullptr  &&  p->deleted == false){
            // hash the key
            unsigned int loc = h(p->item.first) % m;
            // set that location to true to indicate it would be non-empty
            if(tracker[loc] == false) {
                counter++;   // another new chain being used
                tracker[loc] = true;
            }
        }
    }
    return m - counter;
}
```