Boxiao Li 30069613
Chunyu Li 30056553
Sijia Yin 30049836

CPSC331 Assignment2

Date: June 3, 2019.

Group: Boxiao Li 30069613 \ Chunyu Li 30056553 \ Sijia Yin 30049836

1. To prove that if T represents a subset S⊆E such that |S|=n∈N, then T has at most n-1 internal nodes, we eager to use mathematics induction. Since T is a nonempty 2-3 tree, so that it includes at least one node, which means that when n≥1, then T has at most n-1 internal nodes.

Base case:

When the depth d=0, which means there is |S|=n=1∈N, and x is both root and leaf, and x is not an internal node. Since 0≤ n-1=1-1=0, which conform to the condition.

Inductive step:

Let k be an integer such that k≥1.

Inductive hypothesis:

Assume that any 2-3 tree T with depth k containing n elements so that the 2-3 tree T has at most n-1 internal nodes.

Inductive claim:

Assume that any 2-3 tree T with depth k+1 containing n elements so that the 2-3 tree T has at most n-1 internal nodes.

Proof:

Because we have k≥1, thus k+1≥2. Known from the 2-3 tree properties *(b) Every internal node of T has (exactly) either two or three children—which are each either leaves or internal nodes of T*. So that we have two cases on this problem.

Case 1: When the internal node of T has exactly two children.

Since we have properties *(h) Each node x of T is at the root of a 2-3 tree as well. That is, the subtree with root x also satisfies properties (a)-(g), for every node x in T*. So that we can assume that a and b are the children of T. Let a be the first child of the first subtree of T and b be the second child of the second subtree of T. Since |S|=n∈N, we can say that $n_a+n_b=n$. By the inductive hypothesis, the subtree a has at most n_a -1 internal nodes and the subtree b has at most n_b-1 internal nodes. So, the T has n_a -1+n_b-1+1= n_a+n_b-1=n-1≤ n-1, which conform to the inductive claim.

Case 2: When the internal node of T has exactly three children.

Since we have properties *(h) Each node x of T is at the root of a 2-3 tree as well. That is, the subtree with root x also satisfies properties (a)-(g), for every node x in T*. So that we can assume that a, b and c are the children of T. Let a be the first child of the first subtree of T and b be the second child of the second subtree of T and c be the third child of the third subtree of T. Since |S|=n∈N, we can say that $n_a+n_b+n_c=n$. By the inductive hypothesis, the subtree a has at most n_a -1 internal nodes and the subtree b has at most n_b-1 internal nodes and the subtree c has at most n_c-1 internal nodes. So, the T has n_a -1+n_b-1+ n_c-1+1= $n_a+n_b+n_c$ -2=n-2≤ n-1, which conform to the inductive claim.

Proof closure:

Since the proof above, for any 2-3 tree T with depth k+1 containing n element, it will have

at most n-1 internal nodes. Thus, if T represents a subset $S \subseteq E$ such that $|S|=n \in N$, then T has at most n-1 internal nodes.

2. To prove that $2^d \leq n \leq 3^d$ which d is the non-empty 2-3 tree with depth $d \subseteq N$ and n is the size of the subset of E. We will use the mathematics induction to prove that. From the properties of the 2-3 tree. We have the base case when d=0 and d=1.

Base case:

when d=0, the non-empty tree with n=1, which means there is only one element in the subset of E. Thus, the base cases for d=0, there is n=1, since $2^d=2^0 \leq n=1$ and $1=n \leq 3^d= 3^0$, which conform to the condition.

when d=1, the non-empty tree with n=2 or n=3, which means there are 2 or 3 elements in the subset of E. Thus, the base cases for d=1, when is n=2, since $2^d=2^1 \leq n=2$ and $2=n \leq 3^d=3^1$; when is n=3, since $2^d=2^1 \leq n=3$ and $3=n \leq 3^d=3^1$, which conform to the condition.

Inductive step:

let k be an integer such that $k \geq 2$.

Inductive hypothesis:

Assume that any 2-3 tree T with depth k containing n elements so that $2^k \leq n \leq 3^k$.

Inductive claim:

Assume that any 2-3 tree T with depth k+1 containing n elements so that $2^{k+1} \leq n \leq 3^{k+1}$.

Proof:

Since $k \geq 2$, so that $n \geq 4$. We have two cases, when all the internal nodes have 2 elements in it and when all the internal nodes have 3 elements in it. To be more specific, when every internal node has 2 elements the size of E is the minimum n with the same k. In other word, when every internal node has 3 elements the size of E is the most n with the same k.

Case 1: When all the internal nodes have 2 elements in it.

Since $k \geq 2$, so that $d=k+1 \geq 3$. So that we can assume that a and b are the children of T. Let a be the first child of the first subtree of T and b be the second child of the second subtree of T. Since $| E |=n \in N$, we can say that $n_a+n_b=n$. Thus, by the inductive hypothesis, we can say that $2^k \leq n_a \leq 3^k$ and $2^k \leq n_b \leq 3^k$. Since $n_a+n_b=n$, we have $2^k +2^k \leq n_a+n_b \leq 3^k +3^k$, which means that $2^{k+1} \leq n_a+n_b \leq 2x3^k$, thus, $2^{k+1} \leq n \leq 3^{k+1}$, which conform to the inductive claim.

Case 2: When all the internal nodes have 3 elements in it.

Since $k \geq 2$, so that $d=k+1 \geq 3$. So that we can assume that a, b and c are the children of T. Let a be the first child of the first subtree of T and b be the second child of the second subtree of T and c be the third child of the third subtree of T. Since $| E |=n \in N$, we can say that $n_a+n_b+n_c =n$. Thus, by the inductive hypothesis, we can say that $2^k \leq n_a \leq 3^k$ and $2^k \leq n_b \leq 3^k$. and $2^k \leq n_c \leq 3^k$ Since $n_a+n_b+n_c =n$, we have $2^k +2^k+2^k \leq n_a+n_b+n_c \leq 3^k +3^k+3^k$, which means that $2^{k+1}+2^k \leq n_a+n_b+n_c \leq 3^{k+1}$, thus, $2^{k+1} \leq n \leq 3^{k+1}$, which conform to the inductive claim.

Proof closure:

Since the proof above, $2^d \leq n \leq 3^d$ which d is the non-empty 2-3 tree with depth $d \subseteq N$ and n is the size of the subset of E.

3. To prove that there exists a non-empty 2-3 tree with depth d, representing a subset of E with size exactly 2^d. We will use the strong form mathematics induction to prove that. From the properties of the 2-3 tree. We have the base case when d=0 and d=1.

Base case:

when d=0, the non-empty tree with |E|=1, which means there is only one element in the subset of E. Thus, the base cases for d=0, there is |E|=1, since $1=|E|\geq 2^d=2^0$, which conform to the condition.

when d=1, the non-empty tree with |E|=2 or |E|=3, which means there are 2 or 3 elements in the subset of E. Thus, the base cases for d=1, when is |E|=2, since $2=|E|\geq 2^d=2^1$; when is |E|=3, since $2=|E|\geq 2^d=2^1$, which conform to the condition.

Inductive step:

let k be an integer such that $k\geq 2$.

Inductive hypothesis:

Assume that for $|E|\geq 2^k$, there exists a non-empty 2-3 tree with depth d, representing a subset of E with size exactly 2^k.

Inductive claim:

Assume that for $|E|\geq 2^{k+1}$, there exists a non-empty 2-3 tree with depth d, representing a subset of E with size exactly 2^{k+1}.

Proof:

Since $k\geq 2$, so that $d=k+1\geq 3$. So that we can assume that a and b are the children of T. Let a be the first child of the first subtree of T and b be the second child of the second subtree of T. Since $|E|=n\in N$, we can say that $n_a+n_b=n$. Thus, by the inductive hypothesis, we can say that $|E_a|=\geq 2^k$ and $|E_b|=\geq 2^k$. Thus, we can say that $|E|\geq 2^k+2^k=2^{k+1}$, which conform to the inductive claim.

Proof closure:

Since the proof above, there exists a non-empty 2-3 tree with depth d, representing a subset of E with size exactly 2^d.

4. To prove that the depth of the subtree with root x is a bound function for the recursive algorithm, I must prove all 3 properties of recursive bound functions:

   Since depth of the subtree with root x is specified as an integer input then this is an integer valued-function.

   On line 5, the execution goes into the if-else condition part, which means that the x has two children or x has three children. But whatever how many children x has, the code will be called recursively. To be more specific, if x has two children, then the algorithm will compare the key with the firstMax x, either it will lager or less than, the code will be called recursively. It is obvious that whenever the function is called recursively, the value of the parameter x is reduced by at least one, thus the bound function the depth of the subtree with root x is being reduced by at least one.

   In the algorithm, the value of x must be ≥ 1 as stated in the precondition. Thus, the only way for the function to be called with $x \leq 1$ and for the precondition ($x \geq 1$) to be satisfied is when $x = 1$. When $x = 0$, which means, x is both the root and leaf. Thus, the test at line1 passes, the program continues on to line2 or line4 where execution ends, thus it is not called recursively when $x \leq 1$.

   Since all 3 conditions are satisfied, then the depth of the subtree with root x is a bound function for the recursive algorithm.

5. With induction on the depth d of the subtree with root x as input. We will use the strong form of mathematical induction to prove that this algorithm correctly solves the "Searching in a Subtree of This 2-3 Tree" problem. And from the assumed in the footnote on page 3, that null elements of E are never stored in a 2-3 tree — and that no attempts to search for, insert or delete null elements are ever made. The case that d = −1, which means x is null, will not be considered in the basis. Thus, the case that d=0 will be considered in the basis

Basis:

when d = 0, so that x is both the root and leaf. Thus, the execution of the algorithm will go to the line1. Then the execution will go to the inner if-else condition which is line2, if x equal to the key, then the algorithm will execute line3 and return x. If x not equal to the key, it will execute line4 and a NoSuchElementException being thrown.

Inductive step:

Let k be an integer such that $k \geq 0$.

Inductive Hypothesis:

The search algorithm is executed with a key key∈E and a (impossibly null) node y in this 2-3 tree, such that the depth of the subtree with root y has depth at most k as inputs. Then this execution of the algorithm will eventually stop. If there is no node storing the input E key in the subtree with root y, then a NoSuchElementException is thrown. If there is a node storing the input E key in this subtree then the value stored at the leaf is returned as output.

Inductive Clam:

The search algorithm is executed with a key key∈E and a (impossibly null) node z in this 2-3 tree, such that the depth of the subtree with root z has depth at most k+1 as inputs.

Then this execution of the algorithm will eventually stop. If there is no node storing the input E key in the subtree with root z, then a NoSuchElementException is thrown. If there is a node storing the input E key in this subtree then the value stored at the leaf is returned as output.

Proof Steps:

The search algorithm is executed with an input key key E and a node z in this 2-3 tree, such that the depth of the subtree with root z is k+1. Since k≥0, k≥1 and the node z cannot be null.

Let h∈E be the key stored at z. Then there are two cases, when the node has two elements and when the node has three elements.

Case 1:

When the node has two elements.

If key=h, which means the key we want to search is exactly in the 2-3 tree, then the searching is succeed. Code will go to line5. If the h is firstMax in the z, then the execution will go to line7 will call the recursive. If the h is not firstMax in the z, then the execution will go to line7 will call the recursive. Final the tree will be search at the leaves. Thus, the algorithm will always execute the line1 at the final. If the element key

we want to search is not in the tree, then a NoSuchElementException is thrown. In other word, if the elements we want to search for is in the tree, the value stored at the leaf is returned as output. Since the execution of the algorithm with inputs key and z and ending with a recursive application of the algorithm which establish the Inductive Claim.

If key<h, which means the key we want to search is in the left subtree of the 2-3 search tree. Code will go to line5. Let w be the left child of z. If the w is firstMax in the left subtree root and key=w, then the execution will go to line7 will call the recursive. If the w is not firstMax in the left subtree root and key=w, then the execution will go to line8 will call the recursive. Final the tree will be search at the leaves. Thus, the algorithm will always execute the line1 at the final. If the element key we want to search is not in the tree, then a NoSuchElementException is thrown. In other word, if the elements we want to search for is in the tree, the value stored at the leaf is returned as output. Since the execution of the algorithm with inputs key and z and ending with a recursive application of the algorithm which establish the Inductive Claim.

If key<h, which means the key we want to search is in the right subtree of the 2-3 search tree. Code will go to line5. Let w be the right child of z. If the w is firstMax in the right subtree root and key=w, then the execution will go to line7 will call the recursive. If the w is not firstMax in the right subtree root and key=w, then the execution will go to line8 will call the recursive. Final the tree will be search at the leaves. Thus, the algorithm will always execute the line1 at the final. If the element key we want to search is not in the tree, then a NoSuchElementException is thrown. In other word, if the elements we want to search for is in the tree, the value stored at the leaf is returned as output. Since the execution of the algorithm with inputs key and z and ending with a recursive application of the algorithm which establish the Inductive Claim.

Case 2:

When the node has three elements.

If key=h, which means the key we want to search is exactly in the 2-3 tree, then the searching is succeed. Code will go to out else line between line8 and line9. If the h is firstMax in the z, then the execution will go to line10 will call the recursive. If the h is secondMax in the z, then the execution will go to line12 will call the recursive. If the h is thirdMax in the z, then the execution will go to line13 will call the recursive. Final the tree will be search at the leaves. Thus, the algorithm will always execute the line1 at the final. If the element key we want to search is not in the tree, then a NoSuchElementException is thrown. In other word, if the elements we want to search for is in the tree, the value stored at the leaf is returned as output. Since the execution of the algorithm with inputs key and z and ending with a recursive application of the algorithm which establish the Inductive Claim. And the case is the same as when h is the second or third element in z.

If key<h and h is the first element in z, which means the key we want to search is

in the first subtree of the 2-3 search tree. Code will go to line5. Let w be the first child of z. If the w is firstMax in the first subtree root and key=w, then the execution will go to line10 will call the recursive. If the w is secondMax in the second subtree root and key=w, then the execution will go to line12 will call the recursive. If the w is thirdMax in the third subtree root and key=w, then the execution will go to line13 will call the recursive. Final the tree will be search at the leaves. Thus, the algorithm will always execute the line1 at the final. If the element key we want to search is not in the tree, then a NoSuchElementException is thrown. In other word, if the elements we want to search for is in the tree, the value stored at the leaf is returned as output. Since the execution of the algorithm with inputs key and z and ending with a recursive application of the algorithm which establish the Inductive Claim. And the case is the same as when h is the second or third element in z.

If key>h, which means the key we want to search is in the first subtree of the 2-3 search tree. Code will go to line5. Let w be the first child of z. If the w is firstMax in the first subtree root and key=w, then the execution will go to line10 will call the recursive. If the w is secondMax in the second subtree root and key=w, then the execution will go to line12 will call the recursive. If the w is thirdMax in the third subtree root and key=w, then the execution will go to line13 will call the recursive. Final the tree will be search at the leaves. Thus, the algorithm will always execute the line1 at the final. If the element key we want to search is not in the tree, then a NoSuchElementException is thrown. In other word, if the elements we want to search for is in the tree, the value stored at the leaf is returned as output. Since the execution of the algorithm with inputs key and z and ending with a recursive application of the algorithm which establish the Inductive Claim. And the case is the same as when h is the second or third element in z.

Proof closure:

Since the execution ends and satisfied the post-condition of the algorithm, the algorithm correctly solves the “Searching in a Subtree of This 2-3 Tree” problem.

6. Because the depth of a tree always bigger or equal to zero, thus the input k starts with zero.

   When k = 0 and k as the input, the algorithm executes 3 steps (at lines1、2 and 3).

   When k≥1 and let k be the input, the algorithm will be tested by line 1, 5, 9, 11 and 13, and only passed by 13, then will call itself recursively with (k-1) as the input. Thus, the number of steps will be $5+T_{get}(k-1)$.

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

7. Upper bound: $T_{get}(k) \leq 3+5k$

I will use mathematical induction to improve the upper bound.

K=0, k=1 will be consider as basis case.

Basis:

Let k=0 as input, line 1 will be tested and passed, line 2 and line 3 will be executed. Thus, 3 lines executed, the claim (3+5*0) holds on this case.

Let k=1 as input, line 1 will be tested and failed, line 5、9、11 also test and fails, and line 13 will executed, algorithm will call itself recursively, then line 2 and line 3 will be executed. Thus, 8 lines executed, the claim (3+5*1) holds on this case.

Inductive step:

let k be a non-negative integer input such as k≥2

It should be sufficient to use the following:

Inductive Hypothesis:

Let "i" be an input integer such as 0≤ "i" ≤k. Assume that T_{get}("l") ≤3+5*"i".

To proof the following claim:

Inductive Claim:

let input equal to k+1, we will prove that $T_{get}(k+1)=3+5*(k+1)=8+5k$

Proof:

$T_{get}(k+1) \leq (k+1)*T_{get}(1)-(k-1+1)*T_{get}(0)$

$\leq (k+1)*(3+5*1)-k*(3+5*0)$ (By inductive hypothesis, because k≥2, k+1≥3, thus 1 and 0 satisfies the value which is greater or qual to zero and less or equal to k.)

$\leq 8k+8-3k$

$\leq k+8$

Since the result equal to the claim, so we can proof the upper bound is true.

8. First case:

   Suppose key is a non-null input with the type E. And we should insert the key into an empty 2-3 tree. And suppose empty 2-3 tree "T" satisfies all properties of 2-3 tree. Because this 2-3 tree's root is null, thus its set does not include the "key" before. Thus, "T" will create a new node, as the root with this 2-3 tree. Also, after adding the value "k", this tree "T" does not contradict with the post-condition (a) and (b).

   Second case:

   Suppose Tree "T" satisfies all the properties of 2-3 trees, and "T" has only an element in its root. And we will add one element in "T" 's set. It can have two cases:

   Case1:

   The element "key" with type E we will insert will as same as the element that already exist in its root. So as the post-condition said, the set will not be changed and an ElementFoundException is thrown.

   Case2:

   If element "key" with type E is greater than the element which saved in the note, then "Key" will be stored in the right of the existing element. Otherwise, "key" will be stored in the left of the existing element.

9. Suppose tree “T” has the properties of Modified Tree. After called the insertIntoSubtree method, then “T” ’s other subtree will have either 2 or 3 child---by the property (b). By view other properties of Modified Tree, we can see that expect this very internal node which has four child, other subtree has exactly same properties as the 2-3 tree. Which means if we make “4 children node" back to 2-3 tree’s internal node, then we can make the Modified Tree back to the 2-3 tree. First, we take the largest element and third largest from the “four children node”, that is it has 2 elements now. That is, if we have m1, m2, m3, m4, these four elements in this very node, and m1<m2<m3<m4, then we take m2 and m4. Then we create a new internal node “H” as “four children node”’s father node (now it has 2 elements inside), “H” has two element which is m2 and m4. Splitting “H” so we can have “m1, m2” and “m3, m4” these two elements inside each of them. And the tree has restored to be a 2-3 tree.

10. From the question we know that the input key and a node x that is a leaf in T. In this sentence we can know that the x node is at the bottom of the tree and there are no other children of x.
    Then, we are tracing the code. Line 1 will be passed and checked because x is a leaf. Then, go to the line 2, set e to the be the element of E stored at x.
    Then, it will execute the line3 test. There are two cases in this test. If e is equal to the input, throw an ElementFoundException. Another case, if e is not equal to the input key, throw a NoSuchElementException.
    Finally execute the line 19. Then finish the code which is satisfied the postcondition for “Insertion into Subtree of This 2-3 Tree” problem. Then the postcondition of the problem was satisfied and the algorithm will eventually terminate.

11. From the question, we know the x is internal node whose children are leaves as input.
We suppose there are two cases in this question.
Case 1:
Internal x node has two children.
For this case, the test of line 1 checked and failed. Then, it will into try catch sentence. In line 8(first case)
The line 8 checked and passed (We supposed the Internal x node has two children). Then, there are also two different cases.
1) if the input key less or equal to the x.firstmax, then it will call itself. Test at line 1 will checked and passed. Set e to the be the element of E stored at x. Then, there are two cases in this step, there are e is equal to the input key or not equal to the input key. When is equal, throw an ElementFoundException, if not, throw a NoSuchElementException.
2) if the input key larger than x.firstmax. The test at line 9 will checked and failed. It will go into the line 11. Then it will call itself. Test at line 1 will checked and passed. Set e to the be the element of E stored at x. Cause the k is not equal to firstmax. Test at line 3 will checked and failed. It will execute the line 5.
Case 2:
Internal x node does not have two children, the test at line 9 checked but failed, then go into the line 12. Test(line 12) will check and pass. Now we will analysis the three different cases.
1)If key is less than or equal to x.firstmax .Then the test at line 12 is passed, and the execute line 13 which will call itself. (Key, x.firstchild)as input. Cause the children all have leaves. The test at line 1 will pass. There are two different cases in this part. When e is equal or not equal to input key. If e is equal to the input key. From question 10, an ElementFoundException will throw. The exception will not be caught at the line 18. It means that the key belongs to the subset of E stored at leaves of subtree T with root x, and an ElementFoundException was thrown, it satisfies the postcondition of the “Insertion into Subtree of This 2-3 Tree” problem (part a). If e is not equal to the input key. Trow a NoSuchElementException. This exception will be caught at line 18, and addLeaf (key, x) is executed on line 19. Since e is not equal to the input key, it means that the key does not belongs to the subset of E stored at leaves of subtree T with root x. Then, a new leaf is added to x, it satisfies of the postcondition of the “Insertion into Subtree of This 2-3 Tree” problem (part C).
2) if key larger than x.firstmax. Then the test at line 9 is failed, the test at line14 is checked and passed. It will call itself when reach the line 15. Key and x.secondchild will as input. Back to the test 1, it will pass, and e is set to be the element of E stored at x. Then line 3 will be reached and we have two cases. If e is equal to the input key, throw an ElementFoundException. This question will not be caught on line 18. It satisfies the postcondition of the “Insertion into Subtree of This 2-3 Tree” problem (part a). If e is not equal to the input key. Based on the question, a NoSuchElementException will be throw. This exception will be caught on line 18, and addLeaf (key, x) execute. It means that the key does not belongs to the subset of E stored at leaves of subtree T with root x. Then, a new

leaf is added to x, it satisfies of the postcondition of the “Insertion into Subtree of This 2-3 Tree” problem (part c).

3) If key is larger than both x.firstmax and x.secondmax. Then the test at line 10 and line 14 failed, but line 15 pass. insertIntoSubtree(key, x.thirdchild) will executed. (key, x.thirdchild) as input by call itself. x’s children are all leaves, the test 1 will pass, e is set to be the element of E stored at x. Then line 3will be reached and we have two cases. If e is equal to the input key, throw an ElementFoundException. This question will not be caught on line 18. It satisfies the postcondition of the “Insertion into Subtree of This 2-3 Tree” problem (part a). If e is not equal to the input key. Based on the question, a NoSuchElementException will be throw. This exception will be caught on line 18, and addLeaf(key, x) execute. It means that the key is not belonging to the subset of E stored at leaves of subtree T with root x. Then, a new leaf is added to x, it satisfies of the postcondition of the “Insertion into Subtree of This 2-3 Tree” problem (part c).

Finally, if the “Insertion into Subtree of This 2-3 Tree” problem is satisfied and the insertIntoSubtree method is executed with the input key and an internal node x whose children are leaves, then the postcondition of the problem was satisfied and the algorithm will eventually terminate.

12. Suppose p belongs to integer, p ≥ 0, and p is the depth of a 2-3 tree T with root x.

We will use mathematic induction to prove.

Base case:

For p = 0, the algorithm with 2-3 tree with depth p as input is executed. Then we can know the definition of 2-3 tree, node x is a leaf, so the test on line 1 will check and pass. Then, there are two cases. If e is equal to the input key, then it will throw an ElementFoundException (from question 10). If e is not equal to the input key, It will throw a NoSuchElementException(will be caught at line 18). Finally, the algorithm will terminate with postcondition satisfied.

For p = 1, the algorithm with 2-3 tree with depth p as input is executed. We can know that the x is an internal node with leaf as its children. Next, the test at line 1 will check but fail. Then it will reach to the line 7. There are two cases.

1)x has two children

2)x has not two children.

From the question 11, we have already analysis these two cases. Both cases in this algorithm will terminate with postcondition satisfied.

Inductive hypothesis:

Let K and m belong to integer. K ≥1. Suppose that 0≤m≤k, if this algorithm is executed with 2-3 tree of depth m with root x and a key with type E as input, then the algorithm will eventually terminate, with the postcondition of the "Insertion into Subtree of This 2-3 Tree" problem is satisfied.

Inductive Claim:

Suppose this algorithm is executed with 2-3 tree of depth(P) K+1 with root x and a key as input. Then the algorithm will terminate, with the postcondition of the "Insertion into Subtree of This 2-3 Tree" problem is satisfied.

Proof:

We know k≥1, thus k+1 ≥2, and x is an internal node. The test at line 1 check but fail, and the the line 6 and line 7 will reach. There are two cases

1) case one:

x has two children. key is less than or equal to x.firstmax, or key is larger than x.firstmax. The depth of x's children is k, the children of x have a depth of 1 less than x, and the depth of the x's children is k. insertIntoSubtree(key, x.firstChild) or insertIntoSubtree(key, x.secondChild) will terminate with the postcondition satisfied. After that, line 17 will be reached, and the raiseSurplus(x) will execute, and it satisfy the postcondition of the problem.

2)case two:

x does not have two children. Then there are 3 more cases, (1) key is less than or equal to x.firstmax. (2) key is not equal to x.secondmax. (3)key is larger than x.firstmax and key is larger than x.secondmax. The children of x have a depth of 1 less than x, so the children of x have depth of k. The insertIntoSubtree (key, x.firsthilcd), insertIntoSubtree(key, x.secondchild), or insertIntoSubtree(key, x.thirdchild)will

terminate with the postcondition satisfied. Then, line 17 will reach, and the raiseSurplus(x) will execute and it satisfy the postcondition of the problem.

Based on the Strong Form of Mathematical Induction, when the postcondition satisfied, the algorithm will terminate. Thus, we can say the insertIntoSubtree algorithm correctly solves the "Insertion into Subtree of This 2-3 Tree" problem.

13. To describe how to prove that the number of steps executed by this method is in O(depth(x)) where x is the node given as input. First, we know that the depth of the subtree with root x is a bound function for the recursive algorithm insertIntoSubtree. If we can consider raiseSurplus(x) and addLeaf(key,x) as one step. Thus, if when k= 0, which means the depth is 0 and with one node, so that we have constant steps for T(k); if when k≥1, the line6 will execute, and we have condition with two children and three children, then we have T(k-1)+a steps.
So,

$$T(k) \leq \begin{cases} b & \text{if } k=0 \\ T(k-1)+a & \text{if } k\geq 1 \end{cases}$$

a and b are both constants.

14. First, we will search the element that we will delete. The reason we will do search is we need to see this element exist or not, if it doesn't exist, then a NoSuchElementException will be Thrown. If the element does exist, then we will discuss difference cases:

Suppose the tree is called "T", the element we will delete is called "E", the parent of "E" is called "P", the parent of "P" is called "G"(Grandparent of "E")

Case one:

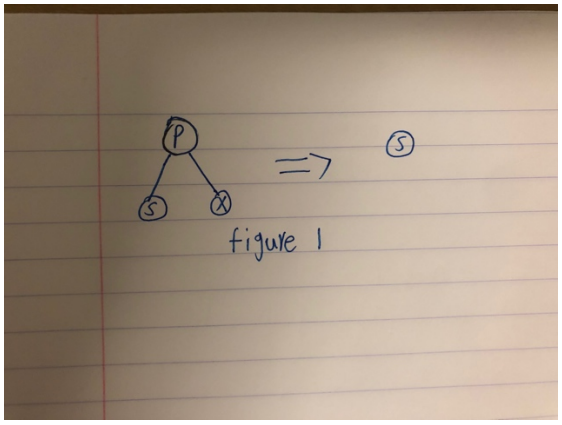
If element "E" is a root, then we will delete "E" directly.

Case two:

If "P" has three children. Then we can remove "E" directly.

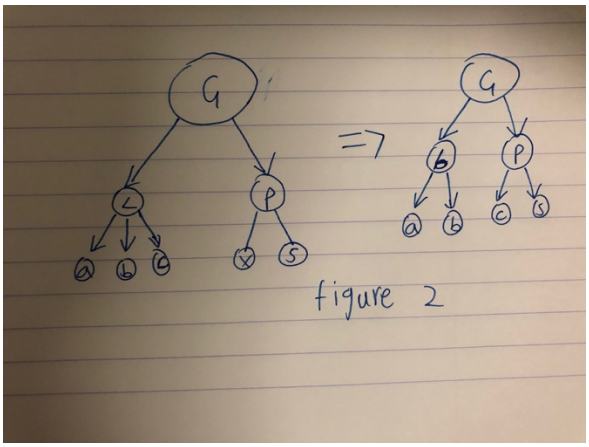
Case two:

If "P" is the root, and "P" has two children, then we delete "P" and "X", leave the other element. ("S" is another child of "P")



Case three:

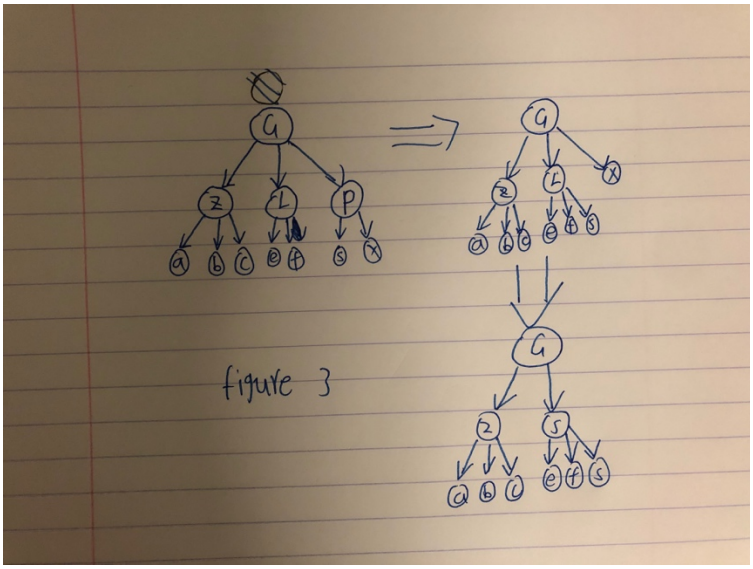
If "P" has 2 children and "P" is not the root. If "P" is the left child of "G", then let "L" be the right sibling of "P", If in this cases "L" has three children. Then we can delete the "X", and move the largest element from "L" to the "P".



Case four:

If "P" has 2 children and "P" is not the root. If "P" is the left child of "G", then let "L" be

the right sibling of “P”, and “X” is the left child of “P”. If in this cases “L” has two children. Then we remove “X”, combine “L” and “P”. Move “S” from “P” to the “L”. Then rename “P” to “X” and recursively remove “X” (*ideas comes from Pro. Rodger, Duke University, CPS 100E: 2-3 Tree https://www2.cs.duke.edu/courses/cps100e/spring99/lects/sect1623treeH.pdf*)



15. See the Java file. The code provided here is copy from the Java file, which is handed in the D2L.

```
//Sijia Yin 30049836; Chunyu Li 30056553; Boxiao Li 30069613
package cpsc331.assignment2;
import java.util.NoSuchElementException;
import cpsc331.collections.ElementFoundException;
import java.util.*;
//These codes for cpsc331 assignment2.
//Author: Sijia Yin 30049836; Chunyu Li 30056553; Boxiao Li 30069613
/**
 * Provides a 2-3 Tree Storing Values from an Ordered Type E.
 */
// 2-3 Tree Invariant: A rooted tree T is represented, so that the
// following 2-3 Tree Properties are satisfied:
// a) Each leaf in T stores an element of type E, and the elements stored at the leaves are distinct.
// b) Each internal node in T has either (exactly) two or three children - which are either leaves or internal
nodes of T.
// c) If an internal node x of T has exactly two children - a first child and a second child, then every
element of E stored at a leaf in the subtree whose root is the first child is less than every element of E
stored at a leaf in the subtree whose root is the second child.
// d) If an internal node x of T has exactly three children - a first child, second child and third child,
then every element of E stored at a leaf in the subtree whose root is the first child is less than every
element of E stored at a leaf in the subtree whose root is the second child, and every element of E stored at
a leaf in the subtree whose root is the second child is less than every element of E stored at a leaf in the
subtree whose root is the third child.
// e) If an internal node x has exactly two children then the largest elements stored in each of the subtrees
whose roots are its children are also stored at x (and are called firstMax and secondMax).
// f) If an internal node x has exactly three children then the largest elements stored in each of the
subtrees whose roots are its children are also stored at x (and are called firstMax, secondMax and thirdMax).
// g) Every leaf in T is at the same level, that is, has the same distance from the root of T.
// h) Each node in T is the root of a 2-3 tree as well. That is, the subtree of T with root x also satisfies
properties (a)-(g).
public class TwoThreeTree<E extends Comparable<E>>{
   // Provides a node in this 2-3 Tree
   class TwoThreeNode{
       // Data Fields
       int numberChildren; // Number of children of this node; an integer between 0 and 4
       E element; // Element stored at this node; null if this is not a leaf
       TwoThreeNode firstChild; // First child
       E firstMax;           // Largest element stored in first subtree
                             // Both are null if this node is a leaf
       TwoThreeNode secondChild; // Second child
```

```
E secondMax;           // Largest element stored in second subtree
                       // Both are null if this node has at mostone child
TwoThreeNode thirdChild; // Third child
E thirdMax;           // Largest element stored in third subtree
                      // Both are null if this node has at most two children
TwoThreeNode fourthChild; // Fourth child
E fourthMax;           // Largest element stored in fourth subtree
                       // Both are null if this node has at most three children
TwoThreeNode parent; // Parent; null if this node is the root of this tree
// Constructor; constructs a TwoTreeNode with no children or parent, storing null
TwoThreeNode(){
   numberChildren = 0;
   element = null;
   firstChild = null;
   firstMax = null;
   secondChild = null;
   secondMax = null;
   thirdChild = null;
   thirdMax = null;
   fourthChild = null;
   fourthMax = null;
   parent = null;
}
void printNode(){
   System.out.println("\nSTART PRINT NODE: ----");
   System.out.println("pointer " + this);
   System.out.println("parent " + parent);
   System.out.println("number of children: " + numberChildren);
   System.out.println("element: " + element);
   System.out.println("firstmax: " + firstMax);
   System.out.println("firstchild: " + firstChild);
   System.out.println("secondmax: " + secondMax);
   System.out.println("secondchild: " + secondChild);
   System.out.println("thirdmax: " + thirdMax);
   System.out.println("thirdchild: " + thirdChild);
   System.out.println("fourthmax: " + fourthMax);
   System.out.println("fourtgchild: " + fourthChild);
   System.out.println("END PRINT NODE: ----");
}
// Returns the number of children of this node
int numberChildren(){
```

```
            return numberChildren;
        }
        // Returns the element stored at this node if it is a leaf; returns null otherwise.
        E element(){
            return element;
        }
        // Returns the first child of this node if it is not a leaf; returns null otherwise
        TwoThreeNode firstChild(){
            return firstChild;
        }
        // Returns the largest value stored at the first subtree of this node if it is not a leaf; returns null
otherwise
        E firstMax(){
            return firstMax;
        }
        // Returns the second child of this node if it has at least two children; returns null otherwise
        TwoThreeNode secondChild(){
            return secondChild;
        }
        // Returns the largest value stored at the first subtree of this node if it has at least two children;
returns null otherwise
        E secondMax(){
            return secondMax;
        }
        // Returns the third child of this node if it has at least three children; returns null otherwise
        TwoThreeNode thirdChild(){
            return thirdChild;
        }
        // Returns the largest value stored at the third subtree of this node if it has at least four children;
returns null otherwise
        E thirdMax(){
            return thirdMax;
        }
        // Returns the fourth child of this node if it has four children; returns null otherwise
        TwoThreeNode fourthChild(){
            return fourthChild;
        }
        // Returns the largest value stored at the fourth subtree of this node if it has four chilren; returns
null otherwise
        E fourthMax(){
            return fourthMax;
```

```
    }
    // Returns the parent of this node
    TwoThreeNode parent(){
        return parent;
    }
}
// Data Fields
private TwoThreeNode root;
/**
 * Constructs an empty 2-3 Tree.
 */
// Precondition: None
// Postcondition: An empty 2-3 Tree (satisfying the above 2-3 Tree Invariant) has been created.
public TwoThreeTree(){
    root = null;
}
// ****************************************************************
//   Searching in a 2-3 Tree
// ****************************************************************
public void printTree(){
    printTreeR(root);
}
public void printTreeR(TwoThreeNode n) {
    Queue<TwoThreeNode> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()) {
        TwoThreeNode tmp = q.remove();
        if (tmp != null)
            tmp.printNode();
        else {
            System.out.println("\nnull node");
            continue;
        }
        q.add(tmp.firstChild);
        q.add(tmp.secondChild);
        q.add(tmp.thirdChild);
        q.add(tmp.fourthChild);
    }
}
/**
 * Returns a TwoThreeNode with a given key<br>
```

```
     * @param key the element to be searched for
     * @return the TwoThreeNode in this 2-3 tree storing the input key
     * @throws NoSuchElementException if the key is not in this tree
     */
    // Precondition::
    // a) This 2-3 Tree satisfies the above 2-3 Tree Properties.
    // b) A non-null key with type E is given as input.
    // Postcondition:
    // a) If the key is stored in this 2-3 tree then the node storing it isreturned as output. A
NoSuchElementException is thrown, otherwise.
    // b) This 2-3 Tree has not been changed, so it still satisfiesthe 2-3 Tree Properties.
    public TwoThreeNode search(E key) throws NoSuchElementException{
        if (root == null) {
            throw new NoSuchElementException();
        } else {
            return get(key, root);
        }
    }
    // Searches for a given key in the subtree with a given node as root
    // Precondition:
    // a) This 2-3 tree satisfies the above 2-3 Tree Properties.
    // b) key is a non-null input with type E.
    // c) x is a non-null TwoThreeNode in this 2-3 Tree, that isalso given as input.
    // Postcondition:
    // a) If the key is stored in the subtree with root x, then the nodestoring the key is returned as output.
A NoSuchElementExceptions thrown otherwise.
    // b) This 2-3 Tree has not been changed, so it still satisfies the 2-3 Tree Properties.
    private TwoThreeNode get(E key, TwoThreeNode x) throws NoSuchElementException{
        if (x.element() != null) { // if x is a leaf
            if (key.compareTo(x.element())== 0) { // if key is equal to element stored at x
                return x;
            } else {
                throw new NoSuchElementException();
            }
        } else if (x.numberChildren() == 2) { // if x has 2 children
            if (key.compareTo(x.firstMax()) <= 0) {
                return get(key, x.firstChild());
            } else {
                return get(key, x.secondChild());
            }
        } else {
```

```
            if (key.compareTo(x.firstMax()) <= 0) {
                return get(key, x.firstChild());
            } else if (key.compareTo(x.secondMax()) <= 0) {
                return get(key, x.secondChild());
            } else {
                return get(key, x.thirdChild());
            }
        }
    }
    // ***********************************************************************
    //  Insertions in a 2-3 Tree
    // ***********************************************************************
    // The following "Modified Tree" properties are satisfied at the beginning or end of private methods called by public ones provided  by this 2-3 tree.
    // a) This tree, T, satisfies 2-3 Tree properties (a), (c), (d), (e), (f), and (g) - but not, necessarily, 2-3 Tree properties (b) or (h).
    // b) Every internal node of T has (exactly) either one, two, three or four children - which are each either leaves or internal nodes of T. There is at most one internal node of T that has (exactly) either one or four children - all other internal nodes of T have either exactly two or three children.
    // c) If an internal node x of T has exactly one child, then this is called a first child, which is the root of a first subtree of the subtree with root x. In this case, the largest element of E stored in a leaf of the first subtree is stored at x, as the value of x.firstMax.
    // d) If an internal node x of T has four children - a first child, second child, third child and fourth child, which are the roots of the first subtree, second subtree, third subtree and fourth subtree of the subtree of T with root x, respectively - then each of the values stored at leaves of the first subtree is less than each of the values stored at leaves of the second subtree, each of the values stored at leaves of the second subtree is less than each of the values stored at leaves of the third subtree, and each of the values stored at leaves of the third subtree is less than values stored at leaves of the fourth subtree.
    // e) If an internal node x of T has exactly four children then the largest values stored at the leaves of the first, second, third and fourth subtrees of the subtree with root x are stored at x - and are called firstMax, secondMax, thirdMax and fourthMax, respectively.
    // f) Each node x of T is the root of a rooted tree satisfying these properties as well. That is, the subtree with root x also satisfies the above properties (a)-(e), for eery node x in T.
    /**
     * Inserts an input key into this 2-3 Tree
     * @param key the key to be inserted into this 2-3 Tree
     * @throws ElementFoundException if the input key is already stored in this tree
     */
    // Precondition:
    // a) This 2-3 Tree, T, satisfies the above 2-3 Tree Properties.
    // b) key is a non-null input of type E.
```

```
    // Postcondition:
    // a) If the input key is not already included in the subset of E represented by T then it is added to this
subset (with this subset being otherwise unchanged). An ElementFoundException is thrown and the set is not
changed, if the key already belongs to this subset.
    // b) T satisfies the 2-3 Tree oroperties given above.
    public void insert(E key) throws ElementFoundException{
        if (root == null) {
            addFirstElement(key);
        } else if (root.element() != null) { // if root is a leaf
            addSecondElement(key);
        } else {
            insertIntoSubtree(key, root);
            if (root.numberChildren() == 4) { // if root has 4 children
                fixRoot();
            }
        }
    }
    // Inserts an input key into this 2-3 Treee when it is Empty
    // Precondition:
    // a) This 2-3 Tree, T, satisfies the above 2-3 tree properties - andis empty.
    // b) key is a non-null input with type E.
    // Postcondition:
    // a) The input key has been added to the subset of E represented by T, which is otherwise unchanged.
    // b) T satisfies the 2-3 Tree properties given above.
    private void addFirstElement(E key) {
        root = new TwoThreeNode();
        root.element = key;
    }
    // Inserts an input key into this 2-3 tree when the root of this tree is a leaf
    // Precondition:
    // a) This 2-3 Tree, T, satisfies the 2-3 Tree properties - and the root of this tree is a leaf, so that T
represents a subset of E with size one.
    // b) key is a non-null input with type E.
    // Postcondition:
    // a) Id the input key does not belong to the subset of E represented by T then they key is added to this
subset - which is otherwise unchanged. If the key does already belong to this subset and an
ElementFoundException is thrown and T is not changed.
    // b) T satisfies the 2-3 Tree properties given above.
    private void addSecondElement(E key) throws ElementFoundException{
        TwoThreeNode temp = new TwoThreeNode();
        TwoThreeNode keyNode = new TwoThreeNode();
```

```
        keyNode.element = key;
        temp.numberChildren = 2; // temp have 2 children
        keyNode.parent = temp;  // setting keyNode's parent as temp
        if (root.element().compareTo(key)== 0) { // if key already contained in the tree
           throw new ElementFoundException(
               "element found inside addSecondElement");
        } else if (root.element().compareTo(key) < 0) { // if root < newKey
           temp.firstMax = root.element();
           temp.firstChild = root; // first child is root
           temp.secondMax = key;
           temp.secondChild = keyNode; // second child is keynode
        } else {                    // if root > newKey
           temp.firstMax = key;
           temp.firstChild = keyNode; // first child is keynode
           temp.secondMax = root.element();
           temp.secondChild = root; // second child is root
        }
        root.parent = temp; // setting root's parent as temp
        root = temp;  // make root point to temp
    }
    // Inserts a given key into the subtree of T with a given node x as root if x is an internal node; throws
an exception to aid the inclusion of the input key if x is a leaf
    // Precondition:
    // a) This 2-3 tree, T, satisfies the 2-3 Tree properties given above.
    // b) key is a non-null input with type E.
    // c) x is a non-null node in T.
    // d) Either key is not stored at any leaf in T or it is stored in a leaf of the subtree of T with root x.
    // Postcondition:
    // a) If the input key already belongs to the subset of E stored at the leaves in the subtree of T with
root x, then an ElementFoundException is thrown and T is not changed.
    // b) If x 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.
    // c) If x is an internal node and the input key does not (initially) belong to the subset of E stored at
the leaves in the subtree of T with root x, then - the input key is added to the subset of E stored at the
leaves of T - which is otherwise unchanged; - either T satisfies the 2-3 Tree properties, given above, or T
satisfies the "Modified Tree" properties, given above, and x is now an internal node with four children.
    private void insertIntoSubtree(E key, TwoThreeNode x) throws ElementFoundException, NoSuchElementException{
        if (x.element() != null) { // if x is a leaf
           E e = x.element();
           if (e.compareTo(key) == 0) {
               throw new ElementFoundException(
```

```
                "element found in insertIntoSubtree");
          } else {
              throw new NoSuchElementException();
          }
       } else {
          try {
              if (x.numberChildren() == 2) { // if x has two children
                 if (key.compareTo(x.firstMax()) <= 0) {
                     insertIntoSubtree(key, x.firstChild);
                 } else {
                     insertIntoSubtree(key, x.secondChild);
                 }
              } else {
                 if (key.compareTo(x.firstMax()) <= 0) {
                     insertIntoSubtree(key, x.firstChild);
                 } else if (key.compareTo(x.secondMax()) <= 0) {
                     insertIntoSubtree(key, x.secondChild);
                 } else {
                     insertIntoSubtree(key, x.thirdChild);
                 }
              }
              raiseSurplus(x);
          } catch (NoSuchElementException ex) {
              addLeaf(key, x);
          }
       }
   }
   // Brings a node with four children closer to the root, if one exists in this modified tree
   // Precondition:
   // a) This tree, T, satisfies the "Modified Tree" properties given above.
   // b) x is an internal node of T whose children are also internal nodes in T.
   // c) Either T is a 2-3 tree (that is, it satisfies the above "2-3 Tree" properties), or one of the
children of x has four children.
   // Postcondition:
   // a) The subset of E stored at the leaves of T has not been changed.
   // b) Either T is a 2-3 tree (that is, it satisfies the above "2-3 Tree" properties), or T satisfies the
"Modified Tree" properties and x has four children.
   private void raiseSurplus(TwoThreeNode x) {
       ArrayList<TwoThreeNode> children = new ArrayList<TwoThreeNode>() {
          {
              add(x.firstChild());
```

```
            add(x.secondChild());
            add(x.thirdChild());
            add(x.fourthChild());
        }
    }; // stores the children of x
    TwoThreeNode fourChildrenNode = new TwoThreeNode();
    boolean hasFourChildren = false;
    int i = 0, indexOfFourChild = 0;
    while (i < children.size() && children.get(i) != null) {
        if (children.get(i).numberChildren() == 4) {
            hasFourChildren = true; // hasFourChildren is true
            fourChildrenNode =
                children.get(i); // fourChildrenNode have same pointer value
                                 // as children.get(i)A
            indexOfFourChild = i;
            break;
        }
        ++i;
    }
    if (hasFourChildren) { // if a child of x has 4 children
        TwoThreeNode tempNodeLeft = new TwoThreeNode();
        TwoThreeNode tempNodeRight = new TwoThreeNode();
        tempNodeLeft.numberChildren = 2;
        tempNodeLeft.parent = x; // tempNodeLeft's parent is x
        tempNodeRight.numberChildren = 2;
        tempNodeRight.parent = x; // tempNodeRight's parent is x
        tempNodeLeft.firstChild = fourChildrenNode.firstChild();
        tempNodeLeft.firstMax = fourChildrenNode.firstMax();
        fourChildrenNode.firstChild().parent = tempNodeLeft;
        tempNodeLeft.secondChild = fourChildrenNode.secondChild();
        tempNodeLeft.secondMax = fourChildrenNode.secondMax();
        fourChildrenNode.secondChild().parent = tempNodeLeft;
        tempNodeRight.firstChild = fourChildrenNode.thirdChild();
        tempNodeRight.firstMax = fourChildrenNode.thirdMax();
        fourChildrenNode.thirdChild().parent = tempNodeRight;
        tempNodeRight.secondChild = fourChildrenNode.fourthChild();
        tempNodeRight.secondMax = fourChildrenNode.fourthMax();
        fourChildrenNode.fourthChild().parent = tempNodeRight;
        x.numberChildren++;
        // modify the array
        // children.remove(indexOfFourChild);
```

```
            children.set(indexOfFourChild, tempNodeLeft);
            children.add(indexOfFourChild + 1, tempNodeRight);
            x.firstChild = children.get(0);
            x.firstMax = getMaxFromNode(children.get(0));
            x.secondChild = children.get(1);
            x.secondMax = getMaxFromNode(children.get(1));
            x.thirdChild = children.get(2);
            x.thirdMax = getMaxFromNode(children.get(2));
            x.fourthChild = children.get(3);
            x.fourthMax = getMaxFromNode(children.get(3));
        } else {
            x.firstChild = children.get(0);
            x.firstMax = getMaxFromNode(children.get(0));
            x.secondChild = children.get(1);
            x.secondMax = getMaxFromNode(children.get(1));
            x.thirdChild = children.get(2);
            x.thirdMax = getMaxFromNode(children.get(2));
            x.fourthChild = children.get(3);
            x.fourthMax = getMaxFromNode(children.get(3));
        }
    }
    private E getMaxFromNode(TwoThreeNode node) {
        if (node == null) {
            return null;
        }
        if (node.element() != null) {
            return node.element();
        }
        if (node.numberChildren == 1) {
            return node.firstMax();
        } else if (node.numberChildren == 2) {
            return node.secondMax();
        } else if (node.numberChildren == 3) {
            return node.thirdMax();
        } else { // node has 4 children
            return node.fourthMax();
        }
    }
    // Adds a leaf storing a given value as a child of a given internal node
    // Precondition:
    // a) This tree, T,is a 2-3 tree (that is,it satisfies the 2-3 Tree properties given above).
```

```
   // b) x is an input internal node in T whose children are leaves.
   // c) key is a non-null input element of E that is not in the set of elements of E stored at leaves of T.
   // d) It is possible to produce a tree satisfying the "Modified Tree" properties, given above, by adding a
leaf storing the input key as a child of x.
   // Postcondition:
   // a) The input key has been added to the set of elements stored at the leaves of T, which is otherwise
unchanged.
   // b) Either T is a 2-3 Tree or T satisfies the above "Modified Tree" properties and x has four children.
   private void addLeaf(E key, TwoThreeNode x){
       TwoThreeNode keyNode = new TwoThreeNode();
       keyNode.parent = x;
       keyNode.element = key;
       ArrayList<TwoThreeNode> children = new ArrayList<TwoThreeNode>() {
          {
              add(x.firstChild());
              add(x.secondChild());
              add(x.thirdChild());
              add(x.fourthChild());
          }
       }; // stores the children of x
       boolean foundLarge = false;
       int i = 0, indexOfKeyInsertion = 0;
       while (children.get(i) != null  && (children.get(i).element().compareTo(key) <= 0)) {
          ++i;
       }
       indexOfKeyInsertion = i;
       children.add(indexOfKeyInsertion, keyNode); // insert
       x.numberChildren++;
       x.firstChild = children.get(0);
       x.firstMax = getMaxFromNode(children.get(0));
       x.secondChild = children.get(1);
       x.secondMax = getMaxFromNode(children.get(1));
       x.thirdChild = children.get(2);
       x.thirdMax = getMaxFromNode(children.get(2));
       x.fourthChild = children.get(3);
       x.fourthMax = getMaxFromNode(children.get(3));
   }
   // Completes the restoration of a 2-3 tree after the "insertIntoSubtree" method has applied and the root
has four children
   // Precondition:
   // a) T is a rooted tree, satisfying the above "Modified Tree" properties, whose root is an internal node
```

```
with four children.
    // Postcondition:
    // a) The subset of E represented by T has not been changed.
    // b) T now satisfies the "2-3 Tree" properties given above.
    private void fixRoot(){
        TwoThreeNode fstChildOfRoot = new TwoThreeNode();
        TwoThreeNode sndChildOfRoot = new TwoThreeNode();
        TwoThreeNode newRoot = new TwoThreeNode();
        fstChildOfRoot.parent = newRoot;
        fstChildOfRoot.numberChildren = 2;
        sndChildOfRoot.parent = newRoot;
        sndChildOfRoot.numberChildren = 2;
        newRoot.numberChildren = 2;
        fstChildOfRoot.firstChild = root.firstChild();
        fstChildOfRoot.firstMax = root.firstMax();
        root.firstChild().parent = fstChildOfRoot;
        fstChildOfRoot.secondChild = root.secondChild();
        fstChildOfRoot.secondMax = root.secondMax();
        root.secondChild().parent = fstChildOfRoot;
        sndChildOfRoot.firstChild = root.thirdChild();
        sndChildOfRoot.firstMax = root.thirdMax();
        root.thirdChild().parent = sndChildOfRoot;
        sndChildOfRoot.secondChild = root.fourthChild();
        sndChildOfRoot.secondMax = root.fourthMax();
        root.fourthChild().parent = sndChildOfRoot;
        newRoot.firstChild = fstChildOfRoot;
        newRoot.firstMax = fstChildOfRoot.secondMax();
        newRoot.secondChild = sndChildOfRoot;
        newRoot.secondMax = sndChildOfRoot.secondMax();
        root = newRoot;
    }
    // ********************************************************************
    //   Deletions from a 2-3 Tree
    // ********************************************************************
    /**
     * Removes an input key from this 2-3 Tree
     * @param key the key to be removed from this 2-3 Tree
     * @throws NoSuchElementException if the input key is not already stored in this tree
     */
    // Precondition:
    // a) This 2-3 Tree, T, satisfies the above 2-3 Tree Properties.
```

```
   // b) key is a non-null input of type E.
   // Postcondition:
   // a) If the input key is included in the subset of E represented by T then it is removed from this subset
(with this subset being otherwise unchanged). A NoSuchElementException is thrown and the set is not changed,
if the key already belongs to this subset.
   // b) T satisfies the 2-3 Tree oroperties given above.
   // took ideas from this website: https://www2.cs.duke.edu/courses/cps100e/spring99/lects/sect1623treeH.pdf
   public void delete(E key)throws NoSuchElementException {
       TwoThreeNode leafPos = search(key);
       // if key not found search method should throw no usch element exception
       deleteNode(leafPos, leafPos.parent());
   }
   public void fixMax(TwoThreeNode p) { should fix root too
       while (p != null) {
          p.firstMax = getMaxFromNode(p.firstChild);
          p.secondMax = getMaxFromNode(p.secondChild);
          p.thirdMax = getMaxFromNode(p.thirdChild);
          p = p.parent;
       }
   }
   // will start at the leaf's parent and delete
   public void deleteNode(TwoThreeNode nodeToDelete, TwoThreeNode p) {
       if (root.element() != null) { // if root is a leaf
          if (root.element().compareTo(nodeToDelete.element())== 0) {     // if root = leaf
              root = null; // delete root
          } else {
              throw new NoSuchElementException();
          }
       } else if (hasThreeChild(p)) { // if p have 3 child
          // first time should have 3 leaf child
          ArrayList<TwoThreeNode> children = new ArrayList<TwoThreeNode>() {
              {
                 add(p.firstChild());
                 add(p.secondChild());
                 add(p.thirdChild());
              }
          }; // stores the children of p
          int i = 0;
          // while the child is not the nodeToDelete
          while (i < children.size() && children.get(i) != nodeToDelete) {
              ++i;
```

```
            }
            // when loop finish it should have found the key to delete
            children.remove(i);
            // p now have 2 children, and the algorithm will terminate
            p.firstChild = children.get(0);
            p.firstMax = getMaxFromNode(children.get(0));
            p.secondChild = children.get(1);
            p.secondMax = getMaxFromNode(children.get(1));
            p.thirdChild = null;
            p.thirdMax = null;
            nodeToDelete.parent = null;
            nodeToDelete = null;
            p.numberChildren = 2;
            // fix max of p
            fixMax(p);
        } else { // p have 2 child
            // first time should have 3 leaf child
            // if p is root
            if (p == root) { // can compare pointers here..
                TwoThreeNode newRoot = new TwoThreeNode();
                if (p.firstChild() == nodeToDelete) { // if remove first child
                    newRoot = p.secondChild();
                } else {
                    newRoot = p.firstChild(); // if remove second child
                }
                // new root with the other child as root
                newRoot.parent = null;
                root = newRoot;
            } else {
                // note that not all these siblings can exist, but one of them
                // must exist
                TwoThreeNode leftSiblingOfP = new TwoThreeNode();
                TwoThreeNode rightSiblingOfP = new TwoThreeNode();
                leftSiblingOfP = leftSibling(p);
                rightSiblingOfP = rightSibling(p);
                if (leftSiblingOfP != null && leftSiblingOfP.numberChildren()== 3) { // if left sibling opf p
exist and it have 3 children
                    // just move left sib child to p
                    if (p.firstChild()== nodeToDelete) { // if want delete first child do nothing
                    } else {            // want delete second child
                        p.secondChild = p.firstChild();
```

```
                    p.secondMax =p.firstMax; // second max = first max, since first max will be replaced
(get smaller)
                }
                // want shift 3rd child of left sib into 1st child
                leftSiblingOfP.thirdChild().parent = p;
                leftSiblingOfP.numberChildren = 2;
                // put left sib into p
                p.firstChild = leftSiblingOfP.thirdChild();
                p.firstMax = leftSiblingOfP.thirdMax();
                // delete from left sib
                leftSiblingOfP.thirdChild = null;
                leftSiblingOfP.thirdMax = null;
                // fix max of p
                fixMax(p);
            } else if (rightSiblingOfP != null && rightSiblingOfP.numberChildren() == 3) { // if right
sibling opf p exist and it have 3 children
                // just move right sib child to p
                if (p.secondChild() == nodeToDelete) { // if want delete second childo nothing
                } else {            // want delete first child
                    // make first child as second child
                    p.firstChild = p.secondChild();
                    p.firstMax = p.secondMax; // since second max will be eplaced (get bigger)
                }
                // move 1st child of right sib into second child of p
                rightSiblingOfP.firstChild().parent = p;
                rightSiblingOfP.numberChildren = 2;
                // put right sib into p
                p.secondChild = rightSiblingOfP.firstChild();
                p.secondMax = rightSiblingOfP.firstMax();
                // move 2nd to 1st, 3rd to 2nd, and make 3rd null......
                rightSiblingOfP.firstChild = rightSiblingOfP.secondChild();
                rightSiblingOfP.firstMax = rightSiblingOfP.secondMax();
                rightSiblingOfP.secondChild = rightSiblingOfP.thirdChild();
                rightSiblingOfP.secondMax = rightSiblingOfP.thirdMax();
                rightSiblingOfP.thirdChild = null;
                rightSiblingOfP.thirdMax = null;
                // fix max of p
                fixMax(p);
            } else {
                // hard case, left and/or right sibling have 2 children
                if (leftSiblingOfP != null) { // if left sib is not null
```

```
            // move the remaining child in p to left
            TwoThreeNode temp = new TwoThreeNode();
            if (p.firstChild() = nodeToDelete) { // if want delete first child
               temp = p.secondChild();
            } else { // want delete second child
               temp = p.firstChild();
            }
            p.secondChild = null; // delete second child
            p.secondMax = null;
            p.firstChild = null; // delete first child
            p.firstMax = null;
            temp.parent = leftSiblingOfP; // change temp's parent
            leftSiblingOfP.numberChildren = 3;
            p.numberChildren = 0;
            // transfer max
            leftSiblingOfP.thirdChild = temp;
            leftSiblingOfP.thirdMax = getMaxFromNode(temp);
            // mark p to be the deleted node, and recursivly delete
            deleteNode(p, p.parent());
            // fix max of p
            fixMax(p);
        }else { // right sib is not null
            TwoThreeNode temp = new TwoThreeNode();
            if (p.firstChild() == nodeToDelete) { // if want delete first child
               temp = p.secondChild();
            } else { // want delete second child
               temp = p.firstChild();
            }
            p.secondChild = null;
            p.secondMax = null;
            p.firstChild = null;
            p.firstMax = null;
            temp.parent = rightSiblingOfP;
            rightSiblingOfP.numberChildren = 3;
            p.numberChildren = 0;
            // transfer max
            rightSiblingOfP.thirdChild = rightSiblingOfP.secondChild;
            rightSiblingOfP.thirdMax = rightSiblingOfP.secondMax;
            rightSiblingOfP.secondChild = rightSiblingOfP.firstChild;
            rightSiblingOfP.secondMax = rightSiblingOfP.firstMax;
            rightSiblingOfP.firstChild = temp;
```

```
                    rightSiblingOfP.firstMax = getMaxFromNode(temp);
                    // mark p to be the deleted node, and recursivly delete
                    deleteNode(p, p.parent());
                    fixMax(p); // fix max of p
                }
            }
        }
    }
    boolean hasThreeChild(TwoThreeNode x) {
        // if 3 child are not null, and the first child is a leaf
        return x.firstChild() != null && x.secondChild() != null && x.thirdChild() != null;
    }
    TwoThreeNode leftSibling(TwoThreeNode x) {
        // can compare pointers here..
        if (x.parent().secondChild()== x) { // if x is second child of its parent
            return x.parent().firstChild();
        } else if (x.parent().thirdChild() == x) { // if x is third child
            return x.parent().secondChild();
        } else { // if x is first child
            return null;
        }
    }
    TwoThreeNode rightSibling(TwoThreeNode x) {
        // can compare pointers here..
        if (x.parent().secondChild()== x) { // if x is second child of its parent
            return x.parent().thirdChild();
        } else if (x.parent().firstChild() == x) { // if x is first child
            return x.parent().secondChild();
        } else { // if x is third child
            return null;
        }
    }
    // ********************************************************************
    //   Additional Code for Testing
    // ********************************************************************
    // Returns a reference to the root of this 2-3 Tree
    TwoThreeNode root(){
        return root;
    }
}
```