$Assignment\ 2$

Haohu Shen UCID: 30063099

CPSC 331 - Data Structures, Algorithms, and Their Analysis

June 5, 2019

Question 1

Claim If a nonempty 2-3 tree T represents a subset $S \subseteq E$ such that $|S| = n \in \mathbb{N}$, then T has at most $n - 1$ internal nodes.

Proof Suppose a nonempty 2-3 tree T represents a subset $S \subseteq E$ such that $|S| = n \in \mathbb{N}$ and the depth of T is $d \in \mathbb{N}$, we try to use the ***Direct Proof*** on d to prove the claim holds for every possible value of d.

- Suppose $d = 0$, the depth of the subtree with root x is zero, there exists only a leaf which is also the root of T, thus $n = 1$ and the number of internal nodes is $0 \leq 0 = 1 - 1 = n - 1$, thus the claim holds in this case.

- Suppose $d \geq 1$, suppose x_i is the number of internal nodes that each has exactly 3 children and y_i is the number of internal nodes that each has exactly 2 children such that all $x_i + y_i$ internal nodes just mentioned are on the same level of the tree, thus i is the number of edges from all $x_i + y_i$ internal nodes just mentioned to the root of T such that $0 \leq i \leq d - 1$. Thus,

  - When $i = 0$, since only a leaf which is also the root of T exists,

    $$x_0 + y_0 = 1$$

  - When $1 \leq i \leq d - 1$, since $i - 1 \geq 0$, by the property **(b)** of a 2-3 tree, we have

    $$x_i + y_i = 3x_{i-1} + 2y_{i-1}$$

  - When $i = d - 1$, since all internal nodes such that the number of edges from each internal node itself to the root of T is $d - 1$, all children of these internal nodes are leaves of the tree, thus we have

    $$n = 3x_{d-1} + 2y_{d-1}$$

  Therefore, we combine all three cases above by the notation of summation, adding the left-hand side of all equations and adding the right-hand side of all equations separately, that is

$$(x_0 + y_0) + \sum_{i=1}^{d-1}(x_i + y_i) + n = 1 + \sum_{i=0}^{d-2}(3x_i + 2y_i) + 3x_{d-1} + 2y_{d-1}$$

Thus

$$\sum_{i=0}^{d-1}(x_i + y_i) + n = 1 + \sum_{i=0}^{d-1}(3x_i + 2y_i)$$

$$n - 1 = \sum_{i=0}^{d-1}(3x_i + 2y_i) - \sum_{i=0}^{d-1}(x_i + y_i)$$

$$n - 1 = \sum_{i=0}^{d-1}((3x_i + 2y_i) - (x_i + y_i))$$

$$n - 1 = \sum_{i=0}^{d-1}(2x_i + y_i)$$

$$n - 1 = \sum_{i=0}^{d-1}(2x_i) + \sum_{i=0}^{d-1} y_i$$

$$n - 1 = 2\sum_{i=0}^{d-1} x_i + \sum_{i=0}^{d-1} y_i$$

Since the number of all internal nodes in T is

$$\sum_{i=0}^{d-1}(x_i + y_i) = \sum_{i=0}^{d-1} x_i + \sum_{i=0}^{d-1} y_i$$

We can suppose that

$$a = \sum_{i=0}^{d-1} x_i$$

and

$$b = \sum_{i=0}^{d-1} y_i$$

Thus

$$n-1=2\sum_{i=0}^{d-1}x_i+\sum_{i=0}^{d-1}y_i=2a+b$$

$$a+b+a=n-1$$

$$a+b=n-1-a$$

Since n is a fixed value and a is not fixed such that $a \geq 0$

$$a+b=n-1-a\leq n-1$$

Thus, the number of internal nodes in T is at most

$$\max(a+b)=n-1$$

In this case $a=0$ and $b=n-1$, which means no internal nodes such that for each internal node that has exactly 3 children, that is, every internal node of T has exactly 2 children. Furthermore, in ***Question 3*** we have proved the claim that, if T is a 2-3 tree with depth $d \in \mathbb{N}$ and each internal node has exactly 2 children, then the size of the subset of E represented by T is 2^d. Thus, in this case n must satisfy $n=2^d$ where d is the depth of T.

Therefore, in all cases for all possible values of the depth of T are considered and the claim holds. Thus, by ***Direct Proof***, we can conclude that, if a nonempty 2-3 tree T represents a subset $S \subseteq E$ such that $|S|=n \in \mathbb{N}$, then T has at most $n-1$ internal nodes.

Question 2

CITATION

Please note before grading, the format of the proof acts in a similar manner to the proof provided from page 48 to page 49 in L01_intro_and_math_review.pdf[2] in order to providing a completed, clear and professional proof.

Claim If T is a non-empty 2-3 tree with depth $d \in \mathbb{N}$ and n is the size of the subset of E represented by T in this case, then $2^d \leq n \leq 3^d$.

Proof We prove the claim by the strong form of mathematical induction on d. Cases that $d=0$ and $d=1$ will be used in the basis.

Basis (d=0) When $d=0$, that is, the depth of the tree is zero, there exists only a leaf which is also the root of T by the definition, thus $n=1$ is the size of the subset of E represented by T in this case. Since $2^0 \leq 1 \leq 3^0$, we have $2^d \leq n \leq 3^d$, as required.

Basis (d=1) When $d=1$, that is, the depth of the tree is one, thus the root of T is an internal node by the

definition and it has exactly either two or three children.

- If the root of T has exactly two children, then all of its children are also leaves of tree since the depth is 1, thus $n = 2$ is the size of the subset of E represented by T in this case. Since $2^1 \leq 2 \leq 3^1$, we have $2^d \leq n \leq 3^d$ when the root of T has exactly two children.

- If the root of T has exactly three children, then all of its children are also leaves of tree since the depth is 1, thus $n = 3$ is the size of the subset of E represented by T in this case. Since $2^1 \leq 3 \leq 3^1$, we have $2^d \leq n \leq 3^d$ when the root of T has exactly three children.

- Since the claim holds in both cases, we can conclude that $2^d \leq n \leq 3^d$ when $d = 1$ and n is the size of the subset of E represented by T.

Inductive Step: Let $k \geq 1$ to be an integer. It is necessary and sufficient to use

Inductive Hypothesis: Suppose for all integers m such that $0 \leq m \leq k$, if T is a non-empty 2-3 tree with depth m and n_m is the size of the subset of E represented by T in this case, then $2^m \leq n_m \leq 3^m$.

to prove

Inductive Claim: If T is a non-empty 2-3 tree with depth $k+1$ and n_{k+1} is the size of the subset of E represented by T in this case, then $2^{k+1} \leq n_{k+1} \leq 3^{k+1}$.

The depth of internal nodes of T whose children are all leaves is k since when the depth is increased by 1 for a 2-3 tree whose depth is k all its leaves will be converted to internal nodes and the children of these new internal nodes are all leaves again. Suppose the number of internal nodes of T whose children are all leaves is n_k, since $k \geq 1$, $k+1 \geq 2$, thus the **inductive hypothesis** is applied and we have $2^k \leq n_k \leq 3^k$. Since for each internal node whose depth is k, it has exactly either two or three children.

Suppose in these n_k internal nodes, there are only p internal nodes that each has exactly two children such that $0 \leq p \leq n_k$. Thus there are $n_k - p$ nodes that each has exactly three children. Thus

$$n_{k+1} = 2p + 3(n_k - p)$$

Since $2^k \leq n_k \leq 3^k$ and $0 \leq p \leq n_k$, the minimum of n_{k+1} is

$$\begin{aligned}
\min(n_{k+1}) &= \min(2p + 3(n_k - p)) \\
&= \min(3n_k - p) \\
&= \min(3n_k - n_k) \\
&= \min(2n_k) \\
&= 2 \cdot 2^k \\
&= 2^{k+1}
\end{aligned}$$

and the maximum of n_{k+1} is

$$\begin{aligned}\max(n_{k+1}) &= \max(2p + 3(n_k - p)) \\ &= \max(3n_k - p) \\ &= \max(3n_k) \\ &= 3 \cdot 3^k \\ &= 3^{k+1}\end{aligned}$$

Thus $2^{k+1} \leq n_{k+1} \leq 3^{k+1}$, as required.

Conclusion: Therefore, by strong form of mathematical induction, we can conclude that, if T is a non-empty 2-3 tree with depth $d \in \mathbb{N}$ and n is the size of the subset of E represented by T in this case, then $2^d \leq n \leq 3^d$.

Question 3

CITATION

Please note before grading, the format of the proof acts in a similar manner to the proof provided from page 48 to page 49 in L01_intro_and_math_review.pdf[2] in order to providing a completed, clear and professional proof.

Solution Let $d \in \mathbb{N}$ and suppose that $|E| \geq 2^d$, in order to prove that there exists a non-empty 2-3 tree with depth d that represents a subset of E with size exactly 2^d, we show that if T is a 2-3 tree with depth $d \in \mathbb{N}$ and each internal node has exactly 2 children, then the size of the subset of E represented by T is 2^d.

Claim If T is a 2-3 tree with depth $d \in \mathbb{N}$ and each internal node has exactly 2 children, then the size of the subset of E represented by T is 2^d.

Proof We prove the claim by the strong form of mathematical induction on d. Cases that $d = 0$ and $d = 1$ will be used in the basis.

Basis (d=0) When $d = 0$, that is, the depth of the tree is zero, there exists only a leaf which is also the root of T and no internal nodes exist by the definition, thus $n = 1$ is the size of the subset of E represented by T in this case. Since $n = 1 = 2^0 = 2^d$, the claim holds in this case.

Basis (d=1) When $d = 1$, that is, the depth of the tree is one, thus the root of T is an internal node by the definition. Since it has exactly two children, $n = 2$ is the size of the subset of E represented by T in this case. Since Since $n = 2 = 2^1 = 2^d$, the claim holds in this case.

Inductive Step: Let $k \geq 1$ be an integer. It is necessary and sufficient to use

Inductive Hypothesis: Suppose for all integers m such that $0 \leq m \leq k$, if T is a 2-3 tree with depth $m \in \mathbb{N}$

and each internal node has exactly 2 children, then the size of the subset of E represented by T is 2^m.

to prove

Inductive Claim: If T is a 2-3 tree with depth $k+1 \in \mathbb{N}$ and each internal node has exactly 2 children, then the size of the subset of E represented by T is 2^{k+1}.

Since $k \geq 1, k+1 \geq 2$. Suppose the number of internal nodes of T whose children are all leaves is n, by the **inductive hypothesis**, we have $n = 2^k$ since when the depth is increased by 1 for a 2-3 tree with depth k all its leaves will be converted to internal nodes whose children are all leaves. Since for each internal node whose children are all leaves, it has exactly 2 children, the size of the subset of E represented by T is

$$2n = 2 \cdot 2^k = 2 \cdot 2^{k+1}$$

as required to establish the claim.

Conclusion: Therefore, by strong form of mathematical induction, we can conclude that, if T is a 2-3 tree with depth $d \in \mathbb{N}$ and each internal node has exactly 2 children, then the size of the subset of E represented by T is 2^d.

Thus, there exists a non-empty 2-3 tree with depth d that represents a subset of E with size exactly 2^d if $d \in \mathbb{N}$ and $|E| \geq 2^d$.

Question 4

CITATION

Please note before grading, the format of the proof acts in a similar manner to the proof provided from page 25 to page 26 in L02_correctness.pdf[1] in order to providing a completed, clear and professional proof.

Claim The depth of the subtree with root x is a ***bound function*** for the recursive algorithm in Figure 3.

Proof We prove the claim by showing that the depth of the subtree with root x satisfied all 3 properties included in the definition of a bound function for a recursive algorithm.

- From part **(a)** in the precondition of the computational problem whose algorithm is in Figure 3, the 2-3 tree T satisfies the 2-3 Tree Properties, thus the depth of T is defined to be the number of edges in a longest path from the root to a leaf in T, thus the depth is an integer, that is, the depth of the subtree with root x is an integer-valued function of the depth of the subtree with root x.
- We can see from the line 7, line 8, line 10, line 12 and line 13 of the algorithm described in Figure 3, when the algorithm is recursively call itself, the parameter of node x will be changed into one of its children, since the depth of the subtree with the root being one of its children is less than the depth of the subtree

with the root x by 1, thus the value of the depth is reduced by at least 1.

- If the function's value is less than or equal to zero when the algorithm is applied, that is, the depth of the subtree with root x is less than or equal to zero. Since x is not-null from part **(c)** in the precondition, the depth is greater or equal to zero. Thus, the depth must be zero, and the root x is a leaf in this case. Thus the execution of the algorithm passed the test at line 1, if the key is equal to the element stored at x, then the execution will pass the test at line 2 and return x at line 3, otherwise the test at line 2 failed and the execution will throw a $NoSuchElementException$ at line 4, we can see in both cases the algorithm does not call itself recursively during the execution.

Since the depth of the subtree with root x satisfies all the properties included in the definition of a ***bound function*** for a recursive algorithm, it is a bound function of the recursive algorithm in Figure 3.

Question 5

CITATION
Please note before grading, the format of the proof acts in a similar manner to the proof provided from page 42 to page 49 in L09_BST_1.pdf[4] in order to providing a completed, clear and professional proof.

Solution In order to prove the algorithm *get* in Figure 3 correctly solves the **Searching in a Subtree of This 2-3 Tree** problem, we prove that if the precondition of the problem is satisfied and the algorithm *get* is called with a non-null key of type E and a non-null node x as input, then the algorithm *get* eventually ends and the postcondition of the problem is satisfied without undocumented side effect.

Claim Suppose the 2-3 tree T satisfies the 2-3 Tree Properties, if the algorithm *get* is called with a non-null key of type E and a non-null node x in T, then the execution eventually ends. If the key is stored in the subtree of T with root x, then the leaf in this subtree storing x is returned as output. A $NoSuchElementException$ is thrown otherwise. And the 2-3 tree has not changed, so it still satisfies the 2-3 Tree Properties.

Proof We prove the claim by the strong form of mathematical induction on d, which is the depth of the subtree with root x. The case that $d = 0$ will be used in the basis.

Basis (d=0) When $d = 0$, that is, the depth of the tree is zero, there exists only a leaf which is also the root of T and no internal nodes exist by the definition, thus x is a leaf in this case, so the execution of the algorithm passed the test at line 1 and continues at line 2.

- If the key is equal to the element stored at x, the test at line 2 is checked and passed and the execution continues at line 3, which caused the execution of the algorithm eventually ends and the node x is returned as output without undocumented side effect, as required.

- If the key is not equal to the element stored at x, the test at line 2 is checked and failed and the execution

continues at line 4, which caused the execution of the algorithm eventually ends and a $NoSuchElementException$ is thrown, as required.

Inductive Step: Let $k \geq 0$ be an integer. It is necessary and sufficient to use

Inductive Hypothesis: Suppose the 2-3 tree T satisfies the 2-3 Tree Properties, if the algorithm *get* is called with a non-null key of type E and a non-null node y in T such that the depth of the subtree with root y has depth at most k, then the execution eventually ends. If the key is stored in the subtree of T with root y, then the leaf in this subtree storing y is returned as output. A $NoSuchElementException$ is thrown otherwise. And the 2-3 tree has not changed, so it still satisfies the 2-3 Tree Properties.

to prove

Inductive Claim: Suppose the 2-3 tree T satisfies the 2-3 Tree Properties, if the algorithm *get* is called with a non-null key of type E and a non-null node z in T such that the depth of the subtree with root z has depth $k+1$, then the execution eventually ends. If the key is stored in the subtree of T with root z, then the leaf in this subtree storing z is returned as output. A $NoSuchElementException$ is thrown otherwise. And the 2-3 tree has not changed, so it still satisfies the 2-3 Tree Properties.

Suppose the 2-3 tree T satisfies the 2-3 Tree Properties and the algorithm *get* is called with a non-null key of type E and a non-null node z in T such that the depth of the subtree with root z has depth $k+1$. Since $k \geq 0$, $k+1 \geq 1$, thus the node z is an internal node.

Since node z is not a leaf, the test at line 1 is checked and failed and the execution of the algorithm continues at line 5.

Case 1 (z has exactly 2 children):

Since z has exactly 2 children, the test at line 5 is checked and passed and the execution of the algorithm continues at line 6.

- If the value of key is less than or equal to the largest value stored at any leaf of the first subtree with root z, then a leaf storing the value of key must be in the first subtree of the 2-3 tree with root z if the value of key is stored in the subtree of T by the definition **(c)** of a 2-3 tree. In this case, the test at line 6 is checked and passed and the execution of the algorithm continues at line 7.

  Suppose the first child of z is p. Since the depth of the subtree with root p is at most k. By the inductive hypothesis, the algorithm *get* is called with key of type E and the node p and the execution of the algorithm eventually ends.

  - If there is a leaf in the subtree with p storing the value of key, then the leaf is returned as output.
  - If there is no leaf in the subtree with p storing the value of key, a $NoSuchElementException$ is

thrown.

Since T is not changed, by the definition **(h)** of a 2-3 tree, the subtree with root p is still a 2-3 tree and it still satisfies the 2-3 Tree Properties.

Thus the recursive call *get(key, x.firstChild)* at line 7 eventually and the inductive claim holds in this case.

- If the value of key is greater than the largest value stored at any leaf of the first subtree with root z, then a leaf storing the value of key must be in the second subtree of the 2-3 tree with root z if the value of key is stored in the subtree of T by the definition **(c)** of a 2-3 tree. In this case, the test at line 6 is checked and failed and the execution of the algorithm continues at line 8.

  Suppose the second child of z is p. Since the depth of the subtree with root p is at most k. By the inductive hypothesis, the algorithm *get* is called with key of type E and the node p and the execution of the algorithm eventually ends.

  - If there is a leaf in the subtree with p storing the value of key, then the leaf is returned as output.
  - If there is no leaf in the subtree with p storing the value of key, a $NoSuchElementException$ is thrown.

  Since T is not changed, by the definition **(h)** of a 2-3 tree, the subtree with root p is still a 2-3 tree and it still satisfies the 2-3 Tree Properties.

  Thus the recursive call *get(key, x.secondChild)* at line 8 eventually and the inductive claim holds in this case without undocumented side effect.

Case 2 (z has exactly 3 children):

Since z has exactly 3 children, the test at line 5 is checked and failed and the execution of the algorithm continues at line 9.

- If the value of key is less than or equal to the largest value stored at any leaf of the first subtree with root z, then a leaf storing the value of key must be in the first subtree of the 2-3 tree with root z if the value of key is stored in the subtree of T by the definition **(c)** of a 2-3 tree. In this case, the test at line 9 is checked and passed and the execution of the algorithm continues at line 10.

  Suppose the first child of z is p. Since the depth of the subtree with root p is at most k. By the inductive hypothesis, the algorithm *get* is called with key of type E and the node p and the execution of the algorithm eventually ends.

  - If there is a leaf in the subtree with p storing the value of key, then the leaf is returned as output.

- If there is no leaf in the subtree with p storing the value of key, a $NoSuchElementException$ is thrown.

Since T is not changed, by the definition **(h)** of a 2-3 tree, the subtree with root p is still a 2-3 tree and it still satisfies the 2-3 Tree Properties.

Thus the recursive call *get(key, x.firstChild)* at line 10 eventually and the inductive claim holds in this case.

- If the value of key is greater than the largest value stored at any leaf of the first subtree with root z but and is less than or equal to the largest value stored at any leaf of the second subtree with root z, then a leaf storing the value of key must be in the second subtree of the 2-3 tree with root z if the value of key is stored in the subtree of T by the definition **(c)** of a 2-3 tree. In this case, the test at line 9 is checked and failed, and the test at line 11 is checked and passed, thus the execution of the algorithm continues at line 12.

  Suppose the second child of z is p. Since the depth of the subtree with root p is at most k. By the inductive hypothesis, the algorithm *get* is called with key of type E and the node p and the execution of the algorithm eventually ends.

  - If there is a leaf in the subtree with p storing the value of key, then the leaf is returned as output.
  - If there is no leaf in the subtree with p storing the value of key, a $NoSuchElementException$ is thrown.

  Since T is not changed, by the definition **(h)** of a 2-3 tree, the subtree with root p is still a 2-3 tree and it still satisfies the 2-3 Tree Properties.

  Thus the recursive call *get(key, x.secondChild)* at line 12 eventually and the inductive claim holds in this case.

- If the value of key is greater than the largest value stored at any leaf of the second subtree with root z, then a leaf storing the value of key must be in the third subtree of the 2-3 tree with root z if the value of key is stored in the subtree of T by the definition **(c)** of a 2-3 tree. In this case, the test at line 9 and 11 both failed and the execution of the algorithm continues at line 13.

  Suppose the second child of z is p. Since the depth of the subtree with root p is at most k. By the inductive hypothesis, the algorithm *get* is called with key of type E and the node p and the execution of the algorithm eventually ends.

  - If there is a leaf in the subtree with p storing the value of key, then the leaf is returned as output.
  - If there is no leaf in the subtree with p storing the value of key, a $NoSuchElementException$ is thrown.

Since T is not changed, by the definition **(h)** of a 2-3 tree, the subtree with root p is still a 2-3 tree and it still satisfies the 2-3 Tree Properties.

Thus the recursive call *get(key, x.thirdChild)* at line 13 eventually and the inductive claim holds in this case without undocumented side effect.

Since in both cases inductive claim holds, we can conclude that the inductive claim holds when the depth of the subtree with root x is greater than or equal to 1.

Conclusion: Therefore, by strong form of mathematical induction, we can conclude that the claim is established. Thus if the precondition of the problem is satisfied and the algorithm *get* is called with a non-null key of type E and a non-null node x as input, the algorithm *get* eventually ends and the postcondition of the problem is satisfied, that is, the algorithm *get* in Figure 3 correctly solves the "Searching in a Subtree of This 2-3 Tree" problem.

Question 6

CITATION
Please note before grading, the format of the solution acts in a similar manner to the solution provided in page 26 and page 27 in L05_efficiency.pdf[3] in order to providing a completed, clear and professional solution.

Solution In order to give a recurrence for $T_{get}(k)$, we suppose the uniform cost criterion is used to define and the precondition of the "Searching in a Subtree of This 2-3 Tree" problem is satisfied, the recursive algorithm is executed with a non-null key of type E and the subtree with the node x as root whose the depth is k such that $k \in \mathbb{N}$, thus k is an integer such that $k \geq 0$.

- If $k = 0$, the depth of the subtree with root x is zero, there exists only a leaf which is also the root of T and no internal nodes exist by the definition, thus x is a leaf in this case, so the execution of the algorithm passed the test at line 1 and continues at line 2.
    - If the key is equal to the element stored at x, the test at line 2 is checked and passed and the execution continues at line 3, which caused the execution of the algorithm eventually ends and the node x is returned as output. Since there are totally 3 steps executed in this case,

$$T_{get}(k) = T_{get}(0) \leq 3$$

    - If the key is not equal to the element stored at x, the test at line 2 is checked and failed and the execution continues at line 4, which caused the execution of the algorithm eventually ends and a $NoSuchElementException$ is thrown. Since there are totally 3 steps executed in this case,

$$T_{get}(k) = T_{get}(0) \leq 3$$

Since in both cases the number of steps taken by the algorithm is the same, we can conclude that if $k = 0$,

$$T_{get}(k) = T_{get}(0) \leq 3$$

- If $k \geq 1$, the depth of the subtree with root x is greater than or equal to one, thus x is an internal node. Therefore, x has exactly either 2 or 3 children.
    - If x has exactly 2 children, during the execution of the algorithm *get*, the test at line 1 is failed and the test at line 5 is checked and passed, thus execution continued with line 6.
        - If the value of key is less than or equal to the largest value stored at any leaf of the first subtree with root x, then the test at line 6 is checked and passed, and the execution continued with the step at line 7, which totally took 4 steps, and then called itself recursively with the input key of type E and the first child of x as the new node x. Since the depth of the subtree whose root is the first child of x is $k - 1$,

$$T_{get}(k) = 4 + T_{get}(k - 1)$$

        - If the value of key is greater than the largest value stored at any leaf of the first subtree with root x, then the test at line 6 is checked and failed, and the execution continued with the step at line 8, which totally took 4 steps, and then called itself recursively with the input key of type E and the second child of x as the new node x. Since the depth of the subtree whose root is the second child of x is $k - 1$,

$$T_{get}(k) = 4 + T_{get}(k - 1)$$

    - If x has exactly 3 children, during the execution of the algorithm *get*, the tests at line 1 and 5 both failed and the execution continued with line 9.
        - If the value of key is less than or equal to the largest value stored at any leaf of the first subtree with root x, then the test at line 9 is checked and passed, and the execution continued with the step at line 10, which totally took 4 steps, and then called itself recursively with the input key of type E and the first child of x as the new node x. Since the depth of the subtree whose root is the first child of x is $k - 1$,

$$T_{get}(k) = 4 + T_{get}(k - 1)$$

        - If the value of key is greater than the largest value stored at any leaf of the first subtree with root x but less than or equal to the largest value stored at any leaf of the second subtree with root x, then the test at line 9 is checked and failed, and the test at line 11 is checked and passed, thus the execution continued with the step at line 12, which totally took 5 steps, and then called itself

recursively with the input key of type E and the second child of x as the new node x. Since the depth of the subtree whose root is the second child of x is $k-1$,

$$T_{get}(k) = 5 + T_{get}(k-1)$$

- If the value of key is greater than the largest value stored at any leaf of the second subtree with root x, then the test at line 9 and line 11 are both failed, thus the execution continued with the step at line 13, which totally took 5 steps, and then called itself recursively with the input key of type E and the third child of x as the new node x. Since the depth of the subtree whose root is the third child of x is $k-1$,

$$T_{get}(k) = 5 + T_{get}(k-1)$$

Since in all cases for $T_{get}(k)$ when $k \geq 1$, we have $T_{get}(k) = 4 + T_{get}(k-1)$ or $T_{get}(k) = 5 + T_{get}(k-1)$. Thus, when $k \geq 1$ and k is the depth of the subtree with root x, the number of steps taken by the algorithm is

$$T_{get}(k) \leq max(4 + T_{get}(k-1), 5 + T_{get}(k-1)) = 5 + T_{get}(k-1)$$

To conclude, when the uniform cost criterion is used to define and the precondition of the "Searching in a Subtree of This 2-3 Tree" problem is satisfied, the recursive algorithm is executed with a non-null key of type E and the subtree with the node x as root whose the depth is k such that $k \in \mathbb{N}$, if $T_{get}(k)$ is an upper bound for the number of steps used by the algorithm, then the ***recurrence*** for $T_{get}(k)$ can be expressed as

$$T_{get}(k) \leq \begin{cases} 3 & \text{if } k = 0 \\ 5 + T_{get}(k-1) & \text{if } k \geq 1 \end{cases}$$

Question 7

CITATION

Please note before grading, the format of the proof acts in a similar manner to the proof provided from page 48 to page 49 in L01_intro_and_math_review.pdf[2] in order to providing a completed, clear and professional proof.

Claim If $T_{get} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Question 6*** then when $k \in \mathbb{N}$,

$$T_{get}(k) \leq 5k + 3$$

Proof We prove the claim by the strong form of mathematical induction on k. The case that $k = 0$ will be used in the basis.

Basis (k=0) When $k = 0$, by the definition of the recurrence, we have

$$T_{get}(k) = T_{get}(0) \leq 3 = 5 \cdot 0 + 3 = 5k + 3$$

as required in the case.

Inductive Step: Let $k \geq 0$ be an integer. It is necessary and sufficient to use

Inductive Hypothesis: Suppose that m is a non-negative integer. Suppose for all integers m such that $0 \leq m \leq k$,

If $T_{get} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Question 6*** then when $m \in \mathbb{N}$,

$$T_{get}(m) \leq 5m + 3$$

to prove

Inductive Claim: If $T_{get} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Question 6*** then when $k + 1 \in \mathbb{N}$,

$$T_{get}(k + 1) \leq 5(k + 1) + 3$$

Since $k \geq 0$, $k + 1 \geq 1$, thus by the definition of the recurrence, we have

$$T_{get}(k + 1) \leq 5 + T_{get}(k)$$

Thus by the inductive hypothesis which is applied since $0 \leq k \leq k$,

$$T_{get}(k + 1) \leq 5 + (5k + 3) = 5 + 5k + 3 = 5(k + 1) + 3$$

as required.

Conclusion: Therefore, by strong form of mathematical induction, we can conclude that, if $T_{get} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Question 6*** then when $k \in \mathbb{N}$,

$$T_{get}(k) \leq 5k + 3$$

Question 8

(a) Considered a not-null key of type E is inserted into an empty 2-3 tree T, since its root is a $null$ node before insertion, a new node without any children will be created as the root of T, since it has no children, it is a leaf by the definition of a 2-3 tree and it stores the value of key. Since the input key is added to the subset of E and the new T still satisfies the 2-3 Tree properties, the postcondition of the problem is satisfied.

(b) Considered a not-null key of type E is inserted into a 2-3 tree T whose subset of E being represented has the size of 1, that is, its root is a leaf before insertion.

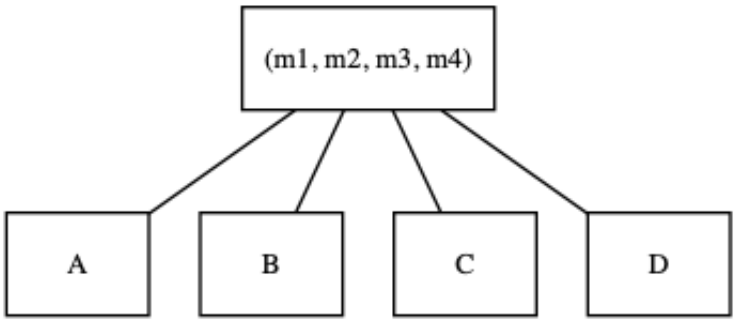
- If the value of key is the same as the value stored in the leaf, then the tree T will remain unchanged and an $ElementFoundException$ is thrown. Thus, the postcondition of the problem is satisfied in this case.

- If the value of key is not the same as the value stored in the leaf, suppose m_1 is the minimum between the value of key and the value stored in the leaf, m_2 is the maximum between the value of key and the value stored in the leaf. After the insertion, the root will be changed to an internal node that has exactly two children, the new root stores a 2-tuple (m_1, m_2), its left child is a leaf storing m_1 and its right child is a leaf storing m_2.

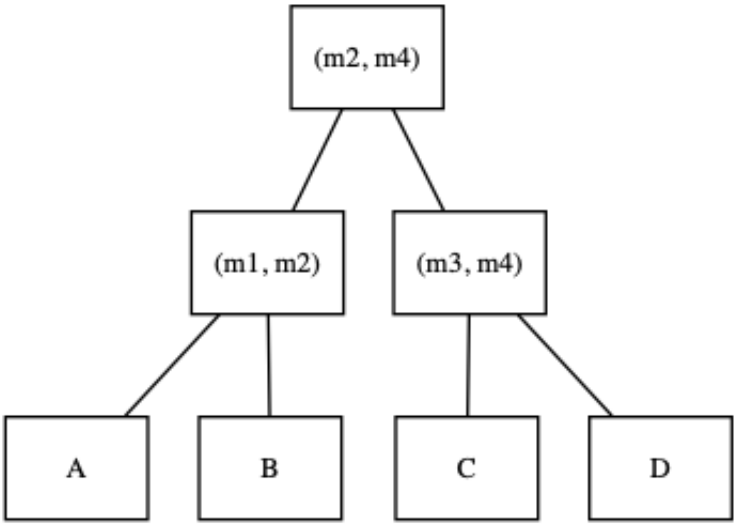
  By checking every item in the definitions of a 2-3 tree, we can confirm that T after insertion is still a 2-3 tree and thus still satisfies the 2-3 Tree properties, since the new subset of E represented by T is $\{m_1, m_2\}$ and the input key has been inserted into it, the postcondition of the problem is satisfied in this case.

Question 9

Solution Since the root of T has exactly 4 children, by ***Modified Tree Properties***, we can suppose the depth of T is d and that the root of T stores a 4-tuple (m_1, m_2, m_3, m_4) where m_1 is the largest value stored at any leaf of the first subtree, m_2 is the largest value stored at any leaf of the second subtree, m_3 is the largest value stored at any leaf of the third subtree, m_4 is the largest value stored at any leaf of the fourth subtree. We also suppose that A is the root of the first subtree of the root of T, B is the root of the second subtree of the root of T, C is the root of the third subtree of the root of T, D is the root of the fourth subtree of the root of T. And here is the layout of the root of T and A, B, C, D:



We modify T by splitting the root of T into three internal nodes, one is the root of T that stores a 2-tuple (m_2, m_4), one is the new root's first child (which is the new parent of A and B) that stores a 2-tuple (m_1, m_2), one is the new root's second child (which is the new parent of C and D) that stores a 2-tuple (m_3, m_4). And here is the layout of the new root of T and new position of A, B, C, D:



Therefore, the depth of the new root of T is $d + 1$, the depth of its 2 children are both d, and the depth of A, B, C, D is still $d - 1$. Thus the new T satisfies the property **(g)** of a 2-3 tree, again. Since the Modified Tree has ***at most*** 1 internal node that has either exactly 1 or 4 children, as well as the new root of T and its children all satisfy all properties of a 2-3 tree and the positions or values of other nodes are not changed, we can conclude that the new T is a 2-3 tree again without changing the subset of E represented by old T. And only a constant number of steps is needed to take since the adjustment only happens on the root of old T.

Question 10

Solution Since the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and a node x that is a leaf in T, we prove that the method would eventually terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied if the condition given is considered as an antecedent of the claim.

Claim If the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and a node x that is a leaf in T, the method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied.

Proof Since the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and a node x that is a leaf in T, by tracing through the execution of the pseudocode shown in Figure 5, we can see that the test at line 1 is checked and passed, and the value of type E storing at x is assigned to a variable e at step of line 2.

- If the value of e is equal to the value of key, then the test at line 3 is checked and passed, thus the execution of the algorithm continues at line 4, where an $ElementFoundException$ is thrown and the execution terminates after it.

  Since the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the

postcondition of the problem is false, the statement **(a)** is satisfied. Since the antecedents for both **(b)** and **(c)** of the postcondition of the problem are false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of e is not equal to the value of key, then the test at line 3 is checked and failed, thus the execution of the algorithm continues at line 5, where an $NoSuchElementException$ is thrown and the execution terminates after it.

  Since the antecedent of **(b)** of the postcondition of the problem is true, and the consequent of **(b)** of the postcondition of the problem is false, the statement **(b)** is satisfied. Since the antecedents for both **(a)** and **(c)** of the postcondition of the problem are false, the statements **(a)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

Therefore, in both cases, we showed that the execution of the algorithm eventually ends with the postcondition for the problem satisfied, thus the claim is established.

Question 11

Solution Since the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and an internal node x whose children are leaves in T, we prove that the method would eventually terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied if the condition given is considered as an antecedent of the claim.

Claim If the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and an internal node x whose children are leaves in T, the method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied.

Proof Since the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and an internal node x whose children are leaves in T, by tracing through the execution of the pseudocode shown in Figure 5, we can see that the test at line 1 is checked and failed, after the execution of the algorithm at line 6, the execution of the algorithm continues at line 7, where a ***try-catch*** statement is initialized, and the execution of the algorithm continues at line 8.

- If x has exactly 2 children, the test at line 8 is checked and passed, and the execution of the algorithm continues at line 9,
  - If the value of the key is less than or equal to the largest value stored at any leaf of the first subtree with root x, then the test at line 9 is checked and passed, thus the execution of the algorithm continues at line 10, where the algorithm is called recursively with the input key of type E and the first

child of x, which is a leaf in T. Suppose the first child of x is y,

- If the value of y is equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.firstChild)*, the steps at line 1, 2, 3, 4 are executed and an $ElementFoundException$ is thrown. Since the ***try-catch*** statement does not catch any exception whose type is $ElementFoundException$, an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x)* as well and the execution of the algorithm terminates.

  Therefore, the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the postcondition of the problem is true, thus the statement **(a)** is satisfied. Since the antecedents for both **(b)** and **(c)** of the postcondition of the problem are false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of y is not equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.firstChild)*, the steps at line 1, 2, 3, 5 are executed and an $NoSuchElementException$ is thrown. Since the ***try-catch*** statement catches every exception whose type is $NoSuchElementException$, the step at line 18 is executed and the method *addLeaf(key, x)* at line 19 is called with the input key and the node x.

  Since x has exactly 2 children, after the execution of *addLeaf(key, x)*, x added another child that is also a leaf storing key, thus x now has exactly 3 children, so T is still a 2-3 tree and the input key is added to the subset of E stored at leaves of T.

  Therefore, the antecedent of **(c)** of the postcondition of the problem is true, and the consequent of **(c)** of the postcondition of the problem is true, thus the statement **(c)** is satisfied. Since the antecedents for both **(a)** and **(b)** of the postcondition of the problem are false, the statements **(a)** and **(b)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of the key is greater than the largest value stored at any leaf of the first subtree with root x, then the test at line 9 is checked and failed, thus the execution of the algorithm continues at line 11, where the algorithm is called recursively with the input key of type E and the second child of x, which is a leaf in T. Suppose the second child of x is y,

  - If the value of y is equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.secondChild)*, the steps at line 1, 2, 3, 4 are executed and an $ElementFoundException$ is thrown. Since the ***try-catch*** statement does not catch any exception whose type is $ElementFoundException$, an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x)* as well and the execution of the algorithm terminates.

Therefore, the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the postcondition of the problem is true, thus the statement **(a)** is satisfied. Since the antecedents for both **(b)** and **(c)** of the postcondition of the problem are false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of y is not equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.secondChild)*, the steps at line 1, 2, 3, 5 are executed and an $NoSuchElementException$ is thrown. Since the ***try-catch*** statement catches every exception whose type is $NoSuchElementException$, the step at line 18 is executed and the method *addLeaf(key, x)* at line 19 is called with the input key and the node x.

  Since x has exactly 2 children, after the execution of *addLeaf(key, x)*, x added another child that is also a leaf storing key, thus x now has exactly 3 children, so T is still a 2-3 tree and the input key is added to the subset of E stored at leaves of T.

  Therefore, the antecedent of **(c)** of the postcondition of the problem is true, and the consequent of **(c)** of the postcondition of the problem is true, thus the statement **(c)** is satisfied. Since the antecedents for both **(a)** and **(b)** of the postcondition of the problem are false, the statements **(a)** and **(b)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If x has exactly 3 children, the test at line 8 is checked and failed, thus the execution of the algorithm continues at line 12,
  - If the value of the key is less than or equal to the largest value stored at any leaf of the first subtree with root x, then the test at line 12 is checked and passed, thus the execution of the algorithm continues at line 13, where the algorithm is called recursively with the input key of type E and the first child of x, which is a leaf in T. Suppose the first child of x is y,
    - If the value of y is equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.firstChild)*, the steps at line 1, 2, 3, 4 are executed and an $ElementFoundException$ is thrown. Since the ***try-catch*** statement does not catch any exception whose type is $ElementFoundException$, an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x)* as well and the execution of the algorithm terminates.

      Therefore, the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the postcondition of the problem is true, thus the statement **(a)** is satisfied. Since the antecedents for both **(b)** and **(c)** of the postcondition of the problem are false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of y is not equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.firstChild)*, the steps at line 1, 2, 3, 5 are executed and an $NoSuchElementException$ is thrown. Since the ***try-catch*** statement catches every exception whose type is $NoSuchElementException$, the step at line 18 is executed and the method *addLeaf(key, x)* at line 19 is called with the input key and the node x.

  Since x has exactly 3 children, after the execution of *addLeaf(key, x)*, x added another child that is also a leaf storing key, thus x now has exactly 4 children, so T satisfies the ***Modified Tree*** properties and the input key is added to the subset of E stored at leaves of T.

  Therefore, the antecedent of **(c)** of the postcondition of the problem is true, and the consequent of **(c)** of the postcondition of the problem is true, thus the statement **(c)** is satisfied. Since the antecedents for both **(a)** and **(b)** of the postcondition of the problem are false, the statements **(a)** and **(b)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of the key is greater than the largest value stored at any leaf of the first subtree with root x but is less than or equal to the largest value stored at any leaf of the second subtree with root x, then the test at line 12 is checked and failed, thus the execution of the algorithm continues the test at line 14, which is checked and passed, where the algorithm is called recursively with the input key of type E and the second child of x, which is a leaf in T. Suppose the second child of x is y,

  - If the value of y is equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.secondChild)*, the steps at line 1, 2, 3, 4 are executed and an $ElementFoundException$ is thrown. Since the ***try-catch*** statement does not catch any exception whose type is $ElementFoundException$, an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x)* as well and the execution of the algorithm terminates.

    Therefore, the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the postcondition of the problem is false, thus the statement **(a)** is satisfied. Since the antecedents for true **(b)** and **(c)** of the postcondition of the problem are false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

  - If the value of y is not equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.secondChild)*, the steps at line 1, 2, 3, 5 are executed and an $NoSuchElementException$ is thrown. Since the ***try-catch*** statement catches every exception whose type is $NoSuchElementException$, the step at line 18 is executed and the method *addLeaf(key, x)* at line 19 is called with the input key and the node x.

    Since x has exactly 3 children, after the execution of *addLeaf(key, x)*, x added another child that

is also a leaf storing key, thus x now has exactly 4 children, so T satisfies the ***Modified Tree*** properties and the input key is added to the subset of E stored at leaves of T.

Therefore, the antecedent of **(c)** of the postcondition of the problem is true, and the consequent of **(c)** of the postcondition of the problem is true, thus the statement **(c)** is satisfied. Since the antecedents for both **(a)** and **(b)** of the postcondition of the problem are false, the statements **(a)** and **(b)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of the key is greater than the largest value stored at any leaf of the second subtree with root x, then the tests at line 12 and line 14 are both failed, thus the execution of the algorithm continues the step at line 16, where the algorithm is called recursively with the input key of type E and the third child of x, which is a leaf in T. Suppose the third child of x is y,

  - If the value of y is equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.thirdChild)*, the steps at line 1, 2, 3, 4 are executed and an $ElementFoundException$ is thrown. Since the ***try-catch*** statement does not catch any exception whose type is $ElementFoundException$, an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x)* as well and the execution of the algorithm terminates.

    Therefore, the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the postcondition of the problem is true, thus the statement **(a)** is satisfied. Since the antecedents for both **(b)** and **(c)** of the postcondition of the problem are false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

  - If the value of y is not equal to the value of key, inside the execution of the recursive call *insertIntoSubtree(key, x.thirdChild)*, the steps at line 1, 2, 3, 5 are executed and an $NoSuchElementException$ is thrown. Since the ***try-catch*** statement catches every exception whose type is $NoSuchElementException$, the step at line 18 is executed and the method *addLeaf(key, x)* at line 19 is called with the input key and the node x.

    Since x has exactly 3 children, after the execution of *addLeaf(key, x)*, x added another child that is also a leaf storing key, thus x now has exactly 4 children, so T satisfies the ***Modified Tree*** properties and the input key is added to the subset of E stored at leaves of T.

    Therefore, the antecedent of **(c)** of the postcondition of the problem is true, and the consequent of **(c)** of the postcondition of the problem is true, thus the statement **(c)** is satisfied. Since the antecedents for both **(a)** and **(b)** of the postcondition of the problem are false, the statements **(a)** and **(b)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

Therefore, we showed that the execution of the algorithm eventually ends and the postcondition is satisfied in all cases, thus if the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and an internal node x whose children are leaves in T, the method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied.

Question 12

CITATION

Please note before grading, the format of the proof acts in a similar manner to the proof provided from page 42 to page 49 in L09_BST_1.pdf[4] in order to providing a completed, clear and professional proof.

Solution In order to prove the algorithm *insertIntoSubtree* in Figure 5 correctly solves the "Insertion into Subtree of This 2-3 Tree" problem, we prove that if the precondition of the problem is satisfied and the algorithm *insertIntoSubtree* is executed with a non-null key of type E and a non-null node x, then the algorithm *insertIntoSubtree* eventually ends and the postcondition of the problem is satisfied.

Claim If the precondition of the problem is satisfied and the algorithm *insertIntoSubtree* is executed with a non-null key of type E and a non-null node x whose depth is d, then the algorithm *insertIntoSubtree* eventually ends and the postcondition of the problem is satisfied.

Proof Since the precondition of the problem is satisfied and the algorithm *insertIntoSubtree* is called with a non-null key of type E and a non-null node x in T and the depth of x is $d \in \mathbb{N}$. We prove the claim by the strong form of mathematical induction on d. Cases that $d = 0$ and $d = 1$ will be used in the basis.

Basis (d=0) When $d = 0$, that is, the depth of the tree is zero, there exists only a leaf which is also the root of T by the definition, thus x is a leaf in T. Since in ***Question 10*** we have proved that, if the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and a node x that is a leaf in T, the method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied. Thus the claim holds in this case, as required.

Basis (d=1) When $d = 1$, that is, the depth of the tree is one, thus x is an internal node by the definition and it has exactly either two or three children and all children of x are leaves in T. Since in ***Question 11*** we have proved that, if the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the *insertIntoSubtree* method is executed with the input key of type E and an internal node x whose children are leaves in T, the method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied. Thus the claim holds in this case, as required.

Inductive Step: Let $k \geq 1$ be an integer. It is necessary and sufficient to use

Inductive Hypothesis: Suppose for all integers m such that $0 \leq m \leq k$, if the precondition of the problem is

satisfied and the algorithm *insertIntoSubtree* is executed with a non-null key of type E and a non-null node x whose depth is m, then the algorithm *insertIntoSubtree* eventually ends and the postcondition of the problem is satisfied.

to prove

Inductive Claim: If the precondition of the problem is satisfied and the algorithm *insertIntoSubtree* is executed with a non-null key of type E and a non-null node x whose depth is $k+1$, then the algorithm *insertIntoSubtree* eventually ends and the postcondition of the problem is satisfied.

Since $k \geq 1, k+1 \geq 2$, x is an internal node. Thus by tracing through the execution of the pseudocode shown in Figure 5, we can see that the test at line 1 is checked and failed, after the execution of the algorithm at line 6, the execution of the algorithm continues at the step of line 7, where a ***try-catch*** statement is initialized, then the execution of the algorithm continues at line 8.

- If x has exactly 2 children, the test at line 8 is checked and passed, and the execution of the algorithm continues at line 9,
    - If the value of the key is less than or equal to the largest value stored at any leaf of the first subtree with root x, then the test at line 9 is checked and passed, thus the execution of the algorithm continues at line 10, where the algorithm is called recursively with the input key of type E and the first child of x, since the depth of the subtree whose root is the first child of x is k, by the inductive hypothesis, the recursive execution of *insertIntoSubtree(key, x.firstChild)* eventually ends with the postcondition of the problem is satisfied when key of type E and the node $x.firstChild$ is given as input.

      Since only one antecedent from **(a)**, **(b)** and **(c)** described in the postcondition of the problem holds for a certain execution, thus in this case, the antecedent and the corresponding consequent hold.

        - If the antecedent and the consequent of **(a)** in the postcondition of the problem hold, that is, the input key already belongs to the subset of E stored at the leaves in the subtree with root $x.firstChild$, then an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x.firstChild)* and the subtree with root $x.firstChild$ is not changed. Since the ***try-catch*** statement does not catch any exception whose type is $ElementFoundException$, an $ElementFoundException$ is thrown by *insertIntoSubtree(key, x)* as well and the execution of the algorithm terminates.

          Therefore, for the execution of the algorithm *insertIntoSubtree* with a non-null key of type E and a non-null node x given as input, the antecedent of **(a)** of the postcondition of the problem is true, and the consequent of **(a)** of the postcondition of the problem is true, thus the statement **(a)** is satisfied. Since the antecedents for both **(b)** and **(c)** of the postcondition of the problem are

false, the statements **(b)** and **(c)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the antecedent and the consequent of **(b)** in the postcondition of the problem hold, that is, $x.firstChild$ is a leaf that stores an element of E that is not equal to the input key then a $NoSuchElementException$ is thrown and T is not changed. Since the depth of the subtree with root $x.firstChild$ is $k \geq 1$, $x.firstChild$ is an internal node, which contradicts that $x.firstChild$ is a leaf, thus the antecedent from **(b)** in the postcondition of the problem does not hold, so it is impossible that the antecedent and the consequent of **(b)** in the postcondition of the problem hold.

- If the antecedent and the consequent of **(c)** in the postcondition of the problem hold, that is, $x.firstChild$ is an internal node and the input key does not initially belong to the subset of E stored at the leaves of the subtree with root $x.firstChild$, then

  - the input key is added to the subset of E stored at the leaves of the subtree with root $x.firstChild$, which is otherwise unchanged;

  - either the subtree with root $x.firstChild$ satisfies the 2-3 Tree properties, or the subtree with root $x.firstChild$ satisfies the ***Modified Tree*** properties and $x.firstChild$ is now an internal node with four children.

  Since no exceptions are thrown during the execution of *insertIntoSubtree(key, x.firstChild)*, this execution of the algorithm continues at line 17 where the method *raiseSurplus* is called with x given as input. From the description of the *raiseSurplus* method, we can see that after the execution of the method *raiseSurplus*, T still satisfies the ***Modified Tree*** properties and either x has 4 children or no internal node of T with either 1 or 4 children. Also this execution of the algorithm terminates after calling the method.

  Therefore, for the execution of the algorithm *insertIntoSubtree* with a non-null key of type E and a non-null node x given as input, the antecedent of **(c)** of the postcondition of the problem is true, and the consequent of **(c)** of the postcondition of the problem is true, thus the statement **(c)** is satisfied. Since the antecedents for both **(a)** and **(b)** of the postcondition of the problem are false, the statements **(a)** and **(b)** are also satisfied. Thus the postcondition of the problem is satisfied in this case, as required.

- If the value of the key is greater than the largest value stored at any leaf of the first subtree with root x, then the test at line 9 is checked and failed, thus the execution of the algorithm continues at line 11, where the algorithm is called recursively with the input key of type E and the second child of x. The proof of this case is almost identical to the above one. One difference is key of type E and the node $x.secondChild$ are given as input for the recursive call *insertIntoSubtree* in the current case.

Therefore, the inductive claim holds if x has exactly 2 children.

- If x has exactly 3 children, the test at line 8 is checked and failed, and the execution of the algorithm continues at line 12,
    - If the value of the key is less than or equal to the largest value stored at any leaf of the first subtree with root x, then the test at line 12 is checked and passed, thus the execution of the algorithm continues at line 13, where the algorithm is called recursively with the input key of type E and the first child of x. The proof of this case is almost identical to that in the case when x has exactly 2 children and the value of the key is less than or equal to the largest value stored at any leaf of the first subtree with root x. And all statements of **(a)**, **(b)**, **(c)** in the postcondition of the problem also must be considered one by one.
    - If the value of the key is greater than the largest value stored at any leaf of the first subtree with root x but is less than or equal to the largest value stored at any leaf of the second subtree with root x, then the test at line 12 is checked and failed, thus the execution of the algorithm continues the test at line 14, which is checked and passed, so the execution of the algorithm continues at line 15, where the algorithm is called recursively with the input key of type E and the second child of x. The proof of this case is almost identical to that in the case when x has exactly 2 children and the value of the key is greater than the largest value stored at any leaf of the first subtree with root x. And all statements of **(a)**, **(b)**, **(c)** in the postcondition of the problem also must be considered one by one.
    - If the value of the key is greater than the largest value stored at any leaf of the second subtree with root x, then the tests at line 12 and line 14 are both failed, thus the execution of the algorithm continues at line 16, where the algorithm is called recursively with the input key of type E and the third child of x. The proof of this case is almost identical to that in the case when x has exactly 2 children and the value of the key is greater than the largest value stored at any leaf of the first subtree with root x. One difference is key of type E and the node $x.thirdChild$ are given as input for the recursive call *insertIntoSubtree* in the current case. And all statements of **(a)**, **(b)**, **(c)** in the postcondition of the problem also must be considered one by one.

  Therefore, the inductive claim holds if x has exactly 3 children.

Conclusion: Thus, by strong form of mathematical induction, we can conclude that, if the precondition of the problem is satisfied and the algorithm *insertIntoSubtree* is executed with a non-null key of type E and a non-null node x whose depth is d, then the algorithm *insertIntoSubtree* eventually ends and the postcondition of the problem is satisfied. That is, the algorithm *insertIntoSubtree* in Figure 5 correctly solves the "Insertion into Subtree of This 2-3 Tree" problem.

Question 13

CITATION

Please note before grading, the format of the solution in **Step 1** below acts in a similar manner to the solution provided in page 26 in L05_efficiency.pdf[3] in order to providing a completed, clear and professional solution.

Also for providing completed, clear and professional proofs, the format of the proof in **Step 2** acts in a similar manner to the proof provided from page 48 to page 49 in L01_intro_and_math_review.pdf[2] , the format of the proof in **Step 3** acts in a similar manner to the proof provided in page 9 in L06_asymptotic_notation.pdf[5] , the format of the proof in **Step 4** acts in a similar manner to the proof provided from page 25 to page 26 in L02_correctness.pdf[1] .

Solution We take 4 steps to answer the whole question.

1. Firstly, in order to show the number of steps executed by this method if it starts with its problem's precondition satisfied is in $O(depth(x))$ where x is the node given as input, we consider an execution of the algorithm with the precondition being satisfied, and the algorithm *insertIntoSubtree* is executed with a non-null key of type E and a non-null node x whose depth is $n \in \mathbb{N}$. Suppose that the number of steps executed by this method is $T_{insertIntoSubtree}(n)$ when the Uniform Cost Criterion is defined and used, we try to obtain the recurrence of the upperbound of $T_{insertIntoSubtree}(n)$.

2. Secondly, base on the recurrence of the upperbound of $T_{insertIntoSubtree}(n)$, we try to give the upperbound a closed form, and prove the closed from is true.

3. Thirdly, base on the closed form of the upperbound, we prove $T_{insertIntoSubtree}(n)$ is in $O(n)$ by the definition.

4. Finally, we try to give the method a bound function and prove the bound function is true.

Step 1 Since n is an integer such that $n \geq 0$,

- If $n = 0$, the depth of the subtree with root x is zero, there exists only a leaf which is also the root of T and no internal nodes exist by the definition, thus x is a leaf in this case, and only steps from line 1 to line 5 are involved in the calculation of $T_{insertIntoSubtree}(n)$ without any recursive calls, thus we can suppose a constant $c_0 > 0$ such that $c_0 \in \mathbb{N}$ and

$$T_{insertIntoSubtree}(n) = T_{insertIntoSubtree}(0) \leq c_0$$

- If $n \geq 1$, the depth of the subtree with root x is greater than or equal to one, thus x is an internal node. Therefore, x has exactly either 2 or 3 children. By inspection, we can see that no matter how many children that x has, after a constant number of steps being executed by the algorithm, the method *insertIntoSubtree(key, y)* is recursively called at some point such that y is one of the children of x, since the depth y is $n - 1$, the number of steps for the execution of *insertIntoSubtree(key, y)* is $T_{insertIntoSubtree}(n - 1)$.

- If an $NoSuchElementException$ is thrown after it, the exception will be caught and *addLeaf(key, x)* is called, whose execution takes at most a constant number of steps by the description given.
- If an $ElementFoundException$ is thrown after it, the exception will not be caught and the execution of the algorithm terminates immediately.
- If no exceptions are thrown after it, *raiseSurplus(x)* is called, whose execution takes at most a constant number of steps by the description given.

Thus, we can suppose a constant $c_1 > 0$ such that $c_1 \in \mathbb{N}$ and

$$T_{insertIntoSubtree}(n) \leq T_{insertIntoSubtree}(n-1) + c_1$$

Therefore, the upperbound of $T_{insertIntoSubtree}(n)$ can be written as

$$T_{insertIntoSubtree}(n) \leq \begin{cases} c_0 & \text{if } n = 0 \\ c_1 + T_{insertIntoSubtree}(n-1) & \text{if } n \geq 1 \end{cases}$$

where $c_0, c_1 \in \mathbb{N}$ and $c_0, c_1 > 0$.

Step 2 Base on the upperbound of the $T_{insertIntoSubtree}(n)$, we prove that

$$T_{insertIntoSubtree}(n) \leq c_1 n + c_0$$

Claim If $T_{insertIntoSubtree} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Step 1*** when $n \in \mathbb{N}$ and constants $c_0, c_1 \in \mathbb{N}$ such that $c_0, c_1 > 0$,

$$T_{insertIntoSubtree}(n) \leq c_1 n + c_0$$

Basis (n=0) When $n = 0$, by the definition of the recurrence, we have

$$T_{insertIntoSubtree}(n) = T_{insertIntoSubtree}(0) \leq c_0 = c_1 \cdot 0 + c_0 = c_1 n + c_0$$

as required in the case.

Inductive Step: Let $k \geq 0$ be an integer. It is necessary and sufficient to use

Inductive Hypothesis: Suppose that m is a non-negative integer. Suppose for all integers m such that $0 \leq m \leq k$,

If $T_{insertIntoSubtree} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Step 1*** then when $m \in \mathbb{N}$ and constants $c_0, c_1 \in \mathbb{N}$,

$$T_{insertIntoSubtree}(m) \leq c_1 m + c_0$$

to prove

Inductive Claim: If $T_{insertIntoSubtree} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Step 1*** then when $k + 1 \in \mathbb{N}$ and constants $c_0, c_1 \in \mathbb{N}$ such that $c_0, c_1 > 0$,

$$T_{insertIntoSubtree}(k + 1) \leq c_1(k + 1) + c_0$$

Since $k \geq 0$, $k + 1 \geq 1$, thus by the definition of the recurrence described in ***Step 1***, we have

$$T_{insertIntoSubtree}(k + 1) \leq c_1 + T_{insertIntoSubtree}(k)$$

By the inductive hypothesis which is applied since $0 \leq k \leq k$, we have

$$T_{insertIntoSubtree}(k + 1) \leq c_1 + (c_1k + c_0) = c_1(k + 1) + c_0$$

as required.

Conclusion: Therefore, by strong form of mathematical induction, we can conclude that, if $T_{insertIntoSubtree} : \mathbb{N} \to \mathbb{N}$ is a function satisfying the recurrence described in ***Step 1*** when $n \in \mathbb{N}$ and constants $c_0, c_1 \in \mathbb{N}$ such that $c_0, c_1 > 0$,

$$T_{insertIntoSubtree}(n) \leq c_1n + c_0$$

Step 3 Base on the closed form of the upperbound of $T_{insertIntoSubtree}(n)$ where n is the depth of the node x, we show that $T_{insertIntoSubtree}(n) \in O(n)$ by the definition.

Claim If an execution of the algorithm *insertIntoSubtree* starts with its problem's precondition satisfied and a non-null key of type E and a non-null node x whose depth is $n \in \mathbb{N}$ are given as input, then the number of steps executed by this method $T_{insertIntoSubtree}(n) \in O(n)$.

Proof By the definition of $O(n)$, it is sufficient to show that there exists $c > 0$ and $N_0 \geq 0$ for all $n \in \mathbb{N}$ such that $n \geq N_0$.

Let $c = c_0 + c_1$ and $N_0 = 1$ such that $c > 0$ and $N_0 \geq 0$. Let n be arbitrarily chosen from $\mathbb{N}$ such that $n \geq N_0 = 1$.

Since $c_0 > 0$ and $n \geq 1$, we have

$$c_0(1 - n) \leq 0$$

Thus

$$c_0 \leq c_0n$$

Then base on the closed form of the upper bound of $T_{insertIntoSubtree}(n)$ in ***Step 2***, we have

$$T_{insertIntoSubtree}(n) \leq c_1 n + c_0 \leq c_1 n + c_0 n = (c_1 + c_0)n = cn$$

Since n is arbitrarily chosen from $\mathbb{N}$ such that $T_{insertIntoSubtree}(n) \leq cn$ for all $n \in \mathbb{N}$ and $c = c_0 + c_1 > 0$, $N_0 = 1 \geq 0$ are both constants, $T_{insertIntoSubtree}(n) \in O(n)$ by the definition.

Step 4

Claim If an execution of the algorithm *insertIntoSubtree* starts with its problem's precondition satisfied and a non-null key of type E and a non-null node x in a 2-3 tree T whose depth is $n \in \mathbb{N}$ are given as input, then

$$f(n) = n$$

is a bound function of this method.

Proof We prove the claim by showing that $f(n) = n$ satisfied all 3 properties included in the definition of a bound function for a recursive algorithm.

- From the 2-3 Tree Properties we can see that n is the depth of T and is defined to be the number of edges in a longest path from the root to a leaf in T, thus n is an integer, that is, $f(n) = n$ is an integer-valued function of the integer n.
- We can see from the line 10, line 11, line 13, line 15 and line 16 of the algorithm described in Figure 5, when the algorithm is recursively call itself, the parameter of node x will be changed into one of its children, since the depth of the subtree with the root being one of its children is less than the depth of the subtree with the root x by 1, thus the value of n is reduced by at least 1.
- If the function's value is less than or equal to zero when the algorithm is applied, that is, $f(n) = n \leq 0$. Since x is not-null from part **(c)** in the precondition, $n \geq 0$. Thus $n = 0$, and the root x is a leaf in this case. Thus the execution of the algorithm passed the test at line 1, and the value of type E storing at x is assigned to a variable e at step of line 2, if the key is equal to the value of e, then the execution will pass the test at line 3 and throw an $ElementFoundException$ at line 4, otherwise the test at line 3 failed and the execution will throw a $NoSuchElementException$ at line 5, we can see in both cases the algorithm does not call itself recursively during the execution.

Since $f(n) = n$ where n is the depth of the subtree with root x satisfies all the properties included in the definition of a ***bound function*** for a recursive algorithm, it is a bound function of the recursive algorithm in Figure 5.

Question 14

Solution We take 3 steps to answer the whole question.

1. Firstly, we describe the problem need to solve.
2. Secondly, we describe the strategy and corresponding algorithm of ***delete***.
3. Thirdly, we describe the sketch of the proof of the correctness of the algorithm.

Step 1

If the precondition of the problem is satisfied, that is
(a) A 2-3 tree T that satisfies the 2-3 tree property is given
(b) A non-null key of type E is given as input.

After the execution of the algorithm **delete**,

the postcondition of the problem list below, is satisfied, that is
(a) If the input key belongs to the subset of E represented by T then the key is removed from the set. Otherwise, a $NoSuchElementException$ is thrown and T is not changed.
(b) T satisfied the 2-3 Tree properties given above.

Step 2 In order to delete a non-null key of type E from a 2-3 tree T whose root is x and the depth of T is d, we need to

1. Pass the key and the node x as arguments to the algorithm described in Figure 3, to confirm if the key belongs to the subset of E represented by T.

2. If the input key belongs to the subset of E represented by T, delete the key from the set, update all internal nodes that each whose tuple include the value. key, re-balance the T if necessary, to make sure the new T is still a 2-3 tree.

3. If the input key does not belong to the subset of E represented by T, throw a $NoSuchElementException$ and do not change T.

So we discuss how the deletion is implemented and we **assume** that if the non-null key of type E is found in T. We split it into 3 cases according to the depth of T.

Case 1 (d=0) When $d = 0$, that is, the depth of the tree is zero, there exists only a leaf which is also the root of T by the definition, thus x is a leaf in T. In this case remove the root directly and make T be an empty 2-3 tree.

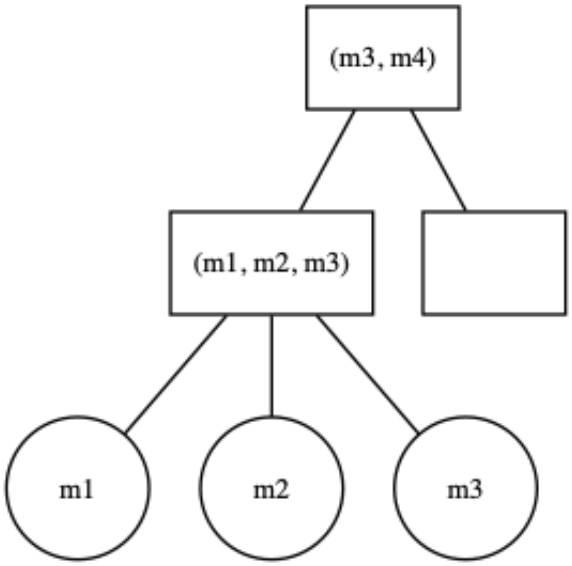
Case 2 (d=1) When $d = 1$, that is, the depth of the tree is one, thus x is an internal node by the definition and it has exactly either two or three children and all children of x are leaves in T. In this case, remove the leaf that stored the value of key.

- If the root has exactly 2 children before the deletion, change the root and make it stores the remaining value that is not the value of key, remove all leaves, and change the root from an internal node to a leaf.

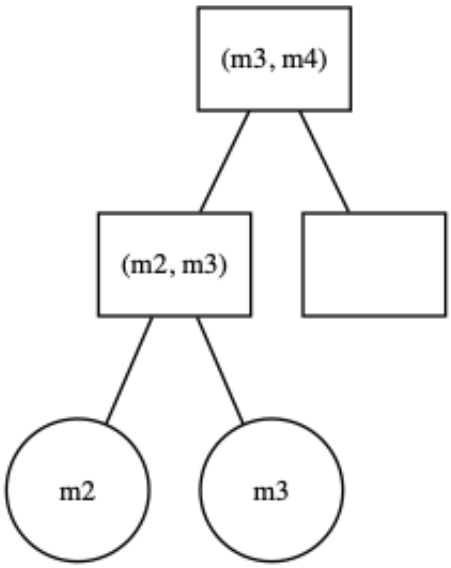
- If the root has exactly 3 children before the deletion, change the root and remove the value of key from its tuple, remove the leaf that stores the value of key as well.

Case 3 (d≥2) When $d \geq 2$, the depth of tree is greater or equal to 2, we split it into numerous subcases according the layout of an internal node whose children are leaves and one of leaves stores the value of key,

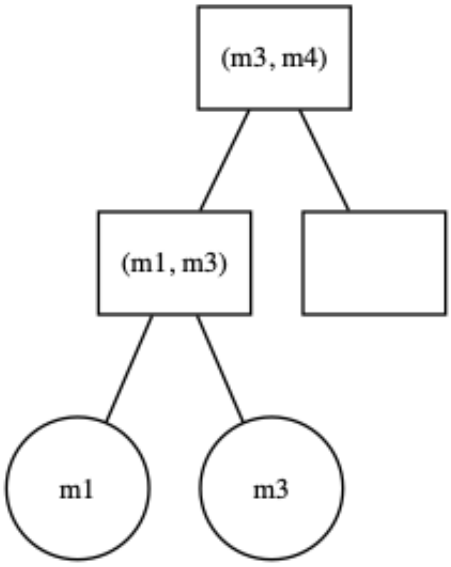
Subcase 1 Consider an internal node whose children are all leaves. The internal node stores a 3-tuple (m_1, m_2, m_3). Its parent is an internal node who has exactly 2 children. It is its parent's first child. The layout before deletion is shown below:



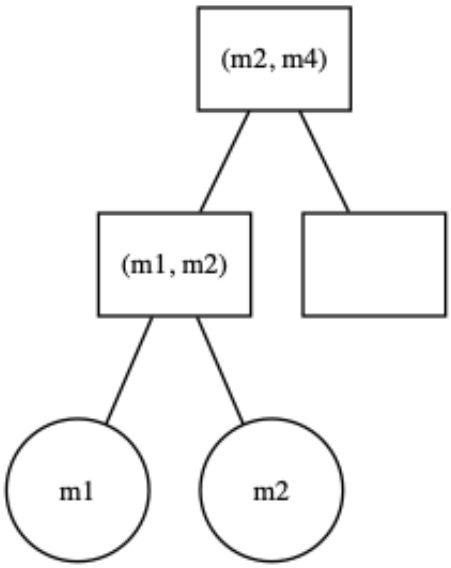
- If we need to remove m_1, then we need to remove the leaf stores m_1, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:



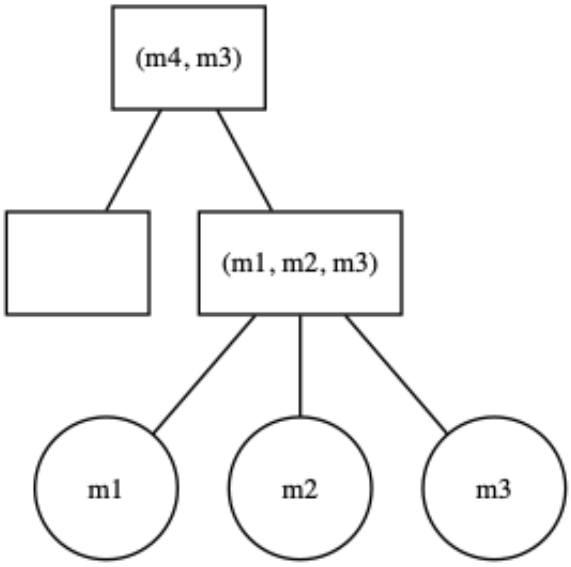
- If we need to remove m_2, then we need to remove the leaf stores m_2, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:



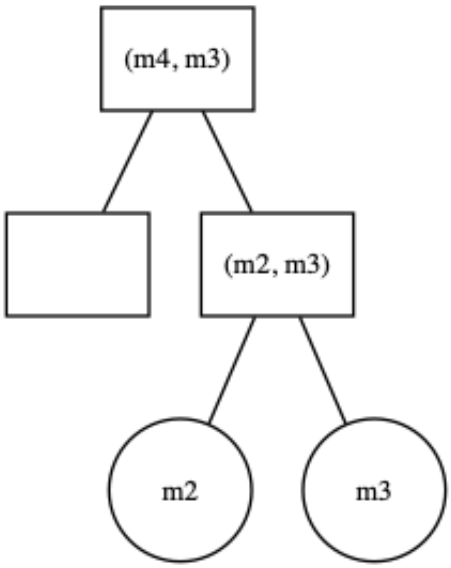
- If we need to remove m_3, then we need to remove the leaf stores m_3, update the internal node which store the 3-tuple (m_1, m_2, m_3) and its parent as well. The layout after deletion is shown below:



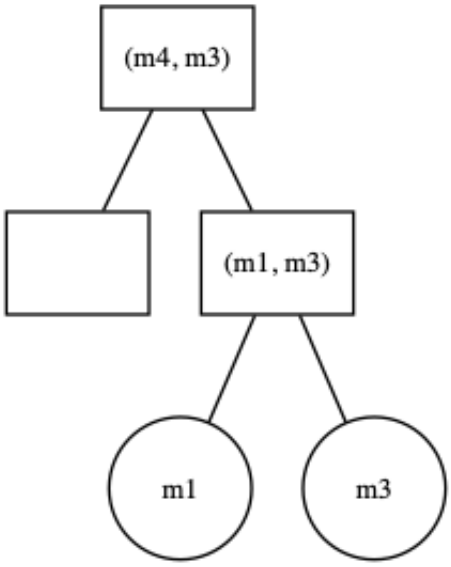
Subcase 2 Consider an internal node whose children are all leaves. The internal node stores a 3-tuple (m_1, m_2, m_3). Its parent is an internal node who has exactly 2 children. It is its parent's second child. The layout before deletion is shown below:



- If we need to remove m_1, then we need to remove the leaf stores m_1, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:

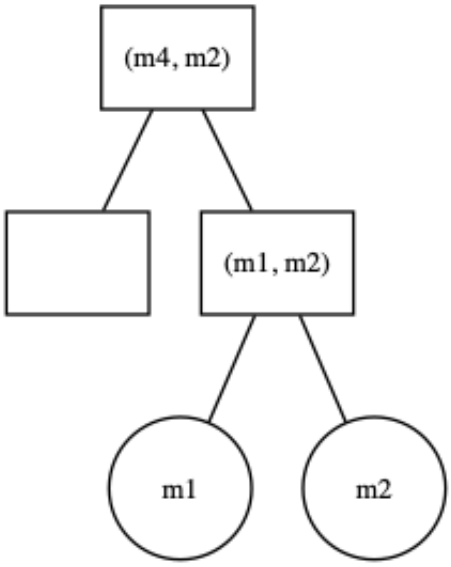


- If we need to remove m_2, then we need to remove the leaf stores m_2, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:

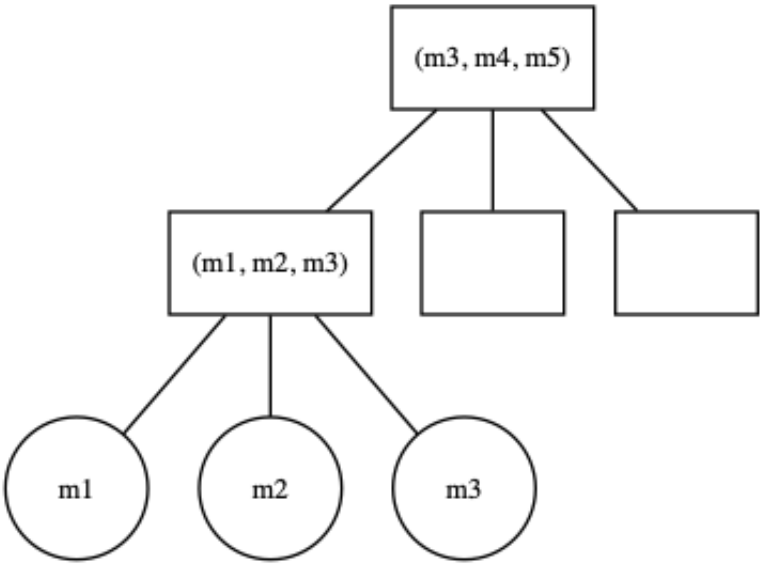


- If we need to remove m_3, then we need to remove the leaf stores m_3, update the internal node which store the 3-tuple (m_1, m_2, m_3) to (m_1, m_2) and back-trace from this internal node to the root,
    - For any internal node who includes m_3 in their tuple on the path while back-tracing, it must replace m_3 by m_2.
    - If encounter any internal node who does not includes m_3 in their tuple on the path while back-tracing, the back-trace can be terminated.

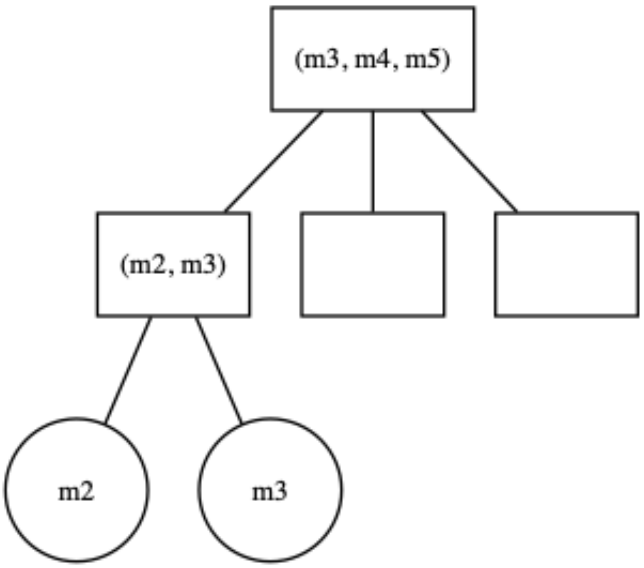
The layout after deletion is shown below:



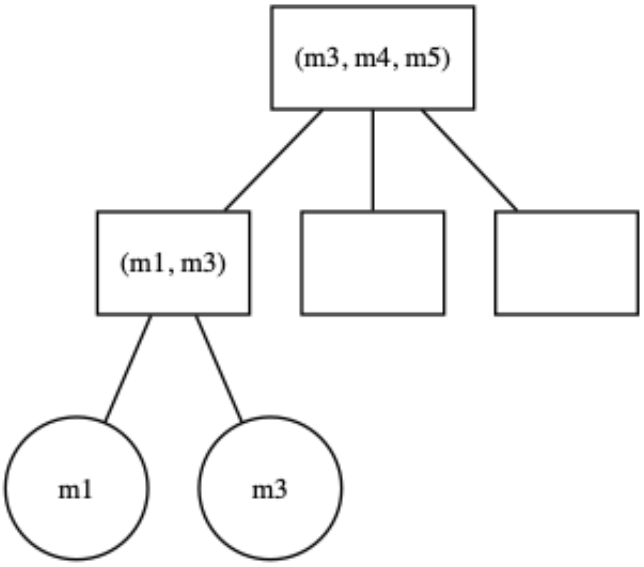
Subcase 3 Consider an internal node whose children are all leaves. The internal node stores a 3-tuple (m_1, m_2, m_3). Its parent is an internal node who has exactly 3 children. It is its parent's first child. The layout before deletion is shown below:



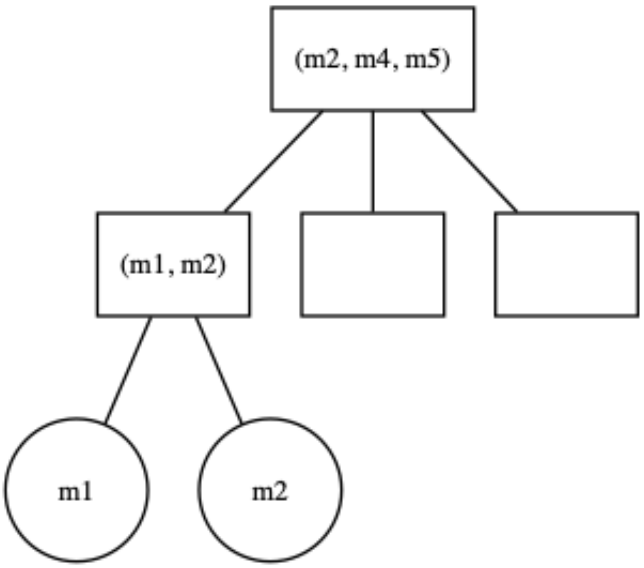
- If we need to remove m_1, then we need to remove the leaf stores m_1, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:



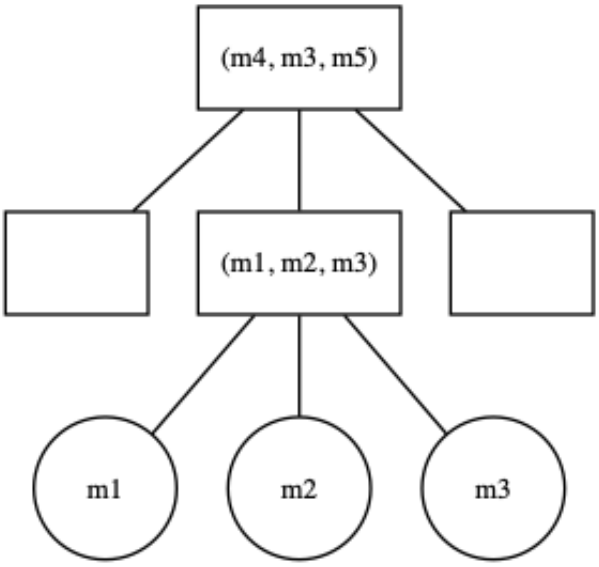
- If we need to remove m_2, then we need to remove the leaf stores m_2, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:



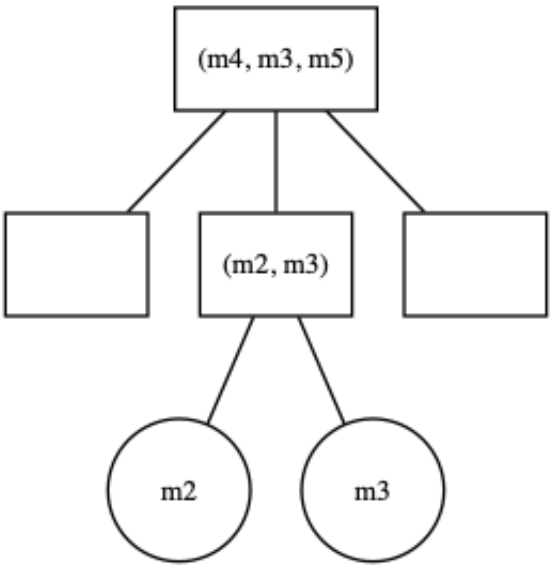
- If we need to remove m_3, then we need to remove the leaf stores m_3, update the internal node which store the 3-tuple (m_1, m_2, m_3) and its parent as well. The layout after deletion is shown below:



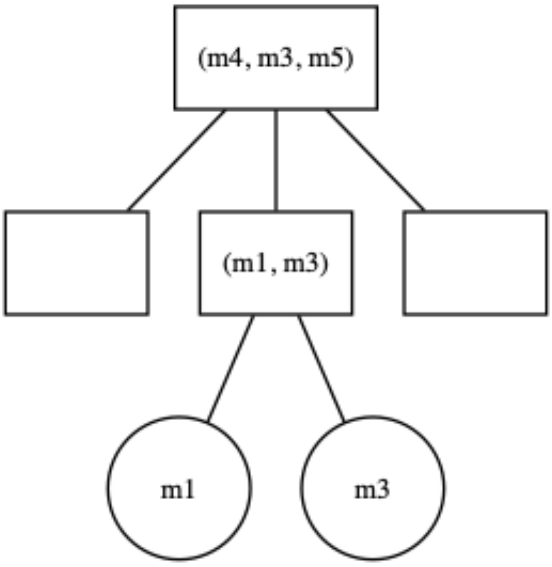
Subcase 4 Consider an internal node whose children are all leaves. The internal node stores a 3-tuple (m_1, m_2, m_3). Its parent is an internal node who has exactly 3 children. It is its parent's second child. The layout before deletion is shown below:



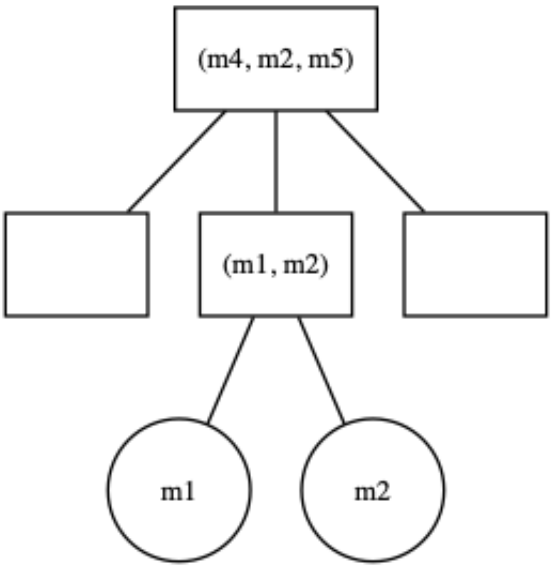
- If we need to remove m_1, then we need to remove the leaf stores m_1, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:



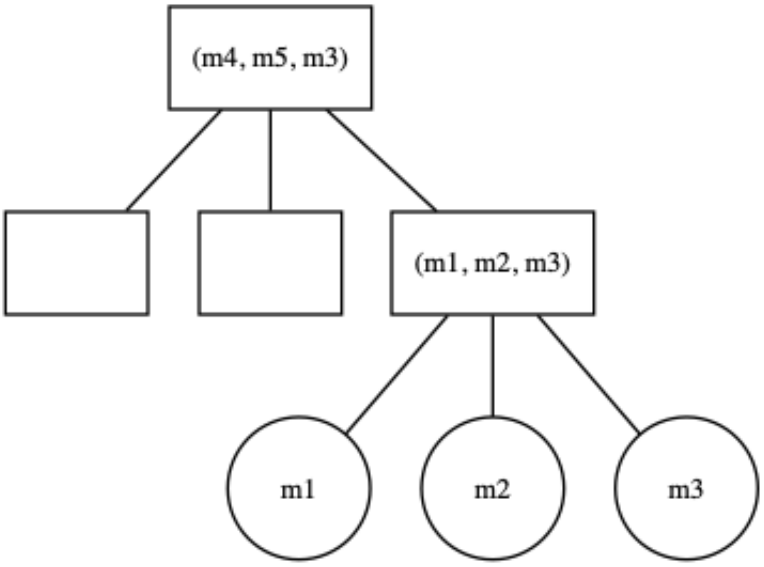
- If we need to remove m_2, then we need to remove the leaf stores m_2, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:



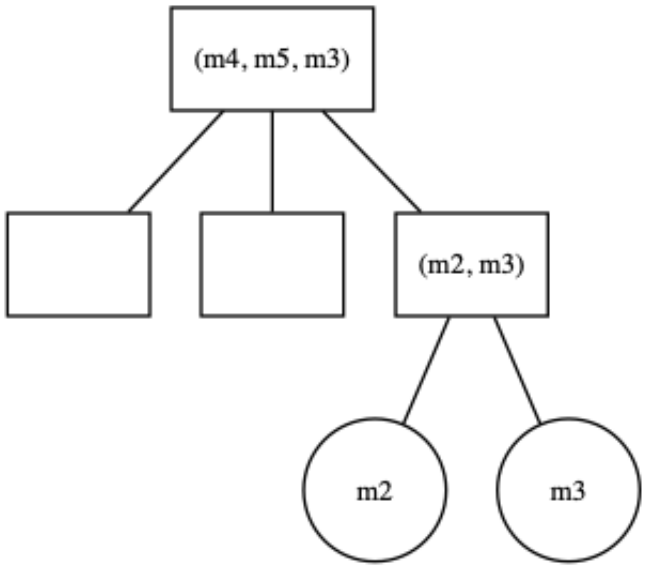
- If we need to remove m_3, then we need to remove the leaf stores m_3, update the internal node which store the 3-tuple (m_1, m_2, m_3) and its parent as well. The layout after deletion is shown below:



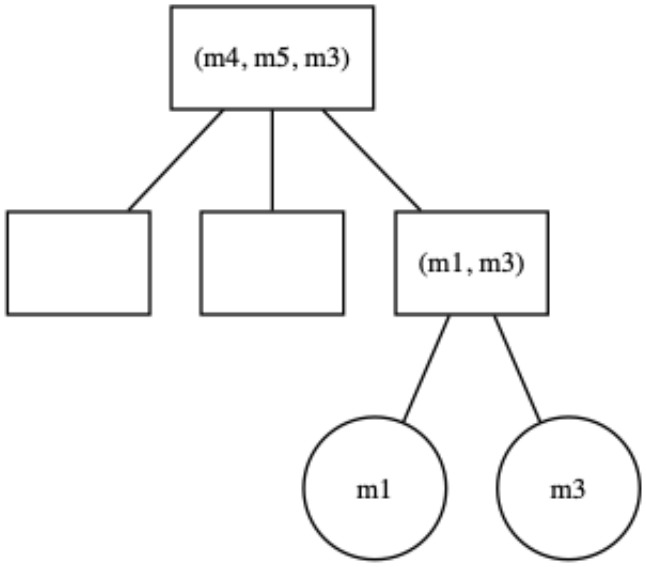
Subcase 5 Consider an internal node whose children are all leaves. The internal node stores a 3-tuple (m_1, m_2, m_3). Its parent is an internal node who has exactly 3 children. It is its parent's third child. The layout before deletion is shown below:



- If we need to remove m_1, then we need to remove the leaf stores m_1, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:

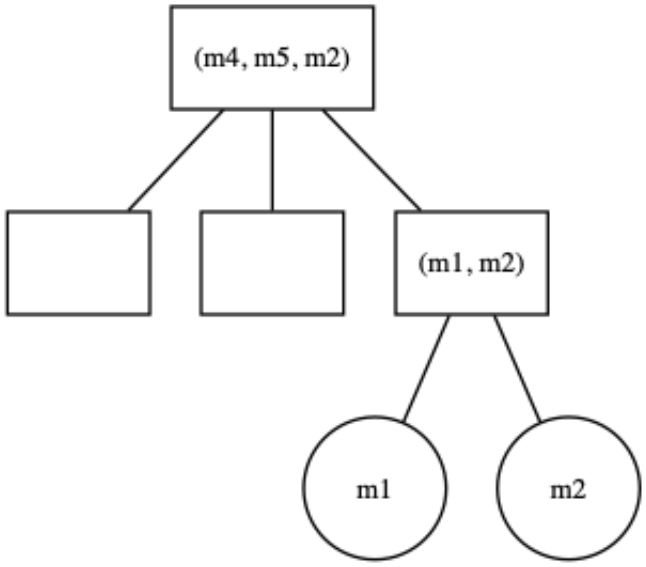


- If we need to remove m_2, then we need to remove the leaf stores m_2, update the internal node which store the 3-tuple (m_1, m_2, m_3) as well. The layout after deletion is shown below:

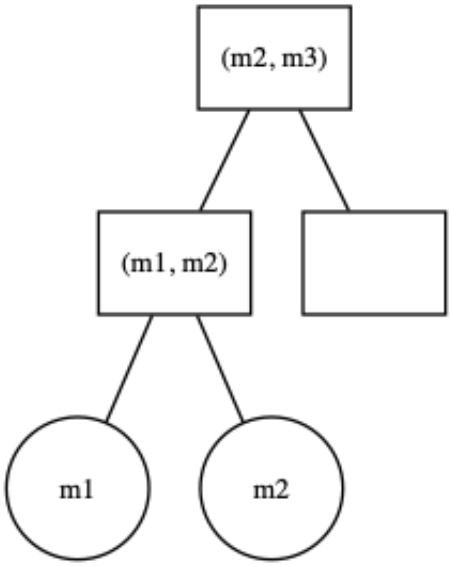


- If we need to remove m_3, then we need to remove the leaf stores m_3, update the internal node which store the 3-tuple (m_1, m_2, m_3) to (m_1, m_2) and back-trace from this internal node to the root,
    - For any internal node who includes m_3 in their tuple on the path while back-tracing, it must replace m_3 by m_2.
    - If encounter any internal node who does not includes m_3 in their tuple on the path while back-tracing, the back-trace can be terminated.

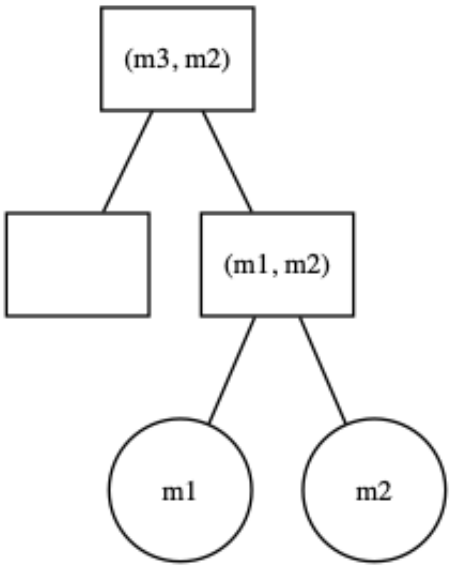
The layout after deletion is shown below:



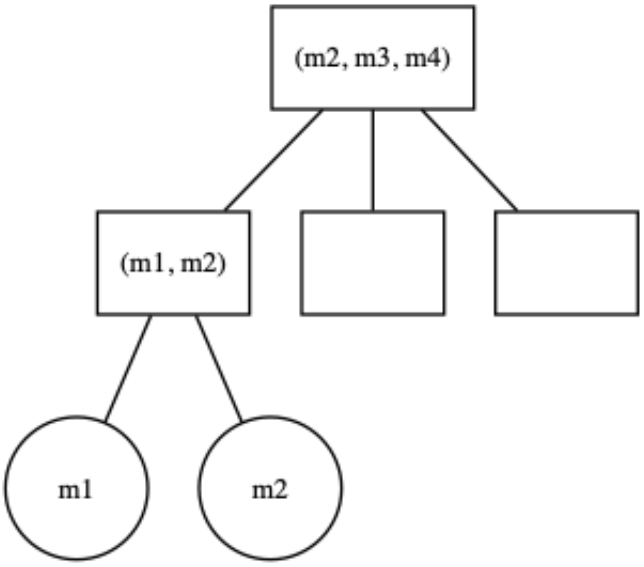
Subcase 6 Consider an internal node whose children are all leaves. The internal node stores a 2-tuple (m_1, m_2). Its parent is an internal node who has exactly 2 children. It is its parent's first child. The layout before deletion is shown below:



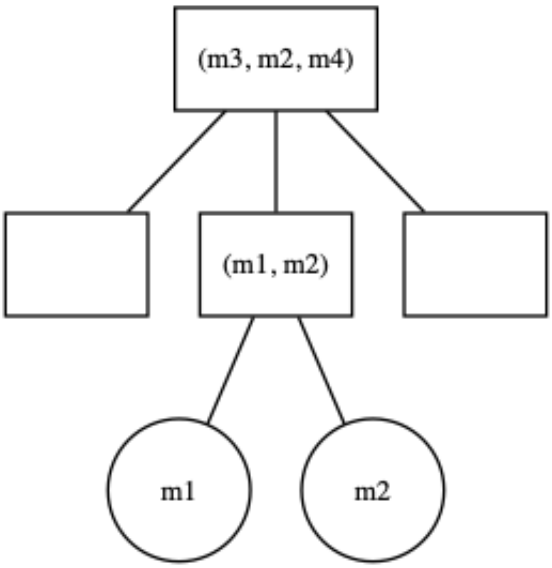
Subcase 7 Consider an internal node whose children are all leaves. The internal node stores a 2-tuple (m_1, m_2). Its parent is an internal node who has exactly 2 children. It is its parent's second child. The layout before deletion is shown below:



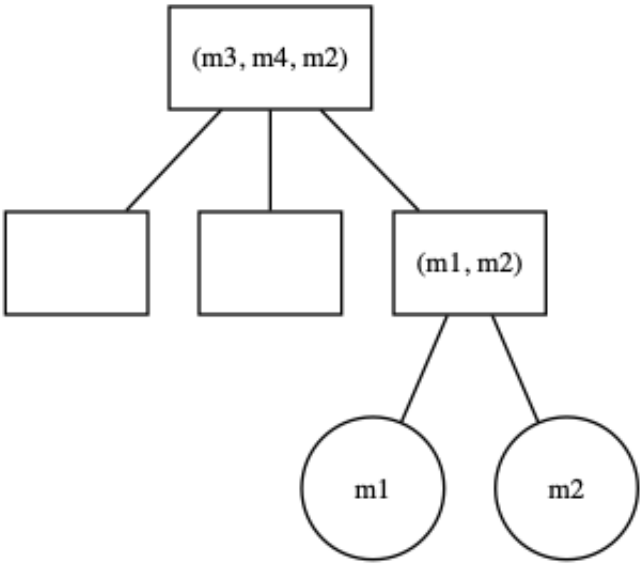
Subcase 8 Consider an internal node whose children are all leaves. The internal node stores a 2-tuple (m_1, m_2). Its parent is an internal node who has exactly 3 children. It is its parent's first child. The layout before deletion is shown below:



Subcase 9 Consider an internal node whose children are all leaves. The internal node stores a 2-tuple (m_1, m_2). Its parent is an internal node who has exactly 3 children. It is its parent's second child. The layout before deletion is shown below:



Subcase 10 Consider an internal node whose children are all leaves. The internal node stores a 2-tuple (m_1, m_2). Its parent is an internal node who has exactly 3 children. It is its parent's second child. The layout before deletion is shown below:



From *Subcase 6* to *Subcase 10*, if we remove m_1 or m_2, since all internal nodes must exactly have 2 or 3 children, thus T may not be balanced after the deletion and we must adjust the relation of the components involved if this happens.

Until the moment I wrote this paragraph I still cannot clearly classify all situations if a step to re-balance T is needed and how re-balance correctly. So far, I tried some cases and figured out it may depend on the structure of the nearest sibling internal node.

On the other hand, since ***no*** specific storage requirements and requirements of running time for the ***delete*** algorithm are described in this assignment, thus another strategy which can be used to remove a leaf is

- Confirm if input key belongs to the subset of E represented by T
- If input key belongs to the subset of E represented by T, then
    - If $d \geq 1$, we can merge all values in all leaves of this 2-3 Tree recursively from the leftmost child to the rightmost child and from the top to down then stores into an array list of type E. By the definition of a 2-3 tree, the array is in ascending order.
    - If $d = 0$, stores the value in the root to the array list directly.
- Another two sub-algorithms **obtainAllLeavesInOrder** and **mergeSubtree** are used in the step above, and both of them will not change T. Both of these algorithms have their own preconditions and postconditions, and their correctness can be proved separately.
    - For **mergeSubtree**:
        - Precondition
            - An non-null node x of a 2-3 tree T is given as input.
            - A copy of a reference of an array list of type E is given as input.
        - Postcondition
            - An array list of type E whose entries are all values in leaves is in ascending order and returned as output.
            - T is not changed and it still satisfies the 2-3 Tree properties.
    - For **obtainAllLeavesInOrder**
        - Precondition
            - This nonempty 2-3 Tree, T, satisfies the 2-3 Tree Properties.
        - Postcondition
            - An array list of type E whose entries are all values in leaves is in ascending order and returned as output.
            - T is not changed and it still satisfies the 2-3 Tree properties.
- Delete the key from the array list obtained, and a deletion in an array will not affect its ascending order.
- Reset the root of T to null.
- Iterate the array list, execute the **insert** algorithm for the "Insertion into This 2-3 Tree" Problem and make

every entry in the list as input key during the iteration.

- Since we have proved that after the execution of the **insert** algorithm T is still a 2-3 tree, thus T is still a 2-3 tree after the iteration.

Therefore, I decide to use this strategy for all cases when $d \geq 0$. And the pseudocode of the algorithm are list in the next page:

```
void mergeSubtree(node x, ArrayList<E> result) {
    switch (the number of children that x has) {
        case 0:
            result.add(x)
            break
        case 2:
            mergeSubtree(x.firstChild, result)
            mergeSubtree(x.secondChild, result)
            break
        default:
            mergeSubtree(x.firstChild, result)
            mergeSubtree(x.secondChild, result)
            mergeSubtree(x.thirdChild, result)
    }
}

ArrayList<E> obtainAllLeavesInOrder() {
    ArrayList<E> result = new ArrayList<>()
    mergeSubtree(root, result)
    return result
}

void delete (E key) {
    if (T is not empty) {
        get(key, root)
        ArrayList<E> leavesInOrder := obtainAllLeavesInOrder()
        remove key from leavesInOrder
        n := leavesInOrder.size
        root := null
        try {
            int i := 0
            while (i < n) {
                insert(leavesInOrder.get(i))
                i := i + 1
            }
        } catch (ElementFoundException ex) {
            do nothing
        }
    }
}
```

Step 3 The sketch of the proof of correctness of the ***delete*** algorithm:

- We first prove the correctness of **mergeSubtree** by strong form of mathematical induction on the depth of

the 2-3 tree with root x.

- Since the **mergeSubtree** algorithm is proved to be correct, by inspection we can see that if the postcondition of **mergeSubtree** is satisfied, then the postcondition of **obtainAllLeavesInOrder** is satisfied as well.

- After the proof of correctness of **mergeSubtree** and **obtainAllLeavesInOrder**, we start to prove the correctness of the **delete** algorithm, note that the algorithm **insert** is involved in the *while* loop and has been proved to be correct before.

- In order to prove the correctness of an algorithm that includes a *while* loop, we prove its partial correctness and its termination.

    - We prove the partial correctness of the **delete** algorithm by its definition.
    - We prove the termination of the **delete** algorithm by **Loop Theorem #2**, thus we have to find a bound function for the *while* loop, from the pseudocode given above we can prove that

      $$f(n, i) = n - i$$

      is a bound function of the **delete** algorithm where n is the length of *leavesInOrder* arraylist and i is an integer variable such that $n \geq 0$ and $0 \leq i \leq n$ by satisfying all 3 properties included in the definition of a bound function for a *while* loop.

- Since no recursive calls on the algorithm itself for **delete**, we can claim the proof of correctness of the **delete** algorithm.

Question 15

NOTE The file *TwoThreeTree.java* has been uploaded to D2L dropbox.

References

[1] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction to the Analysis of Algorithms, Lecture #3: Introduction to the Correctness of Algorithms II — Correctness of Simple Algorithms with a while Loop. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/2_Analysis/L02/L02_correctness.pdf

[2] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction and Mathematics Review, Lecture #1: Introduction and Mathematics Review. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/1_Introduction/L01/L01_intro_and_math_review.pdf

[3] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction to the Analysis of Algorithms, Lecture #5: Analyzing the Running Times of Algorithms. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/2_Analysis/L05/L05_efficiency.pdf

[4] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Binary Search Trees, Lecture #9: Binary Trees and Binary Search Trees — Definitions and Searches. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/4_BSTs/L09/L09_BST_1.pdf

[5] Wayne Eberly, 2019, CPSC 331: Data Structures, Algorithms, and Their Analysis: Spring, 2019, Introduction to the Analysis of Algorithms, Lecture #6: Asymptotic Notation. Retrieved from http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC331/2019a/2_Analysis/L06/L06_asymptotic_notation.pdf