Question 1: Quasi-balanced Search Trees

Part A

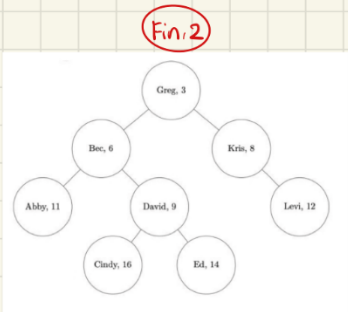
(i.) By considering the names only, we examine the relationship of the **alphabetical order** between parent and its child node.
Here, the right child always has higher alphabetical precedence than its parent node (i.e: **S**hane > **R**uben; **R**owena > **M**elody; **J**ohn > **A**my), similarly the parent node always has higher alphabetical precedence than its left child node (i.e: **R**uben > **L**eo; **D**ana > **A**ngelica; **R**ebecca > **A**my). After checking all three QUBSETs, we conclude the common relationship: left child node < parent node < right child node.
Therefore, those QUBSETs represent the **structure of binary search tree**.

(ii.) By considering the numeric student IDs only, we examine the relationship of the **value order** between parent and its child node.
Here, the value of parent node is always smaller than its child nodes. (i.e 357 < 359 && 357 < 358; 112 < 143 && 112 < 152; 006 < 343 && 006 < 117 && 117 < 984).
After checking all three QUBSETs, we conclude the common relationship: parent node < child nodes.
Therefore, hose QUBSETs represent the **structure of minheap.**

Now, by combing two requirements together, the quasi-balanced search tree (QUBSET) is satisfied if simultaneously its name follows the order of binary search tree (left < parent < right), and its student ID follows the order of minheap (parent < children).

Part B

1. To insert the new node to the QUBSET, we firstly consider following the rule in part A(i) based on the order of **name of student**, regardless of the value of the student ID. Therefore, we perform the **binary search tree insertion:**

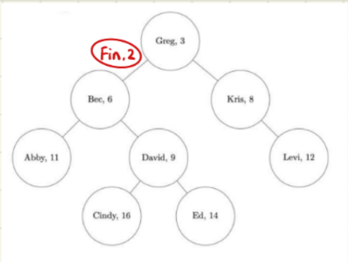


* Note: None of the existing node is Empty *

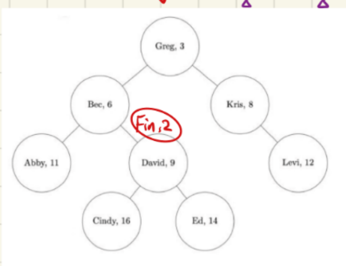
Based on the order of name:

· Inserting Fin, 2 follow by BST rule.

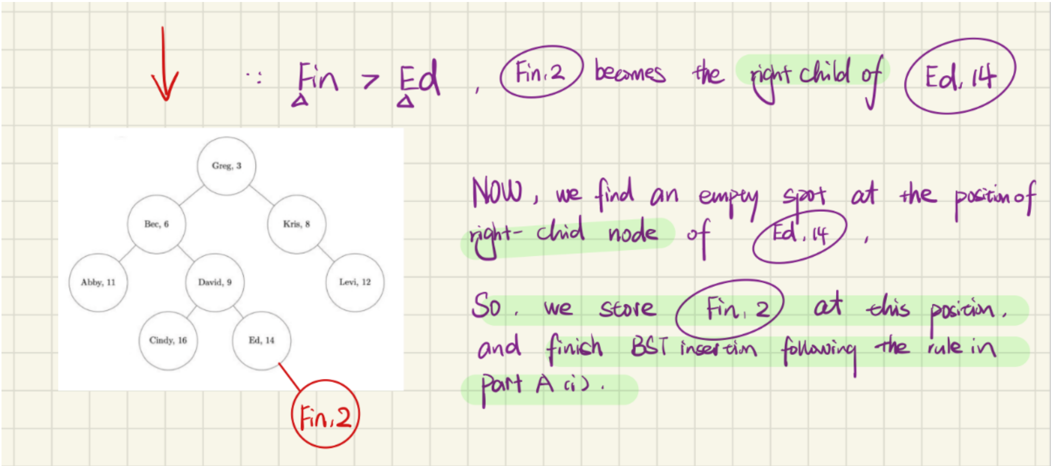
∵ Fin < Greg, Fin, 2 becomes the left child of Greg, 3



∵ Fin > Bec, Fin, 2 becomes the right child of Bec, 6



∵ Fin > David, Fin, 2 becomes the right child of David, 9



2. After fulfilling the correct order of name of the QUBSET, we continue to update the QUBSET following the rule in part A(ii) based on the order of **value of student ID** until every node satisfies the requirement of minheap (value of parent is always smaller than the value of child nodes).

   However, as we also need to maintain the order of name following the rule of part A (i), we cannot simply swap the child node and parent node. Instead, both **upheap and rotation** need to be applied to the QUBSET.

   The idea of rotation is inspired by **AVL Tree**: When handling an imbalanced tree, this strategy updates the child node to become the parent of its previous parent node while maintaining the order of binary search tree.
   The **rule of rotation** is designed as following to deal with imbalance:
   (1.) **Left rotation**: If the inserted node is the **right child node** of its current parent node.
   (2.) **Right rotation**: If the inserted node is the **left child node** of its current parent node.

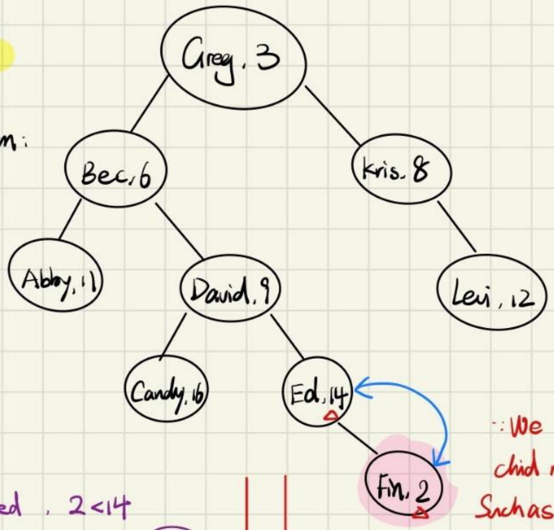
   The diagram below displays the insertion in detail:

After this, the tree does not follow the rule in part A (ii).

We continue to update the diagram until it follows the QUBSET rule.

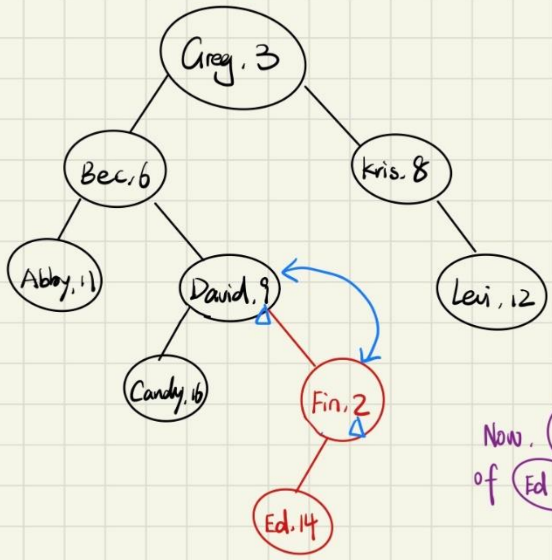
From previous step,

We got this diagram:



∵ We need
child nodes > parent node
Such as: 2 ≯ 14
∴ Need Upheap && Rotation

∵ imbalanced, 2 < 14
Fin, 2 is the right-child of Ed, 14
∴ Apply Left Rotation

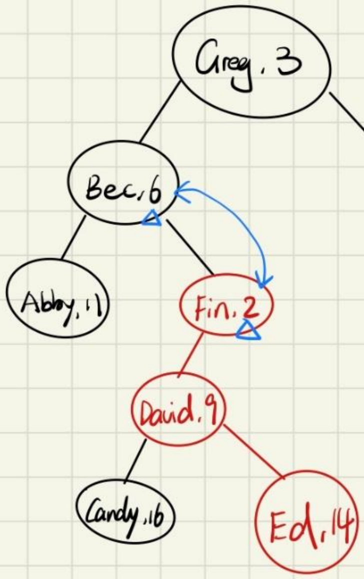


Now, Fin, 2 becomes the parent of Ed, 14 and Ed < Fin
(right < parent)

∵ imbalanced, $2 < 9$

Fin.2 is the right-child of David.9

∴ Apply left Rotation



Notice David.9 becomes the new left-child node of Fin.2,

Ed.14 will now become the right child of David.9

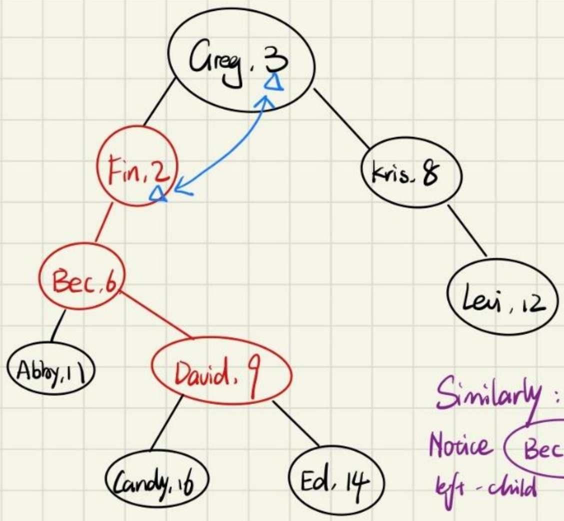
∴ We have:

- Fin.2 becomes the parent of David.9 and David < Fin
  (right < Parent)
- David.2 becomes the parent of Ed.14 and David < Ed
  (parent < left)

∵ imbalanced , $2 < 6$

Fin.2 is the right-child of Bec.6

∴ Apply Left Rotation



Similarly:

Notice Bec, 6 becomes the new left-child node of Fin, 2 ,

David, 9 will now become the right child of Bec, 6

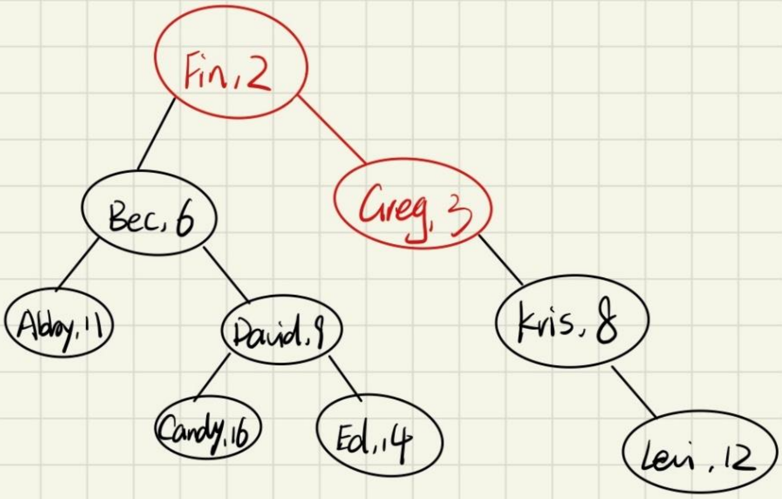
∴ We have:

- Fin, 2 becomes the parent of Bec, 6 and Bec < Fin (right < Parent)
- Bec, 6 becomes the parent of David, 9 and Bec < David (parent < left)

∵ imbalanced , $2 < 3$

Fin,2 is the left-child of Greg,3

∴ Apply Right Rotation

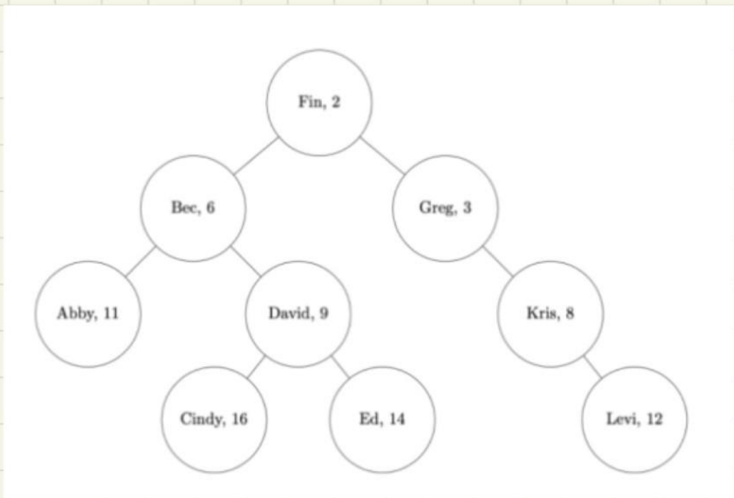


Notice:

· Originally, Fin,2 does not have right-child node, hence there is no extra rotation needed.

Finally: Fin, 2 becomes the root of the QUBSET.

This tree satisfies the answer that has been provided to us:



Part C

Firstly, we make the following assumptions:

Assuming both T and S are nodes of the **same type representing students.**
And we have those structure members:

```
// From the question
Node.name        // return the name of the node
Node.id          // return the id number of the node


// From the assumption
Node.parent      // return the parent of the node
Node.left        // return the left-child of the node
Node.right       // return the right-child of the node
```

Here is the pseudocode:

```
function add_student(T, S)

    // apply binary search tree insertion based on the name of the node
    T = insert_by_name(T, S)

    // apply upheap and rotation based on the id value of the node for updates
    update_by_id (T, S)
```

We design two helper functions as follows:

```
// use recursive function to implement the BST insertion
function insert_by_name(T, S)

    // base case
    // T is null, which indicates an empty subtree to insert S
    if T is empty then
        // S is returned with the given value
        return S

    // recursive case
```

```
    // T is not null. S needs to be inserted either in the left or right subtree
    if T.name > S.name then
        // insert S into the left subtree of T
        T.left = insert_by_name(T.left, S)
    else
        // insert S into the right subtree of T
        T.right = insert_by_name(T.right, S)
    // return the tree after insertion
    return T


function update_by_id(T, S)

    // iteratively update the tree until the relationship between S and its
    // existing parent node satisfies the order of id value.
    while S.parent is non-empty and S.id < S.parent.id do

        // apply rotation when children id < parent id
        // S is the left-child node of its parent
        if S == S.parent.left then
            // apply right rotation
            // define the specific pointers relative to S
            GRANDPARENT = S.parent.parent
            PARENT = S.parent
            RIGHT = S.right

            // update original parent to become the right child of S
            S.right = PARENT
            // update S to the higher precedence
            PARENT.parent = S

            // check whether grandparent of S is not null
            if GRANDPARENT is not empty then
                // update the child of original grandparent to become S
                if PARENT == GRANDPARENT.left then
                    GRANDPARENT.left = S
                else
                    GRANDPARENT.right = S
            S.parent = GRANDPARENT

            // update the right child of S to become the left child of original parent
            PARENT.left = RIGHT
            // check whether right child of S is not null
            if RIGHT is not empty then
```

```
        // update original parent to become the parent of S' right child
        RIGHT.parent = PARENT

// S is the right-child node of its parent
else
        // apply left rotation
        // define the specific pointers relative to S
        GRANDPARENT = S.parent.parent
        PARENT = S.parent
        LEFT = S.left

        // update original parent to become the left child of S
        S.left = PARENT
        // update S to the higher precedence
        PARENT.parent = S

        // check whether grandparent of S is not null
        if GRANDPARENT is not empty then
                // update the child of original grandparent to become S
                if PARENT == GRANDPARENT.left then
                        GRANDPARENT.left = S
                else
                        GRANDPARENT.right = S
        S.parent = GRANDPARENT

        // update the left child of S to become the right child of original parent
        PARENT.right = LEFT
        // check whether left child of S is not null
        if LEFT is not empty then
                // update original parent to become the parent of S' left child
                LEFT.parent = PARENT
```

Rotation Idea:

Some structure members we have:

1. S (new node)
2. T (tree)

3. LEFT
4. RIGHT
5. PARENT
6. GRANDPARENT

Things we need to check during rotation

1. check if grandparent of S is NULL ?

2. check if right-child of S is NULL ?

① Right - Rotation

children updates

① S. right = PARENT
② PARENT. left = RIGHT
③ if GRANDPARENT != NULL
check PARENT on which side,
then assign S to that specific side
of GRANDPARENT.

parent updates

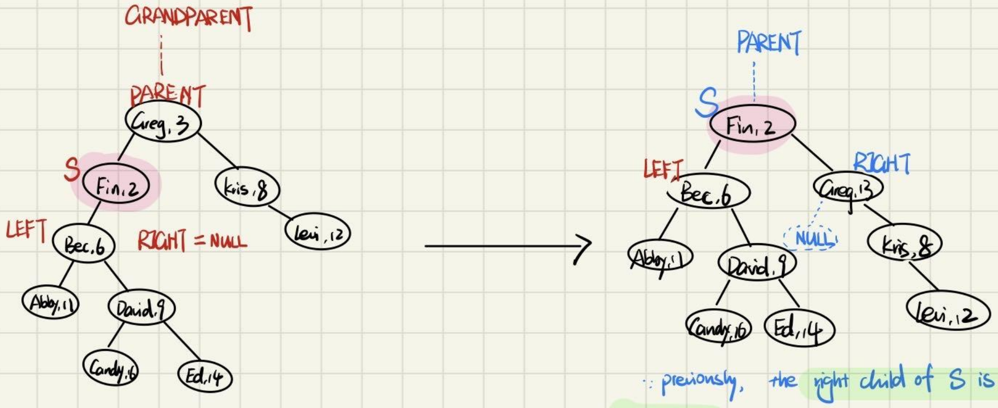
④ S. parent = GRANDPARENT
⑤ PARENT. parent = S
⑥ if RIGHT != NULL
RIGHT. parent = PARENT

② Left - Rotation

① S. left = PARENT
② PARENT. right = LEFT
③ same as above
④ same as above
⑤ same as above
⑥ if LEFT != NULL
LEFT. parent = PARENT

Right - Rotation

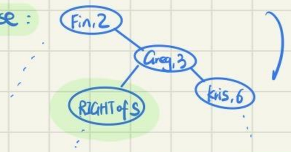
Example :
from Part B



∵ previously, the right child of S is NULL

∴ no update need for Greg, 3

(otherwise :



Part D

The worst case occurs when the QUBSET becomes **a "stick" or a "degenerate tree"**,

while it still follows the requirements of:

(i.) BST: left.name < parent.name < right.name

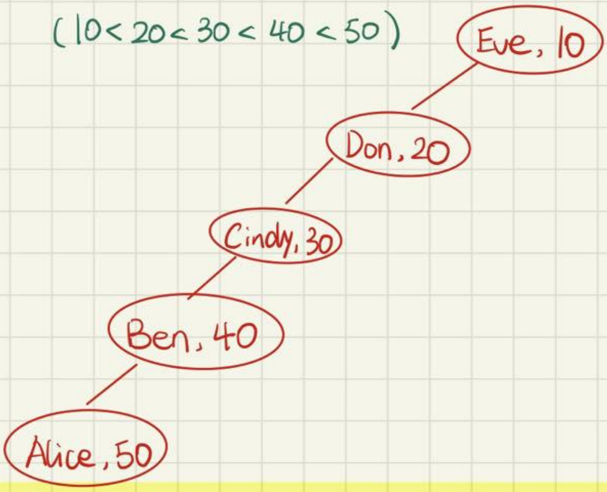
(ii.) Minheap: parent.id < right.id && parent.id < left.id

The height of the QUBSET is equal to the number of the nodes, which is 5 in this case.
The time complexity of the worst case of the QUBSET is O(n).

① Example of a degenerate tree that inclines to Left :

- The name of student is decreasing and the id number of the student is increasing. (E>D>C>B>A)

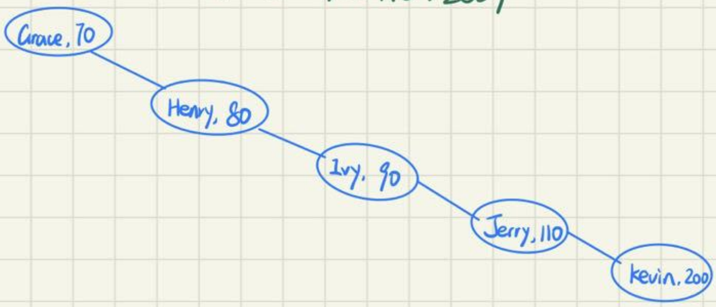
(10<20<30<40<50)



② Example of a degenerate tree that inclines to Right:

- The name of the student is increasing, and the id number of the student is increasing (G>H>I>J>K)

(70>80>90>110>200)



Part E

When the BST degenerates to a tree of height n, it will become a **stick**.
As the group of n student are listed in alphabetical order, those nodes will be added into the BST based on the order of name from small to large.

QUBSET needs to satisfy **two requirements**: one is the **BST condition**, which is fulfilled when all nodes are initially added into the BST and sorted by the name (For left-inclined stick: left child < parent; For right-inclined stick: right child > parent). Simultaneously, we also need to ensure that the **minheap condition is maintained.** The minheap rule suggests that every child node must have larger value of student ID than their parent node (children.id > parent.id).

Here, since the question implies that the group n students will have their **student IDs stored in random distribution,** there is possibility that some child nodes would have smaller value of ID than their parent node, which is unsatisfying for the QUBSET. Thus, we need to perform **rotation** to the QUBSET in order to achieve minheap rules, therefore, resulting in a height of the tree that much smaller than n.

Changes in height of the tree occurs when:

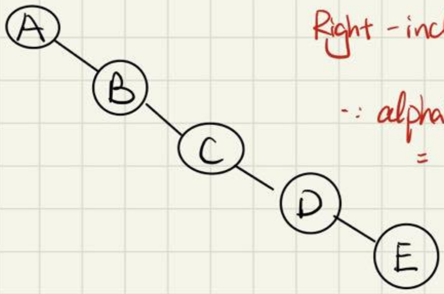
(1.) Every time when the minheap is not maintained → which means the id value of this child node is smaller than its parent. → Unsatisfying result → Rotation is needed to apply to the tree.
(2.) Then, as the node **moves to the upper level** of the tree after rotation, the height of the whole tree will **decrease by one.**
(3.) This strategy will be continually applied until we finally achieve a satisfying QUBSET.

Here is an example to prove the statement:

① Assume we have 5 students : A, B, C, D, E ;
where $n = 5$. and they are listed in alphabetic order.

② Initially, a group of 5 students will be added into BST.

Because they are only sorted by name, hence it will display a diagram of Stick



Right-inclinde stick

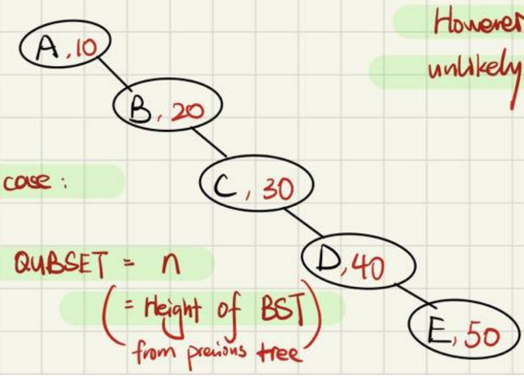
∵ alphabetic order
= name is increasing

③ Then, as every student has a name and a student ID.
We should consider a QUBSET instead of a BST now.

- We have 5 students : (A, ?), (B, ?), (C, ?), (D, ?), (E, ?)

The new tree will degenerate to a stick
ONLY if the student ID is also increasing:
<ascending order>

For example :



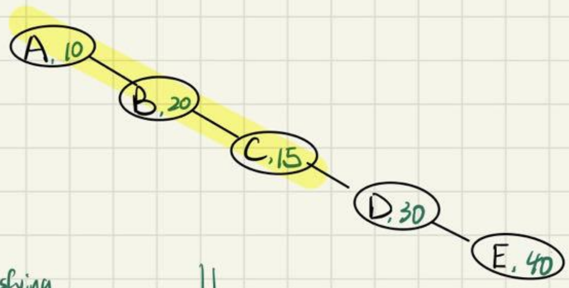
However, this case is unlikely to occur.

In this case :

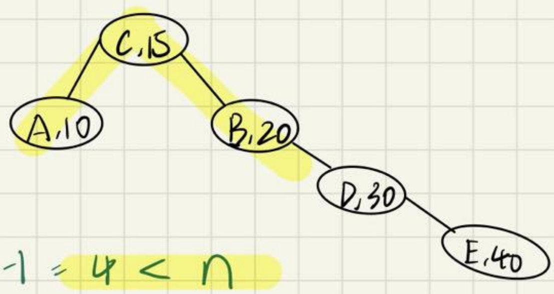
Height of QUBSET = n
(= Height of BST from previous tree)

④ Most likely, the ID number of the students will be randomly distributed in real practice.

For example :



unsatisfying, need AVL rotation



Height = 5 - 1 = 4 < n

⑤ Generally speaking, the height of QUBSET is likely to be smaller than n because we apply AVL rotation to achieve such QUBSET.

- The idea of AVL Rotation always ensures the tree remains balanced and minimise its height.

Question 2: Colourful Study Notes

Part C

As we are using brute force approach to generate all sequences and find the maximum score. Firstly, we need to define the variable maxScore and initialize it with 0 **(maxScore = 0)**. When every time there is a sequence generated during looping, we calculate the relative score and update it to the maximum score.

Therefore, the question will require us to find the **time complexity of:**
generating all possible sequences of **n words of c colours.**

$$\underbrace{\underline{W_1}\quad \underline{W_2}\quad \underline{W_3}\quad \underline{W_4}\quad \underline{W_5}\quad \ldots\quad \underline{W_{n-2}}\quad \underline{W_{n-1}}\quad \underline{W_n}}$$

- we have n spots to store the words,
  every spot from n will have c options of colour
- Total possible matches = C^n

↘ i.e

		Total matches
$n=1$	_	$c = c^1$
$n=2$	_ _	$c \times c = c^2$
$n=3$	_ _ _	$c \times c \times c = c^3$
⋮	⋮	⋮
$n=n$	_ _ … _ _	$\underbrace{c \times \cdots \times c}_{n} = c^n$

Now, we have the **total number of possible matches is C^n**, we then discuss:

(i.) **Best case: this case occurs when all terms in all sequences don't have termColourTable**
$C_{best} = C^n * C^2$

(ii.) **Worst case: $C_{worst} = C^n * n^2 * C^2$**

(iii.) **Average case: $C_{average} = C^n * n^2 * C^2$**

bestScore = 0
bestSequence = NULL

for every sequence do → C^n
    int score = 0 → Constant (1)
    for every term within each sequence do → n times

        get the term Colour Table from the problem → n times
        if the term Colour Table is exist for the term then
            calculate WC score → Constant (1)
            for every element in colour Transicion table → C^2 times
                if this element has preColour C_{i-1} && Current colour C_i then
                    calculate CT score → Constant (1)

    Constant (1) → Score = WC + CT <Sum 2 parts of scores to get the total>

Constant (1) → Compare the total score with bestScore then update bestScore and bestSequence

(i). Best Case : $C_{best} = C^n \cdot n^2$

(ii) Worst Case : $C_{worst} = C^n \cdot n^2 \cdot C^2$

(iii) Average Case : $C_{average} = C^n \cdot n^2 \cdot C^2$

Part D

From the background information of Tala's DP scenario, we notice:

(1.) The best colour of the first term is easy to get from the table.
(2.) Option 1 (c1 does include): total score = maxScore updated from previous colors + maxScore of current color.
Hence, we need to find the specific color that **max {WC(w,c) + CT(c_1,c)}** while looping through every colour.
(3.) Option 2 (c1 does not include): Not only we need to loop through every colour of the term, but also we need to loop through its previous term and check its all the other colour cases.
Here, as c1 denotes the colour of the previous word, similarly we define c2 as one case of all other colours of the previous word, where c2 is in the interval between 0 and total colours minus one, except c1.
Hence, we need to find **max {WC(w,c) + CT(c_2,c)}**, **c** with maximum score and **c2** with maximum score.

The recurrence relation for the score F would have feature as follows:

(1.) F (0, c) represents the score of colour c of the first term.

$$F(0, c) = WC(W_0, c)$$
$$F(0) = \max_c F(0, c)$$

(2.) F (n, c) represents the score of color c that has been checked until the n^{th} term.
(3.) F (i, c) represents the maximum score that has been updated util the n^{th} term is checked.
   (i.) **Option 1:** $F_1(i, c) = F(i-1, c_1) + WC(w_i, c)+CT(c_1, c)$, **where c_1 is the colour that has maximum score.**
   (ii.) **Option 2:** $F_2(i, c) = \max_{c2=0\cdots cn-1!=c1} F(i-1, c2) + WC(w_i, c)+CT(c_2, c)$, **where c_2 is the every specific colour case except c_1.**
(4.) $F(i) = \max_c F(i, c)$

The full recurrence relation would be:

$$F(i, c) = \max\{ F(i-1, c_1) + WC(w_i, c)+CT(c_1, c), \max_{c2=0\cdots cn-1!=c1} F(i-1, c2) + WC(w_i, c)+CT(c_2, c)\}$$

OR in another format:

Option 1: $F(n-1, C) + \text{newScoreUpdate}$

$$F_1(i, c) = F(i-1, C_1) + WC(w_i, c) + CT(C_1, c)$$

Option 2: $\max_{C_2 = 0 \dots C_{n-1} != C_1} F(n-1, C_2) + \text{newScoreUpdate}$

$$F_2(i, c) = \max_{C_2 = 0 \dots C_{n-1} != C_1} F(i-1, C_2) + WC(w_i, C) + CT(C_2, C)$$

In general:

$$F(i, c) = \max\left(F(i-1, C_1) + WC(w_i, c) + CT(C_1, c) \;,\; \max_{C_2 = 0 \dots C_{n-1} != C_1} F(i-1, C_2) + WC(w_i, c) + CT(C_2, c) \right)$$

Part G

(i.) Best case: this case occurs when all terms don't have termColourTable
$C_{best} = n^2$

(ii.) Worst case: $C_{worst} = n^2 + n * C + n * C^4$

(iii.) Average case: $C_{average} = n^2 + n * C + n * C^4$

```
int maxScore = 0
int max Colour = 0
int prev Colour = 0
for every term i do                                   → n times
  find the termColourTable                            → max n times
  if termColourTable for term i does not exist then
    for every colour j of the term i-1 do             → c times
      calculate the best total score                  → Constant (1)
      Store the best score and its colour in respective DP array → Constant (1)
  elseif termColourTable for term i exists then
    for every colour j in the termColour Table do     → c times

      for every element e in colour Transition Table do   → c^2 times
        if this element has prevColour prev colour && current
        colour j then
          calculate score based on Option 1           → Constant (1)

c times →    for every other cases of colour j2 in the termColour Table do
c^2 times →    for every element e in colourTransition Table do
                 if this element has prevColour j2 && current
                 colour j then
Constant (1) →     calculate score based on Option 2.

             → Store the score into DP [i][j]
Constant (1) → Store the prev Colour of highest score into bestPrevColours[i][j]

             → Compare then update the maxScore and maxColour
  else
Constant (1) → update the prev Colour and bestprev Colours array and
               DP array specificly
```

ci.) Best Case : $C_{best} = n^2$

cii.) Worst Case : $C_{worst} = n(n + c + c \cdot c^2 \cdot c \cdot c^2) = n^2 + n \cdot c + n \cdot c^4$

ciii.) Average Case : $C_{average} = n(n + c + c \cdot c^2 \cdot c \cdot c^2) = n^2 + n \cdot c + n \cdot c^4$