Question 1

Part (a)

Required: Compute a tight bound (using asymptotic notation) on the sum of the number of times line 8 and line 12 are executed in the best case.

Want to Show: Best case is $\Theta(1)$.

Since line 8 must always be executed once, we know that the best case is at least $\Omega(1)$.

Now, consider a family of inputs, one for each $n \in \mathbb{N}$ as follows. Consider $k_i = i$ for $i \in \{1, ..., n-1\}$ and $k_n = 1$ and $l = n$. Hence, after $insert(D, k_1), ..., insert(D, k_n)$, we have that $D.size = n - 1$ and D does not contain the key n. Consider $delete(D, l)$ when $l = n$. Hence, after the while loop on line 3 ends, we get that $i = D.size$. Hence, the statement on line 8 is true, and we return on line 9. Hence, we get a best case of $\mathcal{O}(1)$ as line 8 is executed once and line 12 is never executed.

Since we have $\Omega(1)$ and $\mathcal{O}(1)$, we get that the best case is $\Theta(1)$.

Part (b)

Required: Compute a tight bound (using asymptotic notation) on the sum of the number of times line 8 and line 12 are executed in the worst case.

Want to Show: Worst case is $\Theta(n)$.

We know $D.size \leq n$ since we insert at most n distinct keys. Line 8 is always executed once. And the for loop on line 11 has at most $n - 1$ iterations when $i = 0$ and $D.size = n$. Hence, line 12 is executed at most $n - 1$ times. Hence, we get at most $1 + (n - 1) = n$ executions of lines 8 and 12. Hence, the worst case is at most $\mathcal{O}(n)$.

Now, consider a family of inputs, one for each $n \in \mathbb{N}$ as follows. Consider $k_i = i$ for $i \in \{1, ..., n\}$, and $l = 1$. Hence, after $insert(D, k_1), ..., insert(D, k_n)$, we have that $D.size = n$. Consider $delete(D, l)$ when $l = 1$. Since the first element of the array is 1, the while loop on line 3 breaks after the first iteration and we have that $i = 0$ by the time we execute line 8. Hence, the for loop on line 11 has $n - 1$ iterations since $D.size = n$ and $i = 0$. Hence, line 12 is executed $n - 1$ times. Hence, we get a total of $1 + (n - 1) = n$ executions of lines 8 and 12. Hence, the worst case is at least $\Omega(n)$.

Since we have $\Omega(n)$ and $\mathcal{O}(n)$, we get that the worst case is $\Theta(n)$.

Part (c)

Required: Compute the expected sum of the number of times line 8 and line 12 are executed. Show all of your work, and give an exact number as your solution (i.e. no asymptotic notation).

Calculating expected size of D

We know that $D.size$ can be such that $1 \leq D.size \leq n$ after $insert(D, k_1), ..., insert(D, k_n)$ depending on the sampling. Hence, we will first calculate the expectation of $D.size$.

Let z represent the number of unique keys of D. Since D is a dictionary (with no duplicate keys), z just represents the number of keys in D.

For $i \in \{1, ..., n\}$, let y_i be an indicator variable such that $y_i = 1$ if key i is inserted into D, and $y_i = 0$ if key i is not inserted into D.

We have that $z = \sum_{i=1}^{n} y_i$. Hence, for each $i \in \{1, ..., n\}$ we have that,

$$\begin{aligned} E[y_i] &= 1 \cdot Pr(y_i = 1) + 0 \cdot Pr(y_i = 0) \\ &= Pr(y_i = 1) \\ &= 1 - Pr(y_i = 0) \\ &= 1 - \Big(1 - \frac{1}{n}\Big)^n && \text{By Bernoulli distribution} \\ &= 1 - \Big(\frac{n-1}{n}\Big)^n \end{aligned}$$

Hence,

$$\begin{aligned} E[z] &= E\Big[\sum_{i=1}^{n} y_i\Big] \\ &= \sum_{i=1}^{n} E[y_i] && \text{By linearity of expectation} \\ &= \sum_{i=1}^{n} \Bigg(1 - \Big(\frac{n-1}{n}\Big)^n\Bigg) \\ &= n\Bigg(1 - \Big(\frac{n-1}{n}\Big)^n\Bigg) \end{aligned}$$

Hence, we have that the expectation for $D.size$ is $n\Bigg(1 - \Big(\frac{n-1}{n}\Big)^n\Bigg)$

Hereafter, let $m = n\Bigg(1 - \Big(\frac{n-1}{n}\Big)^n\Bigg)$ as a shorthand.

Expected sum of executions of lines 8 and 12

Let $m = n\left(1 - \left(\frac{n-1}{n}\right)^n\right)$ be the expectation for $D.size$. We will use m below.

Defining our sample space:

Let x_0 represent "all inputs where $D.size = m$ and l appears at index 0".

Let x_1 represent "all inputs where $D.size = m$ and l appears at index 1".

In general, for $i \in \{0, ..., m-1\}$, let x_i represent "all inputs where $D.size = m$ and l appears at index i".

Finally, let x_m represent "all inputs where $D.size = m$ and l does not appear in D".

Let $S_m = \{x_0, x_1, ..., x_{m-1}, x_m\}$ be our sample space.

Calculating our probabilities:

The probability that any single array-slot in D has key l is $p = \frac{1}{n}$. Note, we will use p for readability, and substitute the actual value for p towards the end.

For $i \in \{0, ..., m-1\}$ we get that $Pr(x_i) = (1-p)^i p$ by the Bernoulli distribution.

And $Pr(x_m) = (1-p)^m$ when l does not appear in D.

Calculating exact steps:

Let $t(x)$ be the number of executions of lines 8 and 12 on $x \in S_m$.

$t(x_0) = 1 + (m-1) = m$ as line 8 is executed once and line 12 is executed $m-1$ times.

$t(x_1) = 1 + (m-2) = m-1$ as line 8 is executed once and line 12 is executed $m-2$ times.

In general, for $i \in \{0, ..., m-1\}$ we have that $t(x_i) = m - i$.

Finally $t(x_m) = 1$ since line 8 is executed once and we return on line 9.

Now let T be a random variable. We will calculate the expectation for the number of times lines 8 and 12 are executed. Please see the next page.

$$\begin{aligned}
E[T] &= \sum_{i=0}^{m} t(x_i)Pr(x_i) \\
&= t(x_m)Pr(x_m) + \sum_{i=0}^{m-1} t(x_i)Pr(x_i) \\
&= 1 \cdot (1-p)^m + \sum_{i=0}^{m-1} (m-i)(1-p)^i p \\
&= (1-p)^m + \sum_{i=0}^{m-1} (m-i)(1-p)^i p \qquad \text{where } p = \frac{1}{n} \text{ and } m = n\left(1 - \left(\frac{n-1}{n}\right)^n\right)
\end{aligned}$$

Note, the final expression above is **all in terms of** n since $p = \frac{1}{n}$ and $m = n\left(1 - \left(\frac{n-1}{n}\right)^n\right)$. However, we will keep p and m above for readability for the grader.

The professor on piazza indicated that we can leave our exact answer as a summation and that we do not require a closed form.

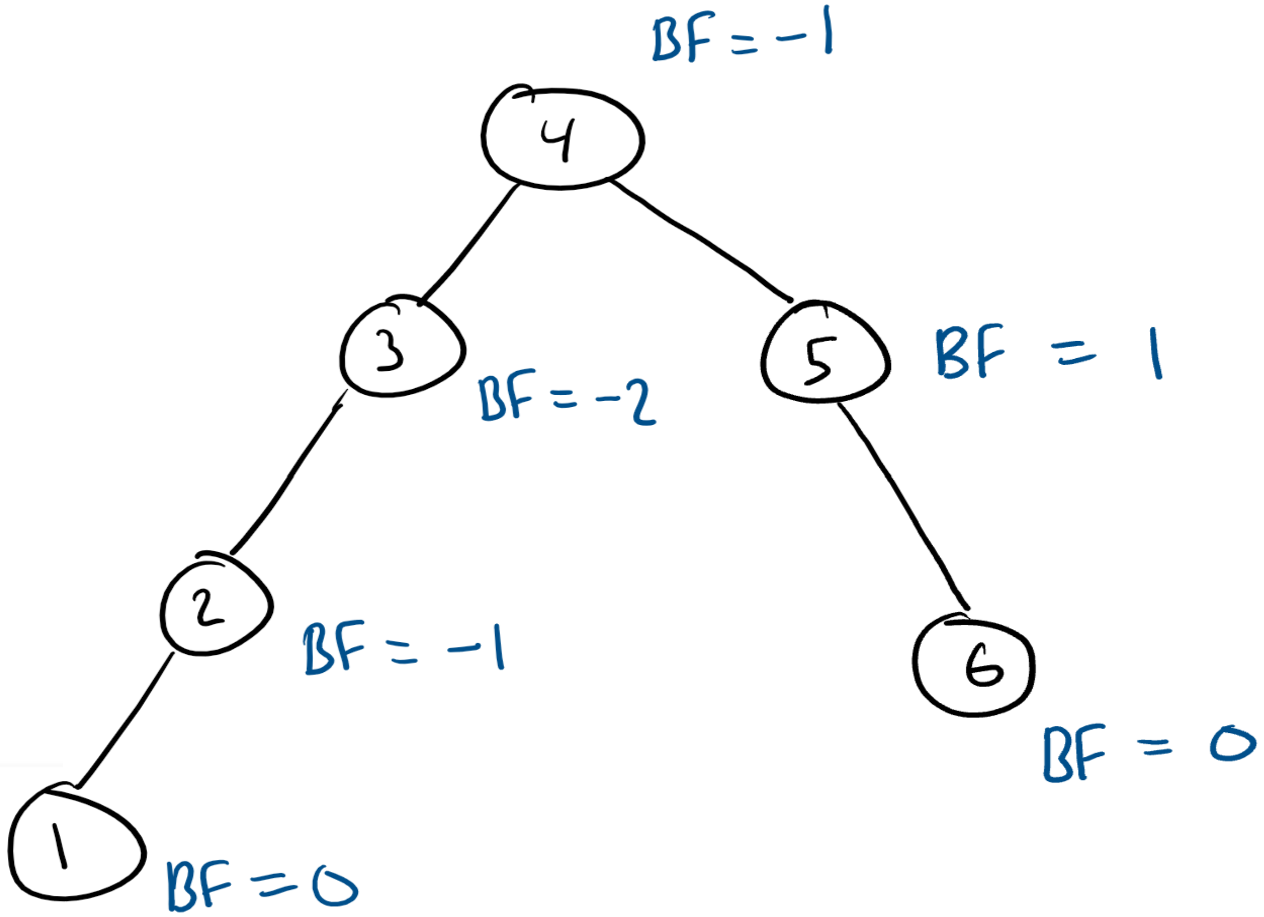
Hence, the last expression above is our final exact answer for the expectation of the number of executions of lines 8 and 12, as required.

Question 2

Part a)

Note: On Piazza, the professor indicated that we may insert hand-drawn trees into LaTeX.

Consider the following tree that is depth balanced, but not AVL balanced. Note, there is a node with balance factor of -2.



Part b)

We will show that $j = 5$ satisfies the following two requirements.

i. for all depth-balanced BSTs T with j or fewer nodes, T is also AVL balanced, and

ii. there exists a depth-balanced BST T' with $j + 1$ nodes that is not AVL balanced.

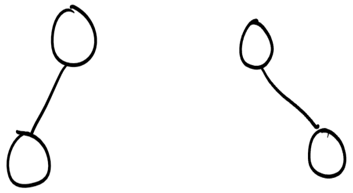
Showing Condition i)

We will show that for each k such that $k \leq j = 5$, we have that for all depth-balanced BSTs T with k nodes, T is also AVL balanced.

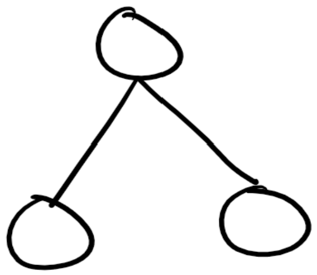
Case $k = 0$: There is exactly one BST with 0 nodes which is trivially depth-balanced and AVL balanced.

Case $k = 1$: There is exactly one BST with 1 node which is trivially depth-balanced and AVL balanced.

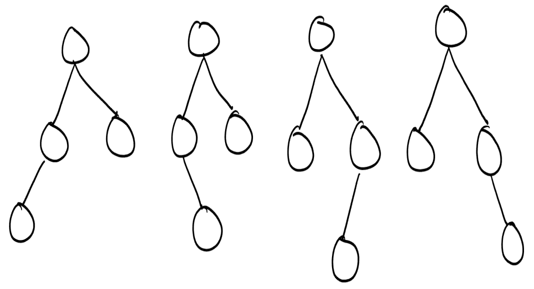
Case $k = 2$: There are exactly 2 BSTs with 2 nodes which are both depth-balanced and AVL balanced.



Case $k = 3$: A BST with 3 nodes has either height 1 or height 2. However, a BST with 3 nodes and height 2 is not depth-balanced because $d(root) + 1 = 0 + 1 = 1$, but we would get that $|h(root.right) - h(root.left)| = 2 > 1$. Hence, there is only 1 BST with 3 nodes and height 1 that is depth balanced. It is drawn below. And clearly it is also AVL balanced.

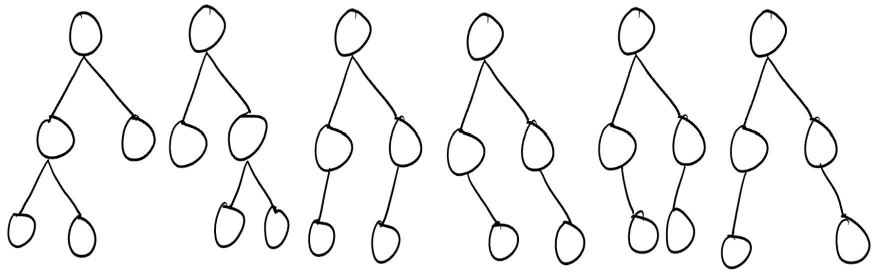


Case $k = 4$: A BST with 4 nodes has either height 2 or 3. However, a BST with 4 nodes and height 3 is not depth-balanced because $d(root) + 1 = 0 + 1 = 1$, but we would get that $|h(root.right) - h(root.left)| = 3 > 1$. Hence, we must consider BSTs with 4 nodes that are of height 2. There are exactly 4 such BSTs and they are drawn below. Each of them is depth-balanced and AVL balanced.



Case $k = 5$: A BST with 5 nodes has either height 2,3, or 4. However, a BST with 5 nodes and height ≥ 3 is not depth balanced because $d(root) + 1 = 0 + 1 = 1$, but we would get that $|h(root.right) - h(root.left)| \geq 2 > 1$. Hence, we must consider BSTs with 5 nodes that are of height 2. There are exactly 6 such BSTs and they are drawn below.

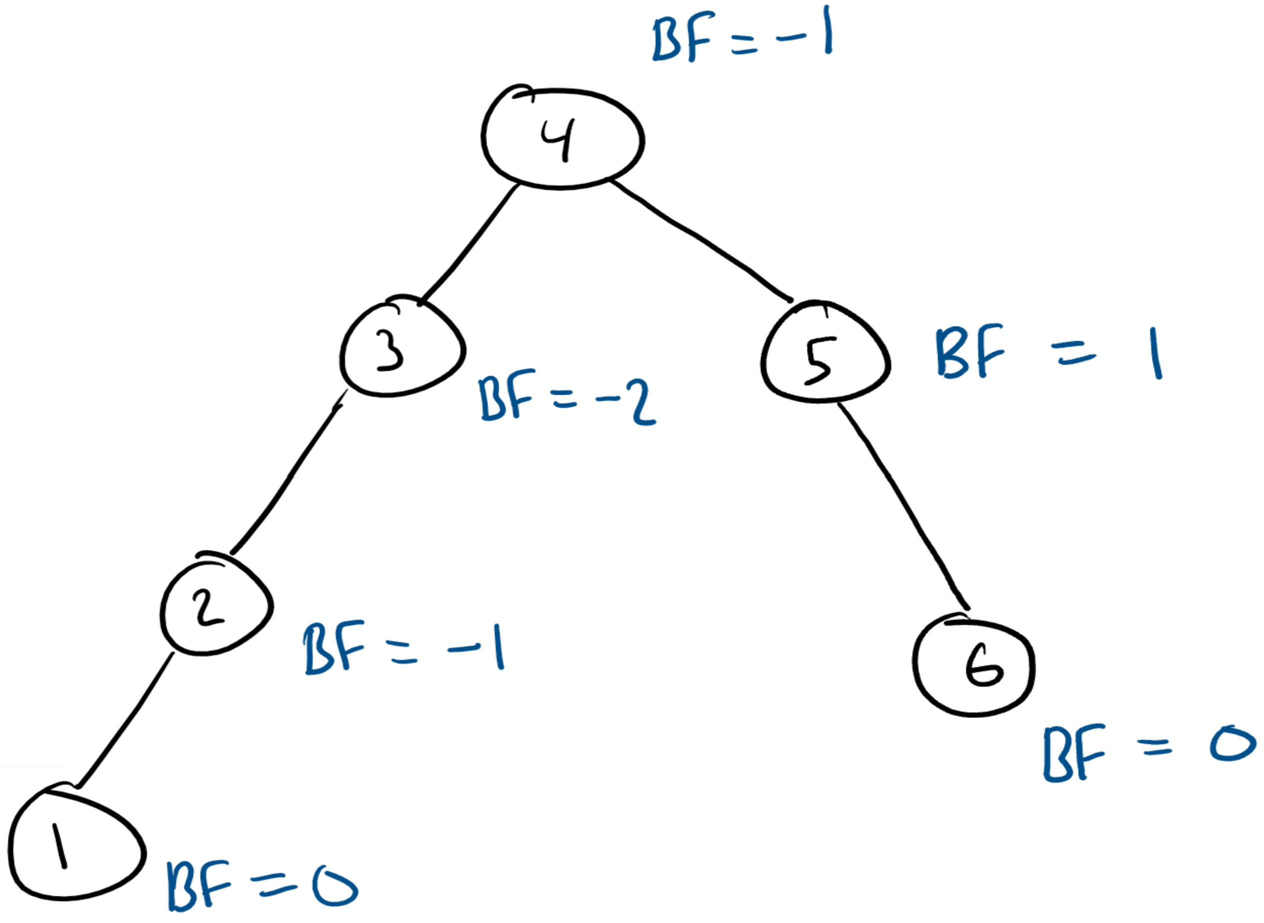
All 6 BSTs with 5 nodes that are depth-balanced are also AVL balanced.



Hence, $j = 5$ satisfies condition i.

Showing Condition ii)

We want to show that there exists a depth balanced BST T' with $j + 1 = 6$ nodes that is not AVL balanced. We can just use the example we used in part a). We will copy it below. Note, there is a node with balance factor of -2.



Hence, we have proven that $j = 5$ is indeed our required number.

Question 3

Part (a)

We will use an AVL-tree implementation of a dictionary plus a few additional agumentations. So we have `Insert(D, (k, v))`, `Search(D, k)` and `Delete(D, k)` operations which we will use.

We will further augment the AVL tree to have a size attribute for each node which stores the size of the subtree rooted at that node. This will allow us to support `Rank(D, k)` and `Select(D, r)` operations as defined for ordered sets.

We will also augment the AVL tree to have a `num_nodes` attribute which stores the total number of nodes. Any Insert or Delete operations will update `num_nodes` accordingly.

Finally we will augment the AVL tree so that each node has `previous_rank` and `next_rank` attributes that are pointers to the previous node and next node in terms of ranking. We could have that `previous_rank` or `next_rank` are `Null` when we are at the first node or last node respectively in terms of rank. And we would have constant time access to these node's attributes. In effect, we are adding properties of a doubly-linked list to our data structure. Note that `Delete`, and `Insert` implementations must be modified accordingly to update these pointers in their algorithms. Since we only add pancakes to the end of the sequence, we do not need to modify `Insert` to deal with arbitrary inserts and hence arbitrary changes to `previous_rank` and `next_rank`.

Each pancake will have a key (represented by an integer) in the tree. The value corresponding to each key will be the pancake object itself with its attributes (flavour and mass). For each pancake that is added to the sequence, its key will be strictly larger than every pancake currently in the list. We will provide more details in part (f) when we describe the operation `cook`.

The order of the sequence of our pancakes will be given by the ranking of the keys.

We will also have a pointer `current` to indicate the current pancake. And `current` is `Null` if and only if the sequence of pancakes (the tree) is empty.

We will use the algorithms for `Insert(D, (k, v)), Search(D, k), Delete(D, k), Rank(D, k)` and `Select(D, r)` (with the aforementioned modifications) along with the attributes `previous_rank` and `next_rank` for each node and the attribute `num_nodes` to implement the required operations in parts (b) through (g).

Please see parts (b) through (g) for those implementations.

Part (b)

The operation `Forward()` will be implemented as follows:

Look at the node that `current` is pointing to. If `current` is `Null`, do not do anything as the sequence of pancakes is empty. Otherwise, look at the `next_rank` attribute of this node. If `next_rank` is `Null`, do not do anything as we are at the last node (pancake). If `next_rank` is not `Null`, update the `current` pointer to the node that `next_rank` points to.

Since we are just updating a pointer, this clearly takes constant time. Hence, we get a worst-case runtime of $\mathcal{O}(1)$.

Part (c)

The operation `Backward()` will be implemented as follows:

Look at the node that `current` is pointing to. If `current` is `Null`, do not do anything as the sequence of pancakes is empty. Otherwise, look at the `previous_rank` attribute of this node. If `previous_rank` is `Null`, do not do anything as we are at the first node (pancake). If `previous_rank` is not `Null`, update the `current` pointer to the node that `previous_rank` points to.

Since we are just updating a pointer, this clearly takes constant time. Hence, we get a worst-case runtime of $\mathcal{O}(1)$.

Part (d)

The operation `Move(j)` will be implemented as follows:

Assume that we are working in 1-based indexing (and not 0). The case of 0-based indexing is nearly identical. And assume the j-th pancake exists.

Execute `Select(D,j)` to find the key in our tree with rank j. This key represents the j-th pancake. Update the `current` pointer to this key.

Since `Select` takes $\mathcal{O}(\log n)$ time in the worst case from lecture, we have that `Move(j)` has a worst-case runtime of $\mathcal{O}(\log n)$.

Part (e)

The operation `Serve()` will be implemented as follows:

If `current` is `Null`, then do not do anything as the sequence of pancakes is empty.

Otherwise, consider the key that `current` is pointing towards. Call this key k.

Execute `Delete(D,k)`. And update the pointer `current` to the `previous_rank` of the node with key k. If `previous_rank` is `Null` and k is first in the sequence, then update the pointer `current` to k's `next_rank` (which may also be `Null` if our sequence had 1 pancake).

If our sequence had only 1 pancake, `current` will point to `Null` after deletion of k.

Note that `Delete(D,k)` is modified to update the `previous_rank` and `next_rank` attributes upon deletion of nodes as discussed in part a).

Since `Delete(D,k)` has a worst-case runtime of $\mathcal{O}(\log n)$, and moving the `current` pointer takes constant time, we have that `Serve()` has a worst-case runtime of $\mathcal{O}(\log n)$.

Part (f)

The operation `Cook(p)` will be implemented as follows:

First consider `num_nodes` which is the total number of nodes/keys in our tree.

Execute `Select(D, num_nodes)` to find the key with rank `num_nodes`. This is the last key in our sequence representing the last pancake.

Let k be the key returned by `Select(D, num_nodes)`. Assume that all our keys are natural numbers so that $k \in \mathbb{N}$. Consider $k + 1$.

Let p be presented by the key $k + 1$. i.e. key $k + 1$ has value p.

Execute `Insert(D, (k+1, p))`.

Since `Select` and `Insert` each has a worst-case runtime of $\mathcal{O}(\log n)$, we have that the worst case runtime of `Cook(p)` is $\mathcal{O}(\log n)$.

Part (g)

The operation `Examine()` will be implemented as follows:

If `current` is `Null`, then do not return anything as the sequence of pancakes is empty.

Otherwise, look at the node that `current` is pointing to. Use the key at this node to access its corresponding pancake object (the key's value). Finally return this pancake's attributes.

Since we know that `current` is just a pointer, `Examine()` takes constant time.

Hence, `Examine()` has a worst-case runtime of $\mathcal{O}(1)$.