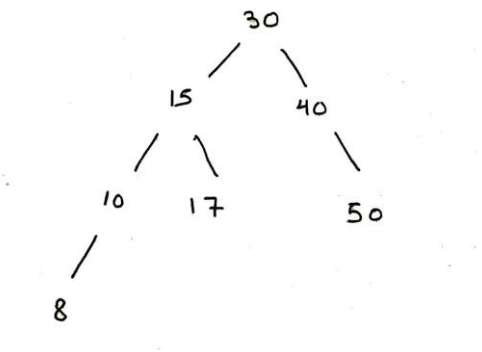
**Aleksandr Itskovich (ai1221)**

**Assignment 2**

**Question 1: Binary Search Trees**

a)

- Insert: 30, 15, 40, 10, 17, 50, 8




- Explain why this tree is an AVL tree.

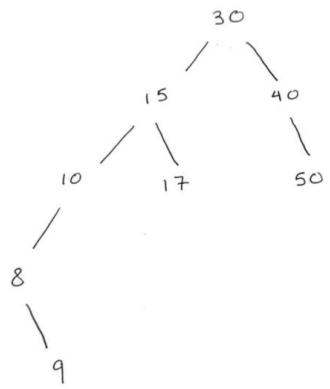
An AVL tree is characterized by the following invariant: that the difference in heights between the subtrees rooted at any node is at most 1.

We see that this invariant is met in the tree drawn above. Take, for example, the root node at 30. The height of its left subtree, LH, (rooted at 15) is 2, since the longest path from 15 to a leaf has length 2. The height of the right subtree, meanwhile is 1. Repeating the same analysis for each node yields:

| Node | Left Subtree Height | Right Subtree Height | Difference |
|---|---|---|---|
| 30 | 2 | 1 | 1 |
| 15 | 1 | 0 | 1 |
| 40 | -1 | 0 | -1 |
| 10 | 0 | -1 | 1 |
| 17 | -1 | -1 | 0 |
| 50 | -1 | -1 | 0 |
| 8 | -1 | -1 | 0 |

We can see that the AVL height invariant is maintained at every node in the binary tree, therefore the tree is an AVL tree.

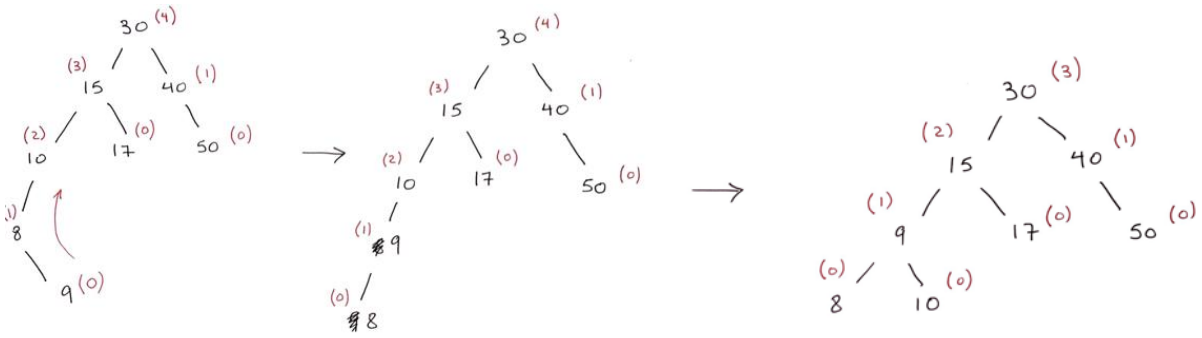
- Next, insert the key 9. Explain why the result of the insert no longer satisfies the AVL property.




We appended an additional node to the left subtree of 30. Now we see that for node 30, the height of its left subtree rooted at 15 is 3, while the height of the right subtree rooted at 40 remains unchanged at 1. Because $|3 - 1| = 2$ the difference in height between the two subtrees is greater than 1. Therefore the tree is no longer an AVL tree.

- Show how you could perform at most 2 rotations in order to restore the AVL tree property.

We could rotate twice to restore the tree heights back to having a difference of at most 1.




In general, when an AVL tree is out of balance there are 4 types of rotations that are similar to the ones discussed in the RB Tree case: two bent rotations (handling left-right and right-left cases) and two straight rotations (handling left-left and right-right cases). After performing either straight rotation, the tree is balanced. Moreover, we can always convert a bent case into a straight case by performing one rotation. That means we are never more than 2 rotations away from having a balanced AVL tree.

We either have

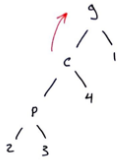
1. Bent Case rotation, followed by a straight case rotation (2 rotations)
2. Or just a straight case (1 rotation)

These cases are summarized below:

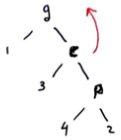
Left Right


Right Left


Left Left

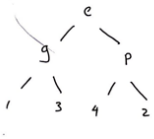



Right Right




Balanced


Balanced



**b)** Suppose T1 and T2 and T3 each reference the root nodes of a binary search tree on n nodes. We wish to combine these three trees into one binary search tree. Explain how this can be done in $O(n)$ time, in such a way that the resulting BST has height $O(log\ n)$. You do not need to write pseudo-code for your procedure, but you must properly describe your steps.

We are given that we have 3 nodes that point at the roots of different binary search trees. Let's assume that the sizes of the trees are $n_1$, $n_2$ and $n_3$, respectively.

We can carry out the following procedure to build a minimal binary search tree in linear time.

1. Perform an in-order traversal of each tree, storing the values in an array as they are encountered. This produces 3 separate, *sorted* arrays, with sizes $n_1$, $n_2$ and $n_3$, respectively.
    a. We know from class that in-order traversal is linear for each tree.
    b. That means the total work done for all 3 trees is $O(n_1 + n_2 + n_3) = O(n)$
2. Merge the 3 sorted arrays produced in step one into a single array. The logic behind this step is identical to the way arrays are merged in merge sort, only now we have a 3-way merge instead of a 2 way merge. This produces a single, *sorted* array.
    a. Overall, we have to combine $n_1 + n_2 + n_3$ elements, so the total time is again $O(n)$.
3. Now that we have a single, sorted array, we know from class that we can produce a minimal BST from a sorted array by *recursively selecting the median element* from the left and right subarrays, and inserting that median element into the tree.
    a. Getting the median element recursively is given by $T(n) = 2T(n/2) + c$, which we know to have a solution of $O(n)$ by the Master Method.
    b. Inserting into the tree takes constant time per element because we don't have to traverse the tree searching for the position of the newly inserted element; we know exactly where to insert elements since we're selecting medians from subsequent sub-arrays, so in this case insertion is constant is constant.
4. Overall, we have 3 steps, each of which is shown to take $O(n)$ time. That means the overall procedure also runs in $O(n)$ time.

**c)**

```
LazyDelete(T, z):
        z.deleted = true
        return T
```

- **TreeSearch**

We would have to update the TreeSearch method only slightly: if we find a node that has a key equal to the target search key, we also need to check if the node was deleted before returning the match; in other words, we just want to make sure we are only matching on non-deleted nodes. Otherwise the algorithm is the same. This change is shown below:

```
TreeSearch(T, z):
        x = T
        while x != NIL and z.key != x.key
                if z.key < x.key
                        x = x.left
                else
                        x = x.right
        if (x != nil AND x.deleted = false)
                return x
        else
                return nil
```

- TreeInsert

```
TreeInsert(T, z):
        if T = NIL
                return z
        else
                x = T
                while x != NIL and x.key != z.key
                        y = x
                        if z.key < x.key
                                x = x.left
                        else
                                x = x.right
                if x != NIL and z.key = x.key
                        x.deleted = false
                else
                        z.parent = y
                        if z.key < y.key
                                y.left = z
                        else
                                y.right = z
                return T
```

- Disadvantages:

*Complexity:*

Implementing lazy deletion makes the implementation of other methods more complicated. E.g. tree search.

*Space Inefficiency:*

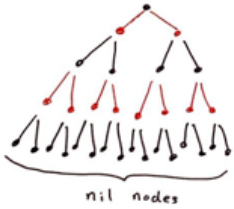
More importantly, it increases the overall space-complexity of the tree. Since deleted nodes are never removed from the tree, they continue to take up space. For example, if we delete half the nodes in a tree containing n nodes, we are effectively wasting half of the nodes. It also means that space might only increase, never shrink, since nodes are never truly removed. All of this also means that the time required to carry out an operation such as search or insert also increases by some constant factor as the number of deleted nodes increases.

**Question 2: Red-Black Trees**

**a)**

- For height b, what is the maximum number of nodes in the tree (excluding NIL nodes)? Describe the shape of the tree and its coloring.

To achieve the maximum number of nodes in a RB tree, we will want to have a full tree (that is, all possible nodes occupied) with the maximum height. We know from class that to achieve maximum height in a RB tree, we will want nodes levels to be alternating between red and black. That way we have half black nodes and half red nodes:




For a given black height, b, we know that for every black level, we are going to have an additional red level (because we want the tree to be as full as possible for a given black height, with alternation). That means that the number of levels is going to be given by 2b.

The number of nodes in each level is given by the level number raised to the 2. That means, the number of nodes in the first level (level is) is $2^0 = 1$, the number of nodes in the second level is given by $2^1 = 2$, etc.

We need to total sum of all internal nodes. Adding up all the nodes in all the levels gives us:

$$total\ number\ of\ internal\ nodes = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots + 2^{2b} = 2^{2b+1} - 1$$

However, because we don't want to count the last level of nil nodes (and only the internal nodes) we want $2^{2b+1}$ to be just $2^{2b}$.

Therefore the total number of internal nodes in the **maximum** case is given by $\mathbf{2^{2b} - 1}$.

- Repeat for minimum number of nodes:

To get the minimum number of nodes, we want to get the smallest possible BT tree. This corresponds to the case when all leaves are black.

The total number of levels of this tree will be just b. In this base, we have

$$total\ number\ of\ internal\ nodes = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \cdots + 2^{b} = 2^{b+1} - 1$$

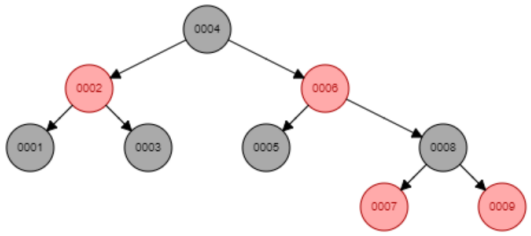
As above, we don't want to count the nil nodes so $2^{b+1}$ becomes just $2^b$.

Therefore the total number of internal nodes in the **minimum** case is given by $\mathbf{2^{b} - 1}$.

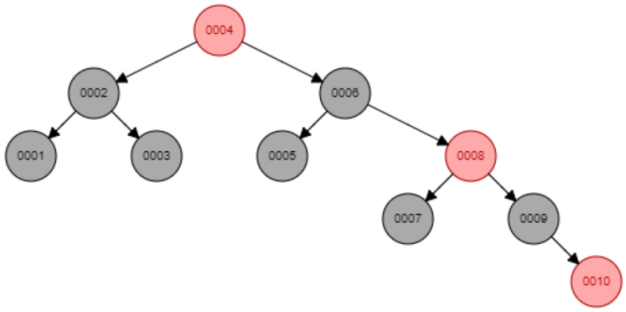
**b)**

In a Red-Black tree, black height is only increased when we insert a red node that results in a recoloring *at the root of the tree*. In other words, black height is increased whenever we have a situation when the root has *two red children* and an insertion requires them to be recolored.

An example of this is shown below:




If we insert node 10, the result is:



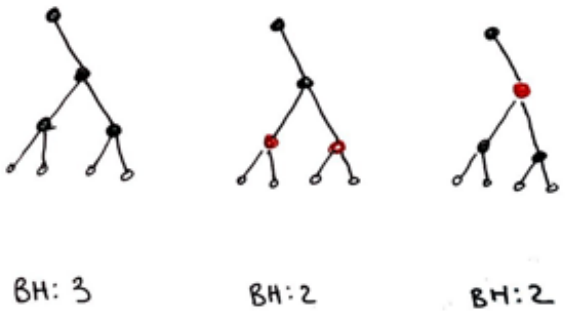

Notice, in the last picture the root node must still be re-colored black.

The reason that we only add height when the node has two red children that need to be recolored is as follows: during a typical recoloring, we swap the colors of the red children with the black parent. Whenever this happens, we increase the black height of the *grandparent node* by 1. This in and of itself does not increase the overall black height of the tree. However, when we have a special situation in which the grandparent is the root node, we increase the black height of the root node by 1, and therefore of the entire tree by 1.

It is apparent, then, that is it **not possible** to insert 2 nodes into a red-black tree (one after the other) in such a way that the black-height is increased by 2. As we see in the example above, once we have increased the black-height after a single insertion, the root and both of its children are black. In order to increase the black height, one of the prerequisite conditions is that the root has two *red* children so that we might recolor at the root. It is impossible to have this be the case immediately after increasing the black height. We would require several rotations before that can happen again. Therefore it is impossible for two successive node additions to increase the black-height twice.

**c)** It is impossible to transform this tree into a proper red-black tree.

Consider the right subtree first. The right height is 2. We can either have all nodes be black, or alternate red and black nodes. Therefore the right subtree has $2 \leq bh \leq 3$.




Now, we can look at the left subtree.

In the left subtree we have multiple paths down to NIL nodes. These paths have lengths 3, 4, and 5. The shortest path has length 3 and the longest path has length 5.

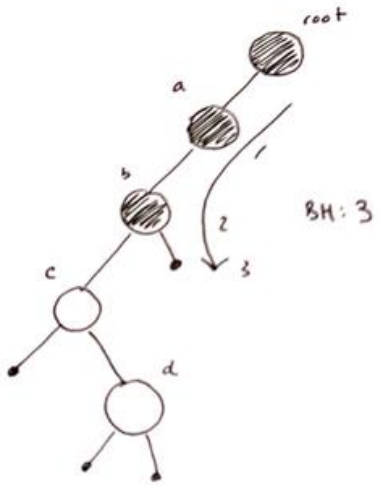
Let's examine the left subtree with respect to the right subtree.

- **Case 1:** the right subtree has Black-Height 2:

If the right subtree has a BH of 2, then it is impossible for the left subtree to maintain a BH of 2 along its longest path of length 5. Because we cannot have adjacent red nodes, they must be alternated with black nodes. The maximum number of red nodes we could have along that height would be 3, which means we would need to have *at minimum* 2 black nodes, which means leaves that path with a black-height of 3 from the root. Because the path of length 5 can only have a minimum black height of 3, that means it is impossible for the right subtree to have a BH of 2 and still create a valid RB tree with the left subtree.

- **Case 2:** the right subtree has Black-Height 3:

If the right subtree has a BH of 3, the only way to maintain a BH of 3 along its shortest path to a nil node of length 2 is to have all nodes along that path to be black. This is depicted in the diagram below:

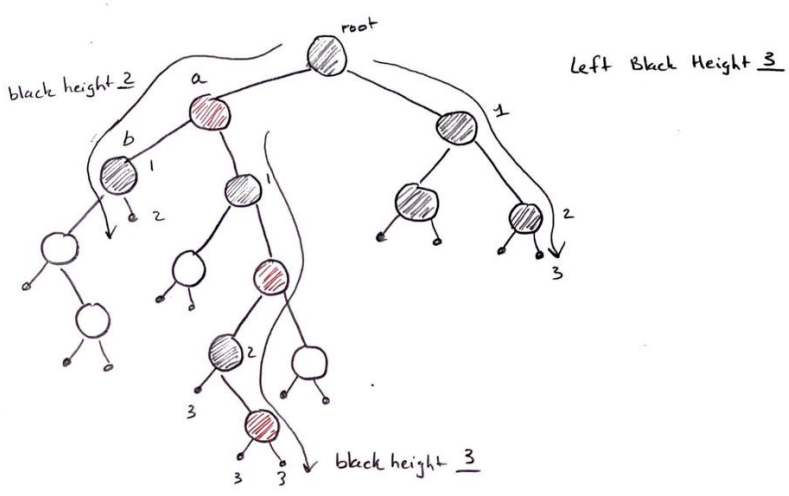



There is no other way to achieve a path of 3 along nodes **a** and **b**, and the NIL node on **b**.

However, if node **a** is black, there is no way to have a black-height of 3 along the longest path of 5. If the first two nodes along that path are black, that leaves 4 nodes remaining. Because we cannot have two adjacent red nodes and still maintain the RB tree invariant, we *must* alternate red and black nodes. Therefore 2 of these 4 remaining nodes must be black. However, that means we would have *at least* 4 black nodes along the longest path, leaving us with a longest path black height of 4. Therefore, once again, it is impossible to have a black height of 3 in the right subtree and a black height of 3 in the left subtree simultaneously.

In fact, the only way to achieve a black height of 3 along the longest path is to have node **a** be red, and to have the remaining nodes alternate red and black. But then, of course, it is impossible to maintain a black height of 3 along **root-a-b-nil**.

Therefore, it is impossible to turn these nodes into an RB tree.



**Question 3: Augmented BSTs**

**a)** Given an unsorted set of *n* distinct numbers, suppose we are interested in carrying out the following operations:

- Selecting the element of rank *k*. This operation will be carries out $1 \leq m \leq n$ times.
- Inserting a new element into the set. This operation will be carried out $1 \leq p \leq n$ times.

The order of the above operations is not known.

In class we saw 3 different ways of solving this problem: one that uses an augmented binary search tree, one that uses the original select algorithm, and a third method which simply sorts the input. For each approach, you must describe how to solve the above two operations and provide the overall runtime.

**Augmented Binary Search Tree**

Elements are stored in a BST. To guarantee $\log n$ height use a self-balancing tree such as a Red-Black tree. Because the tree is self-balancing, we don't need to rebuild the tree after insertion. It maintains itself dynamically. The tree is also augmented with subtree sizes at every node to ensure selection in time $O(\log n)$.

*Building the RD BST:*

- Building a binary search tree takes time $O(n \log n)$.

*Select k:*

- We know from class that selecting an element of rank *k* requires $O(\log n)$ in the worst-case when using a Red-Black tree.
- For *m* select operations, the overall runtime for selection would be $O(m \log n)$.

*Insert element:*

- Similar to select, insertion into a balanced BST also requires time $O(\log n)$.
- Inserting *p* elements would have runtime $O(p \log n)$.

*Overall runtime:*

- The overall runtime for these steps would be: $O(n \log n + m \log n + p \log n)$

**<u>Select algorithm</u>**

Elements are stored in an array. We don't have to perform any additional building steps.

*Select k*:

- The select algorithm takes an array of numbers and a target rank, *k*, and is able to provide the number of the element corresponding to rank *k* in a worst-case runtime of $O(n)$.

- For *m* select operations, the runtime would be $O(mn)$.

*Insert element*:

- Insertion into an array is amortized constant, with worst-case linear insertion time. The expected time per insertion would be $O(1)$.
- For *p* insertions, the overall time would be $O(p)$ where *p* is a constant.

*Overall Runtime:*

- The overall runtime is: $O(mn + p)$.

**Sorting the input (merge sort)**

Elements are stored in an array. In order to be useful for selection, the array must be sorted. Sorting an array with *n* integers with merge-sort would take $O(\mathrm{n} \log n)$ time.

We don't know whether the numbers are uniformly distributed, or that even if they are initially that users will continue to add uniformly distributed values, so we can't necessarily use Bucket Sort. Similarly, we don't know if users will add negative values, so we can't use Counting Sort, or Radix Sort which uses Counting Sort under the hood.

*Building the sorted array*:

- Sorting an array using merge sort takes time $O(n \log n)$

*Select k:*

- Because the array is sorted, we can select any rank by just accessing the array index corresponding to the rank. Accessing an element of an array is constant time: $O(1)$.
- Performing *m* select operations would take time $O(m)$.

*Insert element*:

- Inserting into an array typically takes constant time, $O(1)$. However, in this caes we need to maintain a sorted array after insertion.
- Inserting a single value into a sorted array so that it remains sorted requires finding the insert position, then shifting over some portion of the array. In the worst-case, this takes time $O(n)$ because we either have to shift the entire array or scan the entire array for the insert position.
- For *p* insertions the runtime is $O(pn)$.

*Overall Runtime:*

- The overall runtime would be: $O(n \log n + m + pn)$

**Sorting the input (linear sort)**

- If we *are* allowed to make certain assumptions about the data, we can use a linear sorting algorithm.
- For example, if we know that numbers always originate from a uniformly distributed source, we can use Bucket sort.

*Building the sorted array*

- Sorting an array using a linear sorting algorithm takes time $O(n)$.

*Select k:*

- Because the array is sorted, we can select any rank by just accessing the array index corresponding to the desired rank. Accessing an element of an array takes constant time: $O(1)$.
- Performing *m* select operations would take time $O(m)$.

*Insert element*:

- Inserting into an array typically takes constant time, $O(1)$. However, in this case we need to maintain a sorted array after insertion.
- Inserting a single value into a sorted array so that it remains sorted requires finding the insert position, then shifting over some portion of the array. In the worst-case, this takes time $O(n)$ because we either have to shift the entire array or scan the entire array for the insert position.
- For *p* insertions the runtime is $O(pn)$.

*Overall Runtime:*

- The overall runtime would be: $O(n + m + pn)$

| | **Augmented RB Tree** | **Select** | **Sort (merge sort)** | **Sort (linear sort)** |
|---|---|---|---|---|
| *p, m* | $O(n \log n + m \log n + p \log n)$ | $O(mn + p)$ | $O(n \log n + m + pn)$ | $O(n + m + pn)$ |
| *p = m =* $O(\log n)$ | $O(n \log n + 2\log^2 n)$ = $\boldsymbol{O(n \log n)}$ | $O(\text{nlog}\, n + \log n) = \boldsymbol{O(n \log n)}$ | $O(n \log n + \log n + n \log n)$ = $\boldsymbol{O(n \log n)}$ | $O(n + \log n + n \log n) = \boldsymbol{O(n \log n)}$ |
| *p = 4, m = n* | $O(2n \log n + 4 \log n) = \boldsymbol{O(n \log n)}$ | $O(n^2 + 4) = \boldsymbol{O(n^2)}$ | $O(n \log n + n + 4n) = \boldsymbol{O(n \log n)}$ | $O(n + n + 4n) = \boldsymbol{O(n)}$ |
| *m = 2,* | | | | |

| | | | | |
|---|---|---|---|---|
| *p = n* | $O(n\log n + 2\log + n\log n) = \boldsymbol{O(n\log n)}$ | $O(2n+n) = \boldsymbol{O(n)}$ | $O(n\log n + 2 + n^2) = \boldsymbol{O(n^2)}$ | $O(n+2+n^2) = \boldsymbol{O(n^2)}$ |
| *m = p = n* | $O(n\log n + 2n\log n) = \boldsymbol{O(n\log n)}$ | $O(n^2+n) = \boldsymbol{O(n^2)}$ | $O(n\log n + n + n^2) = \boldsymbol{O(n^2)}$ | $O(n+n+n^2) = \boldsymbol{O(n^2)}$ |

Using the results, decide which method you should use when:

- In case #1, there is a tie between BST, Select and sorting.
- In case #2, the fastest performance is by using a linear sort (if allowed), followed by a tie between BST and merge-sort.
- In cases #3 , the fastest performance is using the Select algorithm.
- In case #4, the augmented BST has the best runtime.

**b)** Suppose we augment a binary search tree so that each node has an additional attribute called *x.leaves* which represents the number of leaves in the subtree rooted at x. Let T be the root of such a BST. Your job is to update the BST-Delete(T, z) algorithm from class so that it correctly deletes node z from the tree T and updates the values of *x.leaves* for all necessary nodes in the tree. You must provide the pseudo-code for the updated version and explain why the runtime is still *O(h)*.

**BST-Delete(T, z)**

```
if z = T and z.left = NIL
        return z.right
else if z = T and z.right = NIL
        return z.left
else
        if z.left = NIL and z.right = NIL
                if z = z.parent.left
                        z.parent.left = NIL
                else
                        z.parent.right = NIL
                if z.parent.right != NIL or z.parent.left != NIL        # if z had a sibling
                        y = z.parent
                        while y != NIL
                                y.leaves = y.leaves – 1
                                y = y.parent
                else
                        z.parent.leaves = z.parent.leaves - 1
        else if z.left = NIL
                Replace(z, z.right)
        else if z.right = NIL
                Replace(z, z.left)
        else
                y = Find-Min(z.right)                                   # successor
                z.key = y.key
                if y.right = NIL
                        if y.parent.right != NIL                        # node has a sibling
                                p = y.parent;
                                while p != NIL
                                        p.leaves = y.leaves – 1
                                        p = p.parent
                        else
                                y.parent.leaves = y.parent.leaves - 1
                Replace(y, y.right)
        return T
```

- Overall we have 3 cases:
    - If the node being deleted is a leaf:
        - We decrement the number of leaves in its parent by 1.
        - If it had no siblings, we are converting it's parent to a leaf. So the overall number of leaves is unchanged.

        - If it did have a sibling, then we have removed a leaf from the tree. We need to decrement the number of leaves for all nodes on the path from root to *z*.
    - If the node being deleted has 1 child:
        - We are not changing the number of leaves in the tree, so this part of the algorithm remains unchanged.
    - If the node being deleted has 2 children:
        - In this case, we find the successor of the node being deleted and actually remove the *successor* from the tree.
        - The successor node can either be a leaf or have a single right child.
            - If the node is a leaf:
                - We decrement its parent by 1.
                - If it has no siblings, we are converting its parent to a leaf. So the overall number of leaves on the path from root to the successor node (except for the parent) is unchanged.
                - If it did have a sibling, we have removed a leaf from the tree. We need to decrement the number of leaves for all nodes on the path from root to the successor node.
            - If the node has a right-child, then the number of leaves in the tree is unchanged, and we don't have to do anything.

- **Overall Runtime**
    - Case 1: When deleting a leaf, in the worst-case we may have to repair leaf count in all the nodes on the path from the root to the leaf being deleted. This is, in the worst-case the longest path of the tree, so has a runtime of $O(h)$. All other operations are constant-time.
    - Case 2: When removing a node with a single child, all operations run in constant time.
    - Case 3: When removing a node with two children, we may be removing a successor node which is a leaf. Similar to the first case, this may require fixing all leaf counts on the path from the root to the successor node. In the worst-case, this takes time $O(h)$. Similarly, finding the successor node in the first place also takes worst-case time $O(h)$. So the overall worst-case time for case 3 is $O(h)$.
    - Therefore, the overall runtime of the delete operation has a worst-case of $O(h)$.

**c)** (8 points) In our practice problems we defined AVL trees. Each node of the AVL tree is augmented with an attribute called the *x.balance-factor* defined as follows: if both subtrees of x have the same height, the balance-factor is 0. Otherwise the balance-factor is 1 (left side is higher) or -1 (right side is higher). Given a tree T that is augmented with this information, update the Tree-Insert(T, z) algorithm from class so that the balance-factors are correctly updated after the insertion. If the resulting tree is not a proper AVL tree after the insertion, you do not need to repair it.
** The height of an empty tree is defined as -1*

```
Tree-Insert(T, z)
        if (T = nil)
                z.balance-factor = 0
                return z
        else
                x = T
                while x != nil
                        y = x
                        if z.key < x.key
                                x = x.left
                        else
                                x = x.right
                z.parent = y
                z.balance-factor = 0
                if z.key < y.key
                        y.left = z
                        y.balance-factor += 1           # adjust the balance-factor of z's parent
                else
                        y.right = z
                        y.balance-factor -= 1           # adjust the balance-factor of z's parent

                # repair balance-factors up to the root, as needed
                while (y.parent != nil AND y.balance-factor != 0)      # if not root and caused imbalance
                        if y.parent.left = y
                                y.parent.balance-factor += 1
                        else
                                y.parent.balance-factor -= 1
                        y = y.parent
                return T
```

**d)** A project manager would like to store a set of n project intervals. Each interval consists of a start and end time (over a year-long period). The manager would like a data structure that organizes the project intervals in such a way that she can carry out the following operations:

**Data Structure**: use a red-black interval tree augmented with subtree sizes.

- Because it is a RB tree, we are guaranteed a tree height of $O(\log n)$.
- Because it is an interval tree, we can store project intervals, i.e. the start and end-times, as well as the max end-time for a subtree.
- The interval tree's BST invariant is maintained by the *x.int.low* property, as we've seen in class.
- Construction time is $O(n \log n)$. We know from class that the time it takes to insert into a RB tree is $O(\log n)$. Because we have *n* elements to insert, the total build time is $O(\log n)$ multiplied by *n* inserts, so $O(\mathrm{n} \log n)$ overall.
- Because we have subtree sizes, we can quickly perform operations like rank, select and range queries.

1. Given a **new project interval**, *i*, determine if project interval i overlaps the project that starts *last* among all the project. Time $O(\log n)$.

- The project that starts last has the greatest *x.int.low* value from all values in the tree.
- Because the tree uses *x.int.low* as the key to satisfy the BST property, the project the starts last is the *max* (rightmost) element in the tree.
- To find the maximum element in the tree, we have to traverse the entire height of the tree, from root to leaf, in the worst-case. Because we are maintaining a RB tree, we know that this height is guaranteed to be $O(\log n)$. Therefore, finding the maximum element in the tree takes time $O(\log n)$.
- Once we have the maximum element (the project starting last among all projects), we can compare it directly to the new project interval, *i*, to see whether there is an overlap. This requires constant time.
- Therefore, overall, the runtime for determining if project interval i overlaps the project that starts *last* among all the project is $O(\log n)$.

2. Given a project start time *t*, determine if there is a project that starts at exactly time t.

- The project start time is the property *x.int.low* in the tree.
- Given a project with start time *t*, to determine if there is already a project in the tree with the same start time, we have to perform an Interval Tree Search for an element where *t* = *x.int.low*.
- We search the tree using the fact that the BST property is maintained by *x.int.low*. If *t* < *x.int.low*, we search left. Otherwise, we search right.
- Once we find the element where *x.int.low* = t, we can return true. Otherwise, if we run out of nodes to search, we return false.
- In the worst-case we have to search all nodes from root to leaf that correspond to the longest path in the tree. We know that with a RB tree in the worst-case this is $O(\log n)$. Therefore, the time required to perform this search operation is $O(\log n)$.

3. Given a start time *t*, where *t* is the start time of some project in the set, the goal is to determine *the next project* to start after time t. Time $O(\log n)$.

- Given some start time, we want to know the *next project* to start after time *t.* That is to say, we want to find the *next closest project start time* to *t*.
- Because we know that that *t* corresponds to an interval (project), *i*, already in our set (tree), this is the same thing as saying that we want to find a *successor* to *i*.
- Before we find a successor, though, we need to find the location of interval *i*. We conduct a search for *i* using the steps outlined in above in 2. This takes time $O(\log n)$.
- Once we have interval *i* we can conduct a successor search:
    - If *i* has a right subtree, we look for an interval $i_2$ that is the minimum element of the right subtree. In other words, we would do *findMin(i.right)*.
    - If *i* does not have a right subtree, we need to look for the successor from the root. We look from the root to interval *i* keeping track of intervals where *i2.int.low* > *i.int.low*. We update to make sure we take the smallest value of *i2.int.low* that is still larger than *i.int.low* as we encounter it.
- Because in the worst case, as before, our search would need to be performed from root to leaf along the longest path of the tree, we know that the worst-case run time of performing a successor search is $O(\log n)$, given that we have a RB tree.
- ALTERNATIVELY:
- First, we use TreeSearch using time *t* to find the interval corresponding to time *t*. Call this interval *i*. This step takes time $O(\log n)$.
- Next, we can use the Rank algorithm to determine the rank of the interval *i* that starts at time *t.* Let's call this rank *r*. This step also takes $O(\log n)$.
- Next, we use the Select algorithm to select the interval with rank *r + 1*. This is the next interval to come after *i.* Because our tree uses *i.int.low* as the key, this is then also the next project to start after interval *i* at start time *t*. The Select algorithm also takes time $O(\log n)$.
- Because both of these steps take time $O(\log n)$, the entire process takes time $O(\log n)$. Note, Select and Rank can run in time $O(\log n)$ because we maintain subtree sizes in our data structure.

4. Given a time *t*, output all projects that starts after time *t*. Time: $O(k + \log n)$ where *k* is the number of projects that start after time *t*.

- To output all projects starting after time t, we need a modification of the range query.
- The idea here is to first find the first element that has a start time (*x.int.low)* greater than time *t*. Call this node *v*. This search takes time $O(\log n)$
- Next, we perform a search for all elements on *v*'s left subtree for all elements where *x.int.low* is greater than *t*. If we encounter a node where *x.int.low* is greater than *t*, we output that element and run an *in-order traversal* on the node's right subtree; finally we search left. Otherwise, we search right. We continue searching until we either encounter an element where *x.int.low = t* or reach a leaf node.
- Because in the worst-case this search is along the longest path, and we perform a constant amount of work at each encountered node, this takes time $O(\log n)$ in the worst-case for a self-balancing tree.
- Finally, we need to output all element's to *v*'s right, all of which have a start time greater (*x.int.low)* greater than *t*. We can do this using an *in-order traversal*. Because the number of elements in-scope for the in-order traversal is at most *k* elements, this takes time $O(k)$.

- Ultimately, we conduct two search operations both taking time $O(\log n)$ and traversal operations that collectively take at most time $O(k)$. Therefore, the entire runtime is therefore $O(\mathrm{k} + \log n)$.
- Note: this is essentially the LargerThan algorithm we saw in the practice set.

5. Output the first 5 projects to start. Time: $O(\log n)$.

- Because we are storing subtree heights at each node, we can simply run the select 5 times, in sequence.
- Specifically, we run the Select algorithm for ranks: 1, 2, 3, 4 and 5.
- Because our tree is keyed by *x.int.low*, these 5 ranks will correspond to the first five projects with the smallest – i.e. earliest – start times. These are the first 5 projects to start.
- Because we have a RB Tree, the select algorithm runs in time $O(\log n)$. Doing it 5 times in succession will result int time $O(5\log n)$ which is still asymptotically $O(\log n)$.

6. Output the project with the latest finish time: $O(\log n)$.

- The max value of the root (*x.max* in our data structure) comes from the element with the greatest *x.int.high* value.
- This value corresponds to the project that has the latest finish time.
- Knowing the value of *x.int.high* that we are interested in, we have to conduct a search for the interval that has that *x.int.high* value. This is the interval (project) we want to output.
- To search for this project, we need to search using *k = root.max* and interval *i*:
    - if *i.int.high* = *k*, return I (we have found the project with the latest start date)
    - else if *x.left != nil* and *x.left.max* = k, search left, max must have come from this tree
    - otherwise, search right
- Because this search, in the worst-case, has to search through the longest path of the tree, we know that the runtime is at worst $O(\log n)$ because we know the height of the tree is at most $O(\log n)$.
- Therefore, the runtime of the entire algorithm is $O(\log n)$.

7. Output the pair of projects whose start times are closest together. Time: $O(n)$.

- To output the pair of projects whose start times are closest together, we need to find the *gap* between successive pairs of projects.
    - I.e. if P1 has start time s1, and P2 has start time s2, the gap is given by: $|s1 - s2|$
- First, we perform an *inorder* traversal of the tree, outputting elements into an array. An *inorder* traversal outputs elements in a BST in sorted order. Because our tree is keyed by *x.int.low*, the projects are ordered from smallest to greatest start time. We have to visit each element once performing constant work, so this takes time $O(n)$.
- Next, we loop through the sorted elements in the array to find the pair of start times with the smallest gap. This corresponds to the start times that are closest together.
    - As we loop through the array, we keep track of the *minimum* gap between two adjacent projects. If we encounter a pair of adjacent projects that has a smaller gap than the current minimum, we update the current minimum and the pair of projects. Because we loop through all elements of the array performing constant work, this step takes time $O(n)$.

- Finally, we can return the pair of elements that correspond to the minimum. These are the projects whose start times are closest together.
- Because both steps take time $O(n)$ the overall runtime of this process is $O(n)$.

8. Output the project that starts immediately before project *x*. Time: $O(log\ n)$.

- Because our RB interval tree is keyed using *x.int.low*, i.e. the project start times, the project that starts immediately before project *x* is project *x's* immediate *predecessor.*
- Finding the predecessor is analogous to searching for the successor; both operations run in time $O(log\ n)$.
- If *x* has a left subtree, the predecessor is the maximum value of the left subtree. In other words, findMax(x.left).
- If *x* does not have a left subtree, we have to search from the root. If *i.int.low < x.int.low*, we keep track of *i* and search right. Otherwise, we search left.
- We keep track of all nodes we encounter where *i.int.low < x.int.low*, keeping track of the maximum value of *i.int.low* we encounter that is still less than smaller than *x.int.low*.
- We stop when *i = x*.
- Because in the worst-case we are searching the longest path of the tree, because we have a RB tree we know this is at most $O(\log n)$.
- ALTERNATIVELY:
- Knowing project *x* we can use the Rank algorithm to get the rank of *x*. Let's call this rank *r*.
- Next, we use the Select algorithm to get the interval associated with rank *r – 1*. This is the interval directly preceding *x*.
- Because this tree uses *i.int.low* as the key, this is the project that start immediately before *x*.
- Because we maintain subtree sizes in our data structure, both Rank and Select algorithms run in time $O(\log n)$.

9. Output the number of projects that start between project *x*'s start time, and project *y*'s start time. Time: $O(log\ n)$.

- Because our data structure is keyed on project start times and maintains subtree sizes at each node, we can use a *range* operation to select all projects that start between project *x*'s start time and project *y*'s start time.
- If we say that project *x*'s start time is *a* and project *y*'s start time is *b*, we want the number of projects (intervals) where $a \leq\ x.int.low \leq b$. This is exactly what the range query gives us.
- Because the problem does not explicitly state otherwise, assume that the start time bounds are inclusive.
- Using the standard range approach, we first locate the first interval where *x.int.low* is between *a* and *b*. Call this node *v*. The worst-case time for this search, as we've described above, is $O(\log n)$.
- Once we have node *v*, we carry out two searches. First, we search the left subtree for all nodes where *x.int.low* is *greater than or equal to a*. We maintain a running sum, *left-total* that counts each node in the path that is greater than or equal to a, as well as that node's right subtree (if it exists). We continue searching the path until we either find the project with *x.int.low = a* or reach a leaf node. Because the search in worst-case is along the longest path in the tree, this takes time $O(\log n)$ for a self-balancing tree.

- Next, we search the right subtree for all nodes where *x.int.low* is smaller than or equal to *b*. We maintain a running sum, *right-total* that counts each node in the path that is less than or equal to b, as well as that node's left subtree (if it exists). We continue searching the path until we find the node with *x.int.low = b* or reach a leaf. Because the search in worst-case is along the longest path in the tree, this takes time $O(\log n)$ for a self-balancing tree.
- Finally, we return *left-total* + *right-total* + 1 as the number of projects that start between project *x*'s start time and project *y*'s start time.
- Because all 3 search operations take time $O(\log n)$ and all other operations are constant, the overall runtime for this algorithm takes time $O(\log n)$.

**Q4: Dynamic Programming**

**a) Update Problem 3 from practice set 10 in such a way that the output is now the number of different ways we can select a subset of size T using the elements with weights w[1, …, n]. You do not need to keep track of the selected weights.**

- **Define the table you are using, and define each entry.**

The table used will be M[0 … n, 0 … T]. It stores values representing the ways that we can select a subset of weights that add to 0 .. T. Each entry M[i, j] will represent the *ways to select a subset of weights 0 through i such that they add up to size j.*

Rows (i) represent weights up to n. Columns (j) represent sizes up to T.

- **Describe how to initialize your table.**

Given 0 items and size j = 0, there is exactly 1 way to select a subset of size j.

Given 0 items, there are 0 ways to select a subset of size j for j > 0.

Given size j = 0, there is exactly one way to select a subset of items of size j; namely, by selecting no items at all (the empty set).

```
M[0, 0] = 1  # there is exactly 1 way to make size 0 with 0 items
for i = 1 to n
        M[i, 0] = 1          # there is 1 way to make size 0 with any items (don't use any items)
for j = 1 to T
        M[0, j] = 0          # there are no ways of making sizes 1 .. n using 0 items
```

- **Describe the relationship between the entries in your table.**

Each entry M[i, j] represents the ways we can have size *j* using weights 0 through i. More specifically, each entry represents two cases: we include $w_i$ in the subset, or we do not include $w_i$ in the subset. That is, M[i, j] is the sum of including $w_i$ (M[i – 1, j – w[i]]) and not including $w_i$ (M[i – 1, j]).

- **Describe which entry in the table stores your final result.**

Entry M[n, T] stores the final result. That is, it stores all the ways to have a size of T using weights [0 .. n].

- **Provide the pseudocode that shows how to fill up your table.**

```
waysToMakeSizeT(w[0..n])

# initialize the table
M[0, 0] = 1  # there is exactly
for i = 1 to n
        M[i, 0] = 1
for j = 1 to T
```

```
        M[0, j] = 0

# fill table top-to-down
for i = 1 to n
        # fill table left-to-right
        for j = 1 to T
                if j < w[i]
                        M[i, j] = M[i – 1, j]
                else
                        M[i, j] = M[i -1, j – w[i]] + M[i – 1, j]

return M[n ,T]
```

- **Justify the runtime of your algorithm.**

There are two loops. One outer loop running from 1 to n (the total number of distinct weights), and one inner loop running from 1 to T (the number of sizes between 1 and T). All other work is constant. Therefore the runtime is $O(nT)$.

**b)**

- **Define the table you are using, and define each entry.**

We will have a 2D table, M[0 … n, 0 … m].

Each cell M[i, j] represents the *maximum profit possible* from the set of weights [0 .. i] that add to *exactly* weight j.

Note: *i* represents rows (weight sets) and *j* represents columns (sizes).

To do this, we essentially want to distinguish between the case where we are *allowed* to use weight *I* from when we are *not allowed* to use weight *i*.

We are only allowed to use w[i] when the set of weights *including* w[i] add to j. That is, if we *use* w[i] then M[i – 1, j – w[i]] was also a weight we could add to *without* w[i]. This is only possible when there are weights that add up exactly to M[i – 1, j – w[i]], not including w[i].

For example, if there is some profit associated with M[i – 1, j – w[i]], then we know there must have been some weights that added up to exactly j – w[i]. In that case, we can take the maximum of M[i – 1, j – w[i]] + p[i] and M[i – 1, j]. Otherwise, we *cannot* use w[i] and just set M[i, j] to M[i – 1, j].

The above shows that we essentially need some way to indicate that there are no weights that add up to j – w[i]. One way to do this is by setting those cells to be -∞. This has the nice property that it will automatically award *maximum(M[i – 1, j – w[i]] + p[i], M[i – 1, j]) = M[i – 1, j]* when M[i – 1, j] is not negative infinity.

Note: the above assumes that any value added to -∞ still yields negative infinity. So maximum(-∞ + p[i], -∞) = -∞.

- **Describe how to initialize your table.**

For 0 capacity and 0 weight, the total profit is 0.
If we have no capacity, then the total profit possible for any set of weights is 0.
If we have no weights, than the total possible profit for any capacity is -∞.

```
M[0, 0] = 0
for i = 1 to n
        M[i, 0] = 0
for j = 1 to m
        M[0, j] = -∞
```

- **Describe the relationship between the entries in your table.**

Each entry represents the total profit possible from weights that total *exactly* capacity j. If no exact addition is possible, the value is set as -∞. Each cell is the maximum result of using the weight of a given row given the capacity of a given column, or not using it.

- **Describe which entry in the table stores your final result.**

Table entry M[n, m] gives the final result. That is, for some capacity m and the set of weights n, this is the total maximum profit possible from weights w[0 .. n] that add up to exactly m.

- **Provide the pseudocode that shows how to fill up your table.**

MaxProfitExactWeight(w[0. . i], p[0 .. i])

```
M[0, 0] = 0
for i = 1 to n
        M[i, 0] = 0
for j = 1 to m
        M[0, j] = -∞

for i = 1 to n
        for j = 1 to m
                if w[i] > j
                        M[i, j] = M[i – 1, j]
                else
                        M[i, j] = maximum(M[i - 1, j – w[i]] + p[i], M[i – 1, j])
return M[n, m]
```

- **Justify the runtime of your algorithm.**

We loop through all n rows and m columns to construct the table. There are two nested loops: the outer loop goes from 0 to n, while the inner loop goes through 0 .. m. Therefore the runtime is $O(nm)$.

**c)**

- Define the table you are using

Table M[0 .. n, 0 .. n] stores the maximum number of flies available along the different paths, or a value of -1 if the lily-pad contains a bird. Specifically, there are two cases:

  - Entry M[i, j] stores the maximum number of flies it is possible to eat upon reaching cell M[i, j] including the flies at M[i, j]
  - Entry M[i, j] stores a bird, in which case entry M[i, j] stores -1 to indicate that we cannot land on this cell

Note: *i* represents a column, *j* represents a row

- Describe how to initialize your table

The table needs to have the last row and first two columns initialized. We have to initialize the first two columns because a cell M[i, j] in the table depends on both cells M[i -1, j] and M[i -2, j]. It also depends on M[i, j - 1] so we initialize the first row. The actual initialization code is provided with the pseudo code below.

- Describe the relationship between the entries in your table.

Entry M[i, j] can be reached from entries M[i – 1, j] (one column to the left), M[i – 2, j] (two columns to the left), and M[i, j - 1] (one row down).

- Describe which entry in the table stores your final result.

Entry M[n, n] stores the final result. That is, the upper-rightmost cell (stump).

- Provide the pseudo code that shows how to fill up your table

```
LilyPads(L[1 .. n, 1 .. n])
        # initialize table
        Initialize M [1 .. n, 1 .. n]
        M[1, 1] = L[1, 1]
        M[2, 1] = L[2, 1]

        # initialize first row
        for i = 3 to n
                if L[i, 1] = -1
                        M[i, 1] = -1
                else if M[i – 1, 1] = -1 and M[i – 2, 1] = -1
                        M[i, 1] = -1
                else
                        M[i, 1] = L[i, 1] + maximum(M[i – 1, 1], M[i – 2, j])

        # initialize first column
        for j = 2 to n
```

```
        if L[1, j] = -1
                M[1, j] = -1
        else if L[1, j - 1] = -1
                M[1, j] = L[1, j]
        else
                M[1, j] = L[1, j] + M[1, j - 1]

# initialize second column
for j = 2 to n
        if L[2, j] = -1
                M[2, j] = -1
        else if L[i – 1, j] = -1 and L[i, j - 1] = -1
                M[2, j] = -1
        else
                M[2, j] = L[2, j] + maximum(M[i – 1, j], M[i, j - 1])

# fill table
for i = 3 to n
        for j = 2 to n
                if L[i, j] = -1
                        M[i, j] = -1
                else if L[i – 1, j] = -1 and L[i – 2, j] = -1 and L[i, j - 1] = -1
                        M[i, j] = -1
                else
                        M[i, j] = L[i, j] + maximum(M[i – 1, j], M[i – 2, j], M[i, j - 1])

return M[n, n]   # maximum number of flies possible
```

- Justify the runtime

  We have to create a table of size n by n. We do this by initializing columns 1, 2 and row 1, and then using two nested for-loops, the outer running from 3 to n, and the inner running from 2 to n. This results in a runtime of $O(n^2)$, similar to insertion.