Section 05: Midterm Review

1. Memory and B-Tree

(a) Based on your understanding of how computers access and store memory, why might it be faster to access all the elements of an array-based queue than to access all the elements of a linked-list-based queue?

(b) Why might `f2` be faster than `f1`?

```
public void f1(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim();        // omits trailing/leading whitespace
    }
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].toUpperCase();
    }
}

public void f2(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim(); // omits trailing/leading whitespace
        strings[i] = strings[i].toUppercase();
    }
}
```

(c) Let k be the size of a key, p be the size of a pointer, and v be the size of a value.

Write an expression (using these variables as well as M and L) representing the size of an internal node and the size of a leaf node.

(d) Suppose you are trying to implement a B-tree on a computer where the page size (aka the block size) is $B = 130$ bytes.

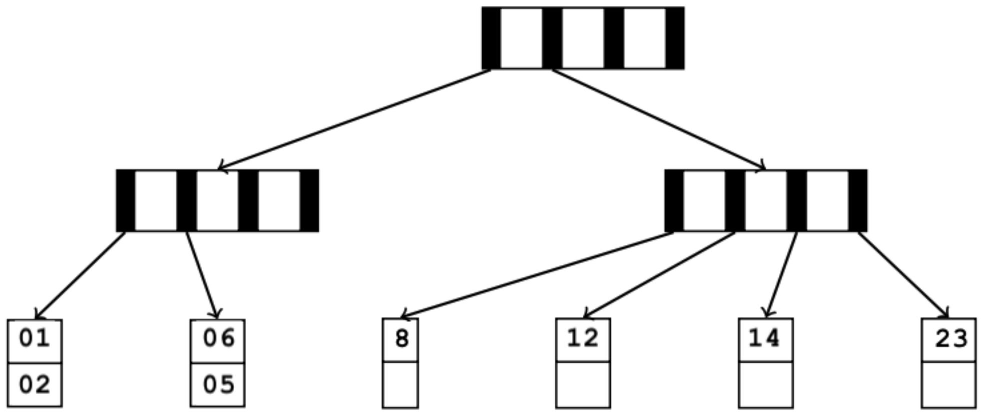
You know the following facts:

- Key size $k = 10$ bytes
- Value size $v = 6$ bytes
- Pointer size $p = 2$ bytes

What values of M and L should you pick to make sure that your internal and external nodes (a) fit within a single page and (b) uses as much of that page as possible.

Be sure to show your work. The equations you derived in the previous part will come in handy here.

(e) Consider the following ”B-Tree”:



(a) What are M and L?

(b) Is there anything wrong with the above B-Tree? If so, what is wrong?

2. Asymptotic Analysis

(a) **Applying definitions**

For each of the following, choose a c and n_0 which show $f(n) \in \mathcal{O}(g(n))$. Explain why your values of c and n_0 work.

(a) $f(n) = 5000n^2 + 6n\sqrt{n}$ and $g(n) = n^3$

(b) $f(n) = n(4 + \log(n))$ and $g(n) = n^2$

(c) $f(n) = 2^n$ and $g(n) = 3^n$

(b) **Runtime Analysis**

For each of the following, give a Θ-bound for runtime of the algorithm/operation:

(a) Worst-case **get** in a B-Tree of size n.

(b) Best-case **get** in a binary search tree of size n.

(c) Best-case **put** in a hash table with size n and a current $\lambda = 1$ if the collision resolution is:

- Separate chaining
- Double Hashing

(d) Pop a value off a stack containing n elements implemented as an array.

(e) Finding the minimum value in a BST of size n.

(f) Finding the minimum value in a AVL tree of size n.

(g) Print out values in AVL tree of size n.

(h) Iterating through and printing every element in an array list using a for loop and the `get(i)` method.

(i) Pop on a stack containing n elements implemented as a singly-linked list.

(j) Inserting a value into an AVL tree of size n, where the value you are inserting is smaller than any other values currently in the tree.

3. Eyeballing Big-Θ bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound. You do not need to justify your answer.

(a)
```
void f1(int n) {
    int i = 1;
    int j;
    while(i < n*n*n*n) {
        j = n;
        while (j > 1) {
            j -= 1;
        }
        i += n;
    }
}
```

(b)
```
int f2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("j = " + j);
        }
        for (int k = 0; k < i; k++) {
            System.out.println("k = " + k);
            for (int m = 0; m < 100000; m++) {
                System.out.println("m = " + m);
            }
        }
    }
}
```

(c)
```
int f4(n) {
    count = 0;
    if (n < 1000) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < i; k++) {
                    count++;
                }
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            count++;
        }
    }
    return count;
}
```

(d)
```
void f5(int n) {
    IList<Integer> arr = new DoubleLinkedList<>();
    for (int i = 0; i < n; i++) {
        if (list.size() > 20) {
            list.remove(0);
        }
        list.add(i);
    }
    for (int i = 0; i < list.size; i++) {
        System.out.println(list.get(i));
    }
}
```

4. Modeling

Consider the following method. Let n be the integer value of the n parameter, and let m be the length of DoubleLinkedList. You may assume that $n > 7$.

```
public int mystery(int n, DoubleLinkedList<Integer> list) {
    if (n < 7) {
        System.out.println(”???”);
        int out = 0;
        for (int i = 0; i < n; i++) {
            out += i;
        }
        return out;
    } else {
        System.out.println(”???”);
        System.out.println(”???”);
        out = 0;
        for (int i : list) {
            out += 1;
            for (int j = 0; j < list.size(); j++) {
                System.out.println(list.get(j));
            }
        }
        return out + 2 * mystery(n - 4, list) + 3 * mystery(n / 2, list);
    }
}
```

Note: your answer to all three questions should be a recurrence, possibly involving a summation. You do not need to find a closed form.

(a) Construct a mathematical function modeling the *approximate* worst-case runtime of this method in terms of n and m.

(b) Construct a mathematical function modeling the *exact* integer output of this method in terms of n and m.

(c) Construct a mathematical function modeling the *exact* number of lines printed out in terms of n and m.

5. AVL/BST

(a) What is the minimum number of nodes in an AVL tree of height 4? Draw an instance of such an AVL tree.

(b) Insert $\{6, 5, 4, 3, 2, 1, 10, 9, 8, 6, 7\}$ into an initially empty AVL tree.

(c) Insert $\{94, 33, 50, 76, 96, 67, 56, 65, 83, 34\}$ into an initially empty AVL tree.

6. Hash tables

(a) What is the difference between primary hashing and secondary clustering in hash tables?

(b) Suppose we implement a hash table using double hashing. Is it possible for this hash table to have clustering?

(c) Suppose you know your hash table needs to store keys where each key's hash code is always a multiple of two. In that case, which resizing strategy should you use?

(d) How would you design a hash table that is designed to store a large amount of data – more then you can fit on RAM?

(e) Consider the following key-value pairs.

(6, a), (29, b), (41, d). (34, e), (10, f), (64, g), (50, h)

Suppose each key has a hash function $h(k) = 2k$. So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

(a) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

(b) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

(c) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

(f) Consider the three hash tables in the previous question. What are the load factors of each hash table?

7. Debugging

Suppose we are trying to implement an algorithm `isTree(Node node)` that detects whether some binary tree is a valid one or not, and returns `true` if it is, and `false` if it isn't. In particular, we want to confirm that the tree does not contain any *cycles* – there are no nodes where the `left` and `right` fields point to their parents.

Assume that the node passed into the method is supposed to be the root node of the tree.

(a) List at least four different test cases for each problem. For each test case, be sure to specify what the input is (drawing the tree, if necessary), and what the expected output is (assuming the algorithm is implemented correctly).

(b) Here is one (buggy) implementation in Java. List every bug you can find with this algorithm.

```
public class Node {
    public int data;
    public Node left;
    public Node right;

    // We are simplifying what the equals method is supposed to look like in Java.
    // This method is technically invalid, but you may assume it's
    // implemented correctly.
    public boolean equals(Node n) {
        if (n == null) {
            return false;
        }
        return this.data == n.data;
    }

    public int hashCode() {
        // Pick a random number to ensure good distribution in hash table
        return Random.randInt();
    }
}

boolean isTree(Node node) {
    ISet<Node> set = new ChainedHashSet<>();
    return isTreeHelper(node, set)
}

boolean isTreeHelper(Node node, ISet<Node> set) {
    ISet<Node> set = new ChainedHashSet<>();
    if (set.containsKey(node)) {
        return false;
    } else {
        set.add(node);
        return isTreeHelper(node.left, set) || isTreeHelper(node.right, set);
    }
}
```