Section 05: Solutions

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?

Solution:

The internal array within the array-based queue is more likely to be contiguous in memory compared to the linked list implementation of an array. This means that when we access each element in the array, the surrounding parts of the array are going to be loaded into cache, speeding up future accesses.

One thing to note is that the array-based queue won't necessarily automatically be faster then the linked-list-based one, depending on how exactly it's implemented.

A standard queue implementation doesn't support the `iterator()` operation, and a standard array-list based queue implements either $\mathcal{O}(n)$ enqueue or dequeue.

In that case, if we're forced to access every element by progressively dequeueing and re-enqueuing each element, iterating over a standard array-based queue would take $\mathcal{O}\left(n^2\right)$ time as opposed to the linked-list-based queue's $\mathcal{O}(n)$ time. In that case, the linked-list version is going to be far faster then the array-list version for even relatively smaller values of n.

The only way we could have the array-based queue be consistently faster is if it supported $\mathcal{O}(1)$ enqueues and dequeues. (Doing this is actually possible, albeit slightly non-trivial.)

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

```java
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();
    }
}
```

Solution:

Temporal Locality. At each iteration, the specific string from the array is already loaded into the cache. When performing the next process `toUppercase()`, the content can just be loaded from cache, instead of disk or RAM.

(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.

Solution:

An internal node has M children, and therefore $M-1$ keys inside. We need a pointer to each child, so we get $Mp+(M-1)k$ bytes.

A leaf node is defined as having L key-value pairs, so the total size is $L(k+v)$ bytes.

Note: This is all assuming the node objects themselves contain no overhead. Java objects *do* have a certain amount of overhead associated with them, but many programming languages let you construct objects (or object-like structures) where their size in bytes is exactly the sum of the size of the fields, with no extra overhead.

For the sake of simplicity, we will be assuming we are using those kinds of programming languages, and that our B-tree nodes have no extra memory overhead.

(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.

Solution:

We want to pick the largest M and L that satisfy $Mp+(M-1)k \le B$ and $L(k+v) \le B$.

We can start by computing L, since that's easier. We have $L(10+6) \le 130$, which simplifies into $L \le \frac{130}{16}$.

If we divide 130 by 16, we get about 8.125. Since L must be a whole number, we know $L = 8$.

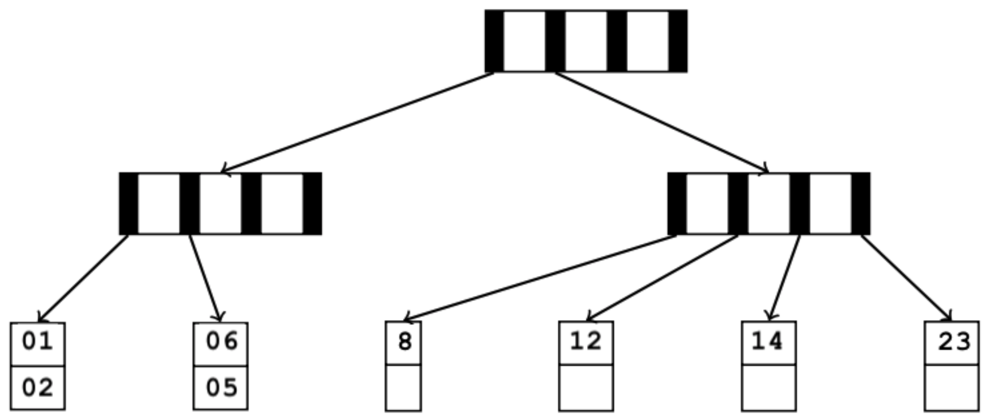
We can do something similar for M. We can rearrange the inequality:

$$\begin{aligned} Mp+(M-1)k &\le B \\ Mp+Mk-k &\le B \\ M(p+k) &\le B+k \\ M &\le \frac{B+k}{p+k} \end{aligned}$$

We plug in numbers, and get $M \le \frac{130+10}{2+10}$. If we divide 140 by 12, we get about 11.66. So, we know $M = 11$.

Our final answer: $M = 11$ and $L = 8$.

(e) Consider the following "B-Tree":



(a) What are M and L?

Solution:

$M = 4$; $L = 2$.

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

Solution:

(a) 6 and 5 should be swapped in the second leaf

(b) The leaves containing 8 and 12 should be consolidated; the leaves containing 14 and 23 should be consolidated.

(c) The inner nodes are missing the values: left-most inner node should have 5 in the first entry, right-most inner node should have 14 in the first entry and the root node should have 8 in the first entry.

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 $g(n) = n^3$

Solution:

Note: The "point" of these types of questions is less about whether or not you can find a working c and n_0, and more about whether or not you can *explain* why your choice of c and n_0 works. To reinforce this point, this answer is deliberately very meticulous – probably a little bit more then is strictly speaking necessary.

We are trying to find a c and n_0 such that $5000n^2 + 6n\sqrt{n} \leq cn^3$ is true for all values of $n \geq n_0$.

First, consider the following inequalities.

$$\begin{aligned}
f(n) &\leq 5000n^2 + 6n\sqrt{n} && \text{for all } n \\
5000n^2 + 6n\sqrt{n} &\leq 5000n^2 + 6n^2 && \text{when } n \geq 1 \\
5000n^2 + 6n^2 &\leq 5006n^2 && \text{for all } n \\
5006n^2 &\leq 5006n^3 && \text{for all } n \geq 1 \\
5006n^2 &\leq cn^3 && \text{when } c \geq 5006 \\
cn^3 &\leq cg(n) && \text{for all } n
\end{aligned}$$

Observe that the above chain of inequalities are true when $c \geq 5006$ and for all values of $n \geq 1$.

This means that when $c \geq 5006$ and $n \geq 1$, it must be the case that $f(n) \leq cg(n)$ is true.

So, we can pick $c = 5006$ and $n_0 = 1$.

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

Note 1: we are presenting this solution in a slightly different way then the previous one to help you get a feel for different ways you can answer this style of question.

Note 2: unlike the previous solution, which walked the reader through the process of discovering a c and n_0, this solution basically provides a c and n_0 up-front, then demonstrates that they satisfy the definition for the given $f(n)$ and $g(n)$. Pragmatically, if you were to answer in this style, you would start by writing down part of the first line, leave it mostly blank, write the rest of the answer, then go back and finish the first line after-the-fact.

Let $c = 5$ and $n_0 = 1$.

Notice that the each of the following equalities and inequalities are true.

$$\begin{aligned}
f(n) &= n(4 + \log(n)) \\
n(4 + \log(n)) &= 4n + n\log(n) \\
4n + n\log(n) &\leq 4n^2 + n^2 && \text{for all } n \geq n_0 \\
5n^2 &= cg(n)
\end{aligned}$$

By the properties of inequalities, we then can conclude that $f(n) \leq cg(n)$ is also true.

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

Solution:

As before, we must find a c and n_0 such that $2^n \leq c3^n$ for all $n \geq n_0$.

Notice that since $2 \geq 3$, multiplying together 2 n times is always going to be smaller the multiplying together 3 n times (so long as n is a positive number.

So, we can pick $c = 1$ and $n_0 = 1$.

(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.

Solution:

$\Theta\left(\log_M(n)\log_2(M)\right)$.

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

Solution:

$\Theta\left(\log(n)\right)$

(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

Solution:

For separate chaining, $\Theta(1)$. For double-hashing, $\Theta(n)$. (If $\lambda = 1$, we must resize first.)

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

Solution:

$\Theta(1)$

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

Solution:

In the worst case, $\Theta(n)$ (the tree could be degenerate, skewing left).

In the average case, where the tree is balanced, $\Theta\left(\log(n)\right)$.

In the best case, $\Theta(1)$ (the tree could be degenerate, skewing right).

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

Solution:

$\Theta\left(\log(n)\right)$

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

Solution:

$\Theta(n)$

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

Solution:

$\Theta(n)$

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

Solution:

The answer depends on how the stack is implemented.

If we pop from the front, $\Theta(1)$. If we pop from the end (which would be a somewhat silly decision), $\Theta(n)$.

(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.

Solution:

$\Theta(\log(n))$

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;
    }
}
```

Solution:

$\Theta(n^4)$

One thing to note that the while loop has increments of $i+ = n$. This causes the outer loop to repeat n^3 times, not n^4 times.

(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);
            }
        }
    }
}
```

Solution:

$\Theta(n^2)$

Notice that the last inner loop repeats a small constant number of times – only 100000 times.

(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;
}
```

Solution:

$\Theta(n)$

Notice that once n is large enough, we always execute the 'else' branch. In asymptotic analysis, we only care about behavior as the input grows large.

(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));
    }
}
```

Solution:

$\Theta(n)$

Note that `arr` would have a constant size of 20 after the first loop. Since this is a `DoubleLinkedList`, `add` and `remove` would both be $\Theta(1)$.

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.

Solution:

$$T(n,m) = \begin{cases} 1 & \text{when } n < 7 \\ m^3 + T(n-4,m) + T(n/2,m) & \text{otherwise} \end{cases}$$

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

Solution:

$$I(n,m) = \begin{cases} \sum_{i=0}^{n-1} i & \text{when } n < 7 \\ m + 2I(n-4,m) + I(n/2,m) & \text{otherwise} \end{cases}$$

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

Solution:

$$P(n,m) = \begin{cases} 1 & \text{when } n < 7 \\ 2 + m^2 + P(n-4,m) + P(n/2,m) & \text{otherwise} \end{cases}$$

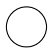
5. AVL/BST

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

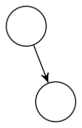
Solution:

We can approach this problem iteratively (or recursively, depending on your point of view).

First, what is the smallest possible AVL tree of height 0? Well, just a single node:



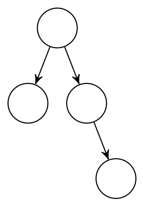
What about the smallest possible AVL tree of height 1? With some thought, we arrive at this answer:



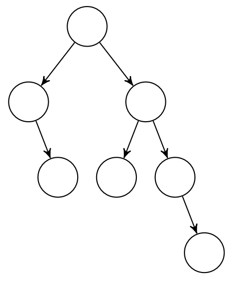
What about the smallest possible AVL tree of height 2? Well, if we think about this critically, we know that our tree must contain at least one subtree of height 1 (if it's any other height, our AVL tree wouldn't have height 2). If we're trying to minimize the number of nodes, we might as well make the other subtree have a height of 0. That minimizes the total number of nodes.

Now, if only we know what the smallest possible AVL trees of height 1 and 0 were...

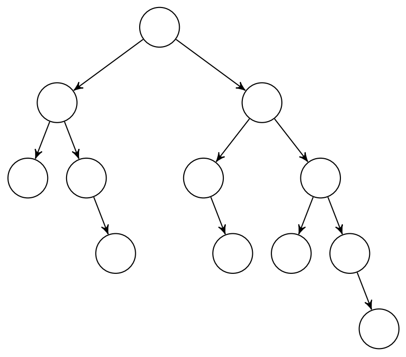
We can reuse our answers above to get:



We repeat to get the smallest possible AVL tree of height 3: combine together the smallest possible AVL trees of heights 1 and 2:



We repeat one more time, for the smallest possible AVL tree of height 4:



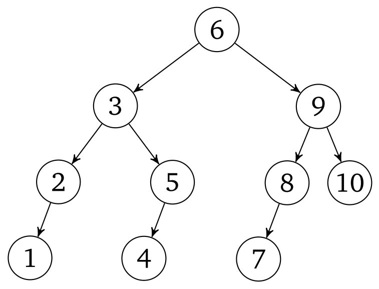
If we count the number of nodes, we get 12.

More generally, we can find the number nodes in the smallest possible AVL tree by computing the following recurrence:

$$\text{minNumNodes}(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ 1 + \text{minNumNode}(n-2) + \text{minNumNodes}(n-1) & \text{otherwise} \end{cases}$$

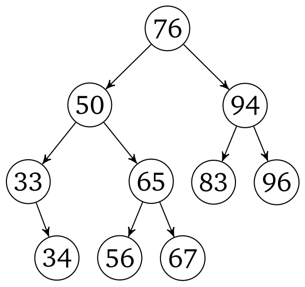
(b) Insert $\{6, 5, 4, 3, 2, 1, 10, 9, 8, 6, 7\}$ into an initially empty AVL tree.

Solution:



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

Solution:



6. Hash tables

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

Solution:

Primary clustering occurs after a hash collision causes two of the records in the hash table to hash to the same position, and causes one of the records to be moved to the next location in its probe sequence. Linear probing leads to this type of clustering.

Secondary clustering happens when two records would have the same collision chain if their initial position is the same. Quadratic probing leads to this type of clustering.

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

Solution:

Yes, though the clustering is statistically less likely to be as severe as primary or secondary 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?

Solution:

Any strategy where the size of the table is not a multiple of two. For example, we could either make the hash table's array have an odd initial starting size then double to resize, or we could have use the prime doubling strategy.

If the table size *were* a multiple of two, we would end up using only half the available table entries (if we were using separate chaining) or have more collisions then usual (if we were using open addressing).

(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?

Solution:

One idea: use separate chaining, have each bucket take up one or more pages. Rather then resizing when $\lambda \approx 1$, resize once the each bucket's size is starting to approach some multiple of a page size. Store the initial array in RAM.

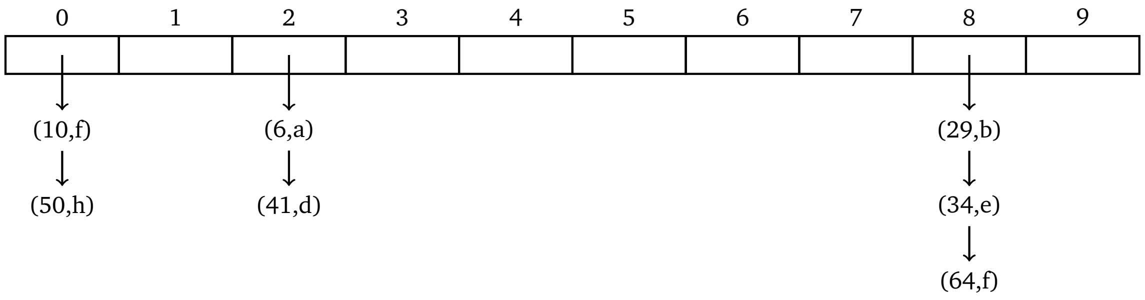
(e) Consider the following key-value pairs.

(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.

Solution:



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

Solution:

0	1	2	3	4	5	6	7	8	9
(10, f)	(64, g)	(6, a)	(41, d)	(50, h)				(29, b)	(34, e)

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

Solution:

0	1	2	3	4	5	6	7	8	9
(10,f)	(50, h)	(6,a)	(41,d)				(64, g)	(29,b)	(34,e)

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

Solution:

$$\lambda = \frac{7}{10} = 0.7$$

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).

Solution:

Some examples of inputs. On an exam, be sure to provide a wide variety of test cases. Some of your cases should test the 'happy' case, some cases should test weird inputs, some cases should test invalid/incorrect input...

(a) Input: null
Output: true

(b) Input: a regular, standard tree
Output: true

(c) Input: a degenerate tree that looks like a linked list
Output: true

(d) Input: a tree containing a node where either the `left` or `right` pointers is pointing to itself
Output: false

(e) Input: a tree containing a node where either the `left` or `right` pointers is pointing to some parent node
Output: false

(f) Input: a tree containing a node where the `left` or `right` pointers is pointing to a neighbor
Output: false

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

```java
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);
    }
}
```

Solution:

Some bugs:

(a) The method does not attempt to handle the case where the input node is null

(b) If the hashCode returns a random int every time it's called, it's going to play havoc with the set – we wouldn't be able to reliably insert the node anywhere.

(c) We are trying to use the `ChainedHashSet` as a way of keeping track of every node object we've previously visited. However, we override the set with a new, empty one on each recursive call, wiping out our progress.

(d) The "or" in the last line should be an "and" – if the left subtree or the right subtree isn't actually a tree, we should return 'false' right away. Currently, we return false only if *both* subtrees aren't actually trees.