assignment 2

Question 1

Part A

If you remove the student IDs and only consider the names, what do you notice about the order/structure?

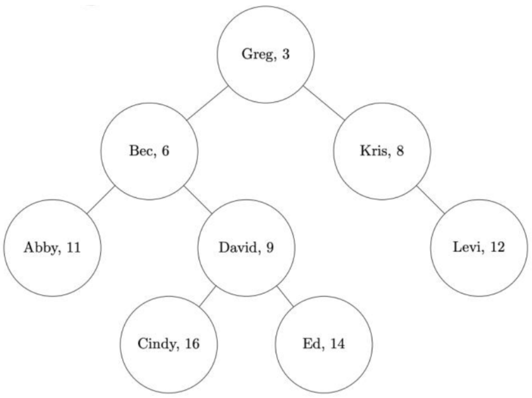
The name of a node is alphabetically greater than all nodes on the left, and smaller than all nodes on the right.

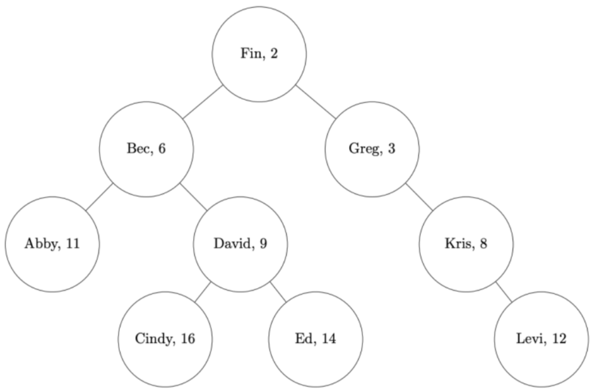
If you remove the names and only consider the student IDs, what do you notice about the order/structure?

The StudentID of a node is always smaller than it’s children.

Part B

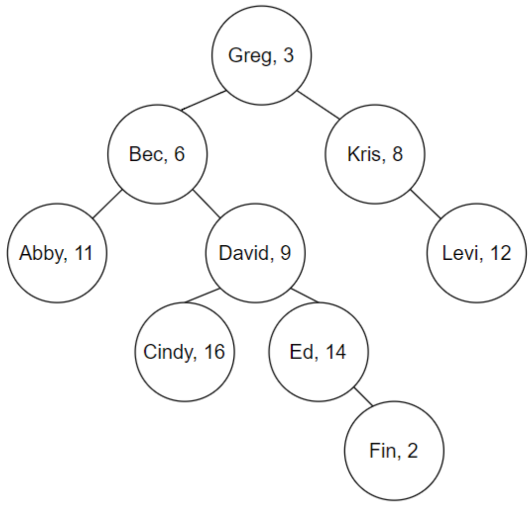
If we add the student Fin who has a student ID of 2, the *QUBSET* changes from left to right.



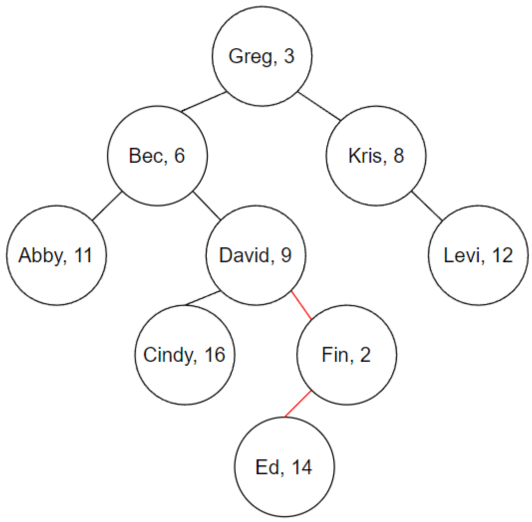


▼ Write down all the missing steps in this process. You should provide just as much detail as the examples shown at the bottom. **2 marks**

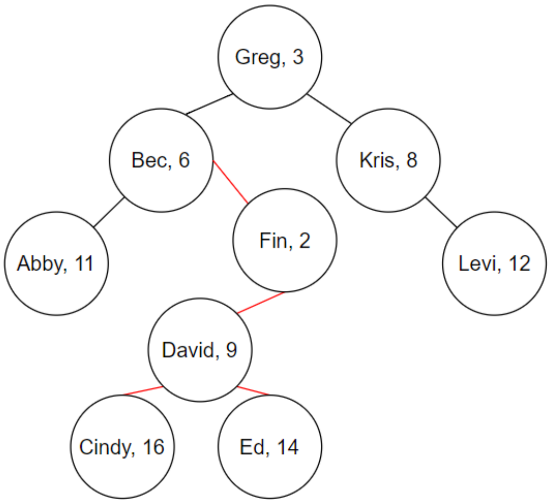
1. Using alphabetic comparisons, find the appropriate location for Fin.



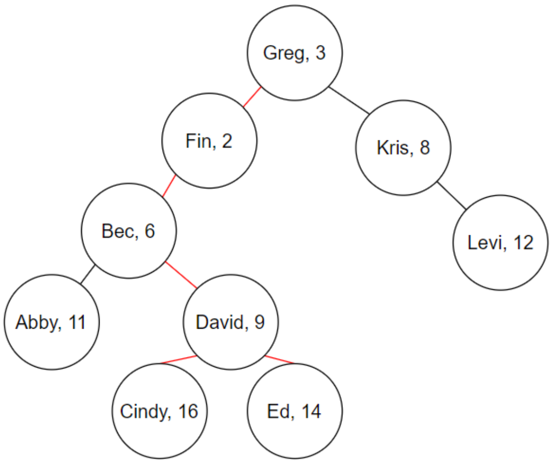
2. Left Rotation on Ed



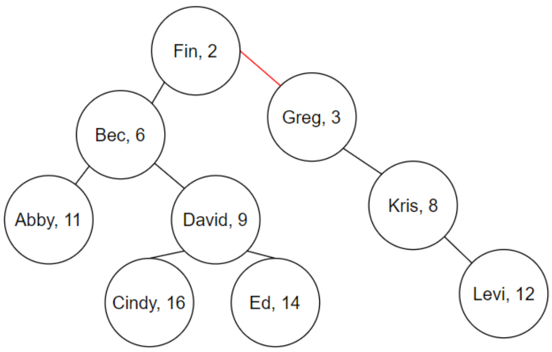
3. Left Rotation on David



4. Left Rotation on Bec



5. Right Rotation on Greg



Part C

▼ Given an arbitrary *QUBSET*, *T*, and a new student *S*, write a new function `add_student( T , S )` that adds the new student to the diagram. Assume the operation is done in place (there should be no return value). You can assume $T name$ and $T id$ give the student name/ID respectively. $T parent$ gives the parents. *T* and *S* are of the same type and you can assume *S* has no children. Your pseudocode should look like the pseudocode that is given in Lecture 9. **3 marks**

```
function add_student(T, S):
  T, path ← bst_add(T, S)
  T ← rotate_up(T, S, path)

// Standard Recursive Binary Search Tree Insertion
function bst_add(T, S, path=[]):
  if T = NULL do
    return S

  // `path` stores the steps taken to reach target node.
  // Structure: [NODE, DIRECTION]
  path.append([T, T.name > S.name ? 'L' : 'R'])

  if T.name > S.name do
    T.left ← bst_add(T.left, S, path)
  else
    T.right ← bst_add(T.right, S, path)

  return T, path

// Once inserted, use rotations so the rule from part a) ii is matched.
// "path" is used here instead of keeping track of parents in each node.
function rotate_up(T, S, path):
  for i ← len(path) - 1 down to -1 do
    parent_node ← path[i]
    curr_node ← path[i + 1]

    // If node reached from right leaf, rotate left on it's parent
    rotation_dir ← (curr_node[1] = 'R') ? 'L' : 'R'
    parent_dir ← parent_node[1]

    // Iteratively rotate until the parent node's student id is smaller.
    if S.id < curr_node[0].id then
      if i = 0 then
        T ← rotate(T, rotation_dir)
      else if parent_dir ='R' then
        parent_node[0].right ← rotate(parent_node[0].right, r_dir)
      else
        parent_node[0].left ← rotate(parent_node[0].left, r_dir)

    return T

function rotate(N, rotation_dir):
```

```
  return (rotation_dir = 'L') ? l_rotate(N) : r_rotate(N)

function r_rotate(N):
  M ← N.left
  N.left ← M.right
  M.right ← N
  return M

function l_rotate(N):
  M ← N.right
  N.right ← M.left
  M.left ← N
  return M
```

Part D

▼ We want the height of *QUBSET* to be as small as possible. Give an example of the worst case height when we add 5 students to an empty *QUBSET*. **1 mark**

- The worse case height occurs when the tree becomes a linked list. (i.e. only one side of the leaf has items.
- An example would be students `(A, 1), (B, 2), (C, 3), (D, 4), (E, 5)`
- The worst case height would be 4.

```
(A, 1)
    \
    (B, 2)
        \
        (C, 3)
            \
            (D, 4)
                \
                (E, 5)
```

Part E

▼ Suppose we have a group of *n* students that are listed in alphabetical order. If we add them to a binary search tree (sorted just by name and ignoring their student IDs), it can be shown that it degenerates to a tree of height *n*. Explain why the *QUBSET* we have used in this question is likely to have a height much smaller than *n*. **1 mark**

- The advantage QUBSET has is that student IDs are also considered when inserting a student.
- For the example shown in part D, we can achieve an best case height of 2 by modifying the Student IDs.

```
             (C, 1)
              /  \
       (B, 2)     (D, 4)
        /              \
(A, 3)                 (E, 5)
```

- However, our QUBSET can also degenerate into a tree of height n if both the student ids and names are sorted. (as shown in Part D)

```
(A, 1)
    \
    (B, 2)
        \
        (C, 3)
            \
            (D, 4)
                \
                (E, 5)
```

Question 2

Part C - Complexity: $O(C^{(n-1)})$

1. Generate all possible sequence of colors.

- First word only has one color. `O(C)` to find the most scored color.
- The rest of the words have `≤C` colors.

- Generating all the possible results: `O(C ^ (n - 1))`

2. Calculate their scores
    - Because a dictionary can be implemented, the lookup is `O(1)` for both. (https://edstem.org/au/courses/11598/discussion/1379720?comment=3126886)
    - `O(1+1)` to look up the CTT and the WC tables.
3. Select the one with the maximum score.
    - Can be done during step 1 & 2 - no additional cost.

Part D

$$F(n) = max_{c \in C}\{max_{prev_c \in C}[F(n-1) + CT(prev_c, c) + WC(n, c)]\}$$

- `F(n)` is the score of a given word in the sequence.
- `F(n-1)` refers to the score of the previous word.

Part G - Complexity: $O(C^2 * (n-1))$

- First word: `O(C)` lookups.
- Second to last word:
    - First word: `O(C)` to find the WC score. No CTT score possible.
    - Total to compute table: `O((n-1) * (C * (C * 2)))`
        - Second to last word: `O(C * (C * 2))` to check all previous colors.
    - Part F Backtracking: `O(n)`