# Midterm Review Sheet: CS15 Spring 2025

## About the Exam

- Covers all material through Huffman Coding

## About this Review Sheet

- **This is not a practice exam.** The length of this review and the types of questions asked do not necessarily reflect what the actual exam will look like.
- This review *does* give you a chance to practice and apply your knowledge

## Topics Covered so far (no particular order)

- Data Structures and ADTs
  - Array Lists
  - Linked Lists
  - Stacks/Queues
  - Sets
  - Trees
    - N-ary Trees
    - Binary Trees
    - Binary Search Trees
    - AVL Trees
- Complexity
- Lists
  - Array Lists
  - Linked Lists
- Problem Solving
  - Recursion
- Huffman Coding

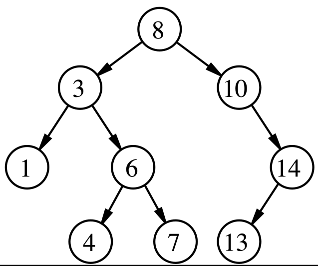
## Practice Problems

1. Time Complexity

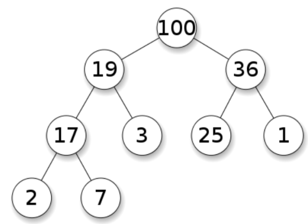
<table>
<tr><td>$$T(n) = n^2 + \frac{3}{2} n \log n + 3$$</td><td>Solution:<br>$$O(n^2)$$</td></tr>
<tr><td><pre>for (int i = 0; i < n; i++) {
        for (int j = 0; j < 7; j++) {
                // constant work
        }
}</pre></td><td>Solution:<br>$$n * 7 = 7n = O(n)$$</td></tr>
<tr><td><pre>for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                // constant work
        }
}</pre></td><td>Solution:<br>$$n * n = n^2 = O(n^2)$$</td></tr>
<tr><td><pre>for (int i = 0; i < n * n; i++) {
        for (int j = 0; j < n; j++) {
                // constant work
        }
}</pre></td><td>Solution:<br>$$n * n * n = n^3 = O(n^3)$$</td></tr>
<tr><td><pre>for (int i = 0; i < n * n; i++) {
        for (int j = 0; j < 5n - 3; j++) {
                // constant work
        }
}</pre></td><td>Solution:<br>$$n * n * (5n - 3) = 5n^3 - 3n^2 = O(n^3)$$</td></tr>
<tr><td><pre>for (int i = 0; i < 2 * n; i++) {
        for (int j = 0; j < i; j++) {
                // constant work
        }
}</pre></td><td>Solution:<br>$$0 + 1 + \ldots + 2n = \sum_{j=0}^{2n} j = 2n^2 + n = O(n^2)$$</td></tr>
</table>

2. To Tree or Not To Tree

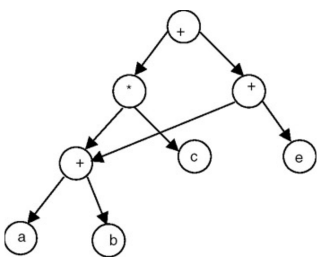
   Given the following linked structures, determine whether it is a **Tree, Binary Tree, or Binary Search Tree**. Be as **specific** as possible; if a structure is both a BST and a Binary Tree the answer would be BST. If, the structure is none of the above, then say **Not a Tree**.



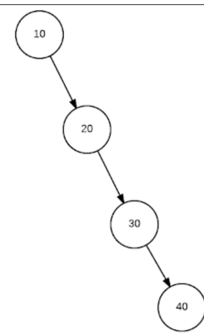

**Solution:** Binary Search Tree




**Solution:** Binary Tree




**Solution:** Not a tree




**Solution:** Binary Search Tree

*empty tree*

**Solution:** An empty tree satisfies all BST properties

3. Miscellaneous

   For each of **Unsorted Array, Sorted Array, Unsorted Linked List, Sorted Linked List**, say whether binary search can be performed. Justify your answer.

**Solution:** Binary search can only be performed on a sorted sequence, so neither the unsorted linked list nor the unsorted array are possible. Binary search also depends on constant time access to elements in the sequence, so neither linked list can be used. This leaves the Sorted Array as the only viable choice.

4. Zip

**Write a function that *zips* two linked lists.** The zip operation creates a list by inserting nodes from two other lists in alternating positions.

$A = 5 \rightarrow 7 \rightarrow 17$

$B = 12 \rightarrow 10 \rightarrow 2 \rightarrow 4 \rightarrow 6$

$Zip(A, B) = 5 \rightarrow 12 \rightarrow 7 \rightarrow 10 \rightarrow 17 \rightarrow 2 \rightarrow 4 \rightarrow 6$

**Note that the extra nodes of the longest list were appended to the end**

```
struct Node {
        DataType data;
        Node    *next;
};
/**
 * @brief      "Zips" the two lists beginning by a and b
 *
 * @param      a     The front of list a
 * @param      b     The front of list b
 *
 * @return     The pointer to the head of the zipped list
 *
 * @note       This operation will be done *inplace*.
 *             That is, there should be no dynamic memory
 *             allocation (new) or memory deallocation (delete)
 */
Node *zip(Node *a, Node *b) {
```

**Solution:**

```
// If either list is empty, return the other list
if (a == nullptr)
        return b;
if (b == nullptr)
        return a;

Node *next_a = a->next;
Node *next_b = b->next;

a->next = b;
b->next = zip(next_a, next_b);
return a;
```

```
}
```

5. Stacks and Queues.

Implement a Queue using 2 Stacks.

```
struct Queue {
    stack<int> s1, s2;
    void enqueue(int i);
    int  dequeue();
};

void Queue::enqueue(int i) {
```

**Solution:**

```
// Move all elements from s1 to s2
while (not s1.empty()) {
        s2.push(s1.top());
        s1.pop();
}
// Push item into s1
s1.push(i);
// Push everything back to s1
while (not s2.empty()) {
        s1.push(s2.top());
        s2.pop();
}
```

```
}

int Queue::dequeue() {
```

**Solution:**

```
// Dequeue an item from the queue
// if first stack is empty
if (s1.empty())
        throw runtime_error("empty_queue");
// Return top of s1
int i = s1.top();
s1.pop();
return i;
```

```
}
```

6. N-ary Trees

   (a) Write a recursive function that recursively counts the number of leaf nodes in the tree.

```
int Tree::countLeaves(Node* curr) {
```

**Solution:**

```
if (curr == nullptr)
      return 0;
if (curr->isLeaf())
      return 1;

int numLeaves = 0;
for (int i = 0; i < curr->numChildren; i++) {
        numLeaves += countLeaves(curr->children[i]);
}
return numLeaves;
```

```
}
```

   (b) What traversal would you use to delete all nodes in a tree? Why?

**Solution:** Post order, since you want to delete all children before current node is deleted

7. Binary Trees

   (a) Write a recursive that recursively finds the height of a Binary Tree

```
int BinaryTree::height(Node* curr) {
```

**Solution:**

```
if (curr == nullptr)
      return -1;
else
      return 1 + max(height(curr->left), height(curr->right));
```

```
}
```

(b) Write a recursive function that returns whether an element is in a Binary Tree

```
bool BinaryTree::find(Node* curr, int data) {
```

**Solution:**

```
if (curr == nullptr)
      return false;

if (curr->data == data)
      return true;

return find(curr->left, data) or find(curr->right, data);
```

```
}
```

8. Binary Search Trees

Write a recursive function that returns whether an element is in a Binary Search Tree

```
bool BinarySearchTree::find(Node *curr, int data) {
```

**Solution:**

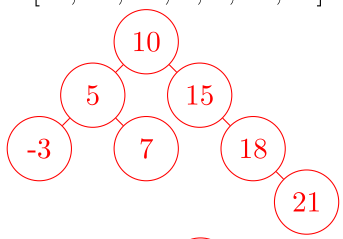
```
if (curr == nullptr)
      return false;
else if (curr->data == data)
      return true;
else if (data < curr->data)
      return find(curr->left, data);
else  // data > curr->data
      return find(curr->right, data);
```

```
}
```

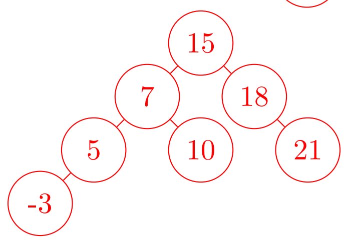
9. AVL Trees

1. Insert the following numbers, in order from left to right, into a Binary Search Tree and an AVL Tree.

   (a) Tree 1: [10, 15, 18, 5, 7, 21, -3]

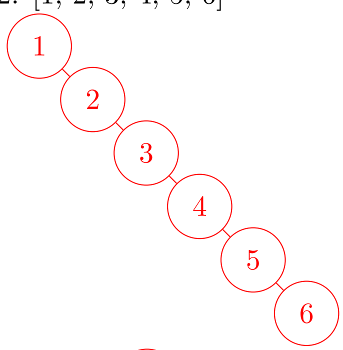
   

   BST:

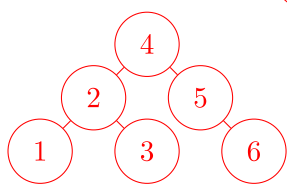
   

   AVL:

   (b) Tree 2: [1, 2, 3, 4, 5, 6]

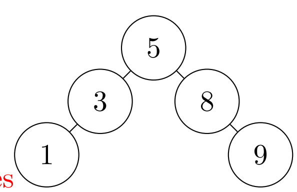
   

   BST:

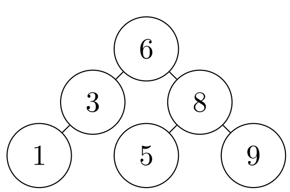
   

   AVL:

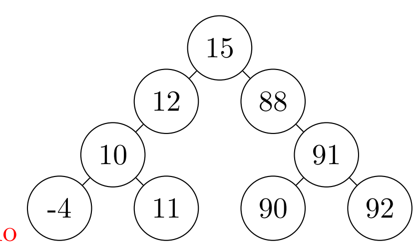
2. Are the following trees valid AVL trees?

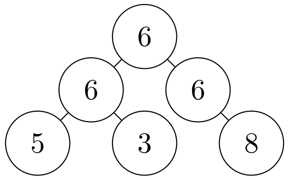
   (a) yes

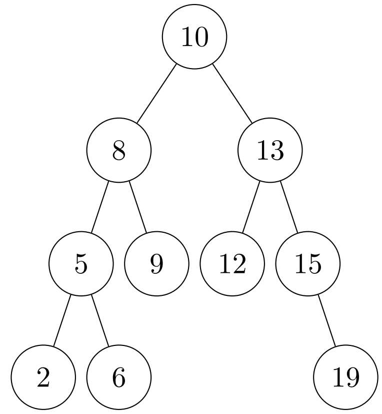
   

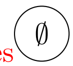
   (b) no

   (c) no

(d) no 


(e) yes 


(f) yes 

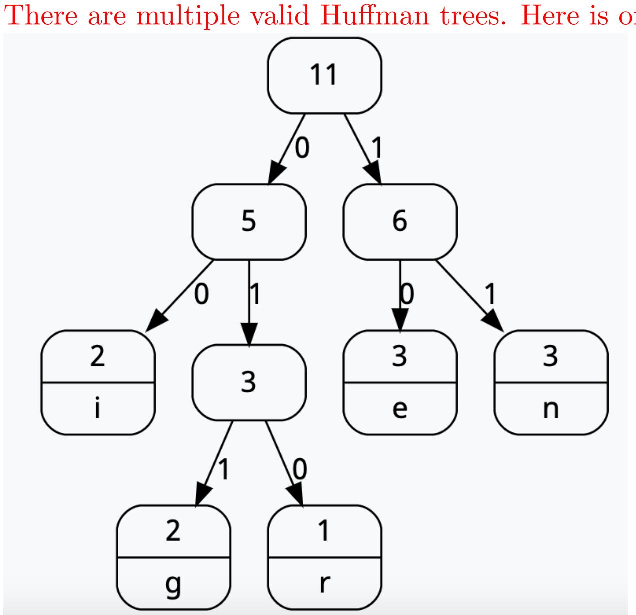

3. Runtime Questions: Fill in the worst-case runtime in big-O notation; then for all the below, describe the tree that gives the worst case runtime.

| | BST | AVL Tree |
|---|---|---|
| insert | O(n) - Tree from 1b | O(logn) - any AVL b/c balanced |
| delete | O(n) - Tree from 1b | O(logn) - any AVL b/c balanced |
| find | O(n) - Tree from 1b | O(logn) - any AVL b/c balanced |
| findMax | O(n) - Tree from 1b | O(logn) - any AVL b/c balanced |
| findMin | O(n) - Tree from 1b (reversed) | O(logn) - any AVL b/c balanced |

10. Huffman Coding

(a) Draw a Huffman tree for the word "engineering".

There are multiple valid Huffman trees. Here is one:




(b) Using tree you created in part (a), what is the size of the encoding for the word "engineering"?

25 bits (this is the answer for any valid Huffman tree)

11. Recursion

(a) Fill in the blank

Every recursive function needs a base case to know when to stop, and needs to call itself to continue recursing

(b) Reverse a Linked List

```
class LinkedList {
    public:
        // Reverse function that client will call
        void reverse();
    private:
        // Private reverse for recursion
        Node* reverse(Node *curr, Node *newNext);
}
void LinkedList::reverse() {
    front = reverse(front, nullptr);
}
LinkedList::Node *LinkedList::reverse(Node *curr, Node *newNext) {
```

**Solution:**

```
// curr will only be nullptr if list is empty
if (curr == nullptr)
        return nullptr;

Node *rest = curr->next;
curr->next = newNext;

if (rest == nullptr)
        return curr;
else
        return reverse(rest, curr);
```

```
}
```

12. Your boss needs an ArrayList that holds **positive** numbers. But they get confused really easily, so they ask you to reduce the number of member variables. That is instead of the usual `size, capacity ints` and a pointer to `data`, your ArrayList should *only* have a pointer to data and a pointer to "something."

```
class ArrayList {
public:
        void insert(int i);
private:
        ...
        int *data;
        int *something;
};
```

(a) Describe how you would design this ArrayList for your boss. Be sure to mention what each pointer points to.

**Solution:** This question is has plenty of solutions that are valid. We outline two below.

1. Something points to an array of capacity 2. This solution amounts to storing the size and capacity inside the array pointed to by something. The data pointer would be used as normal.
2. Something points to the element at position `capacity`. The user only cares about **positive** numbers, so initializing unused element in the array to a *negative* number provides a way to track the size.

(b) Describe how you would insert an element into your ArrayList, and how it would expand.

**Solution:** Elaborating on the previous approaches:

1. This would be identical to the normal approach for ArrayLists, except that size and capacity are stored in the array that `something` points to
2. Insertion can be handled by shifting elements in the array to the right, starting from the provided index and stopping once you overwrite a negative number. If `something` points to a value that is no longer negative after insertion, expand the array as usual. But, the added unused capacity must be initialized to a negative number and `something` must be updated to maintain the invariant established in part a.