CPSC 331 — Assignment #2

$\mathbf{2}$-$\mathbf{3}$ Trees

About This Assignment

This assignment introduces another data structure, related to a binary search tree, that can be used to provide an efficient implementation of a `Dictionary`. It can be completed by groups of up to three students and is due by 11:59pm on Monday, June 3.

$\mathbf{2}$-$\mathbf{3}$ Trees: Definition

Suppose that E is an ordered type, that is, a nonempty set of values that have a total order. A ***2-3-tree***, for type E, is a finite rooted tree T (like a binary search tree or a red-black tree) that satisfies the following ***2-3 Tree Properties:***

(a) Every ***leaf*** in T stores an element of E. and the elements stored at the leaves of T are *distinct*. Thus T represents a finite ***subset*** of E, namely, the set of values stored in the leaves of T.

(b) Every ***internal node*** of T has (exactly) either two or three children — which are each either leaves or internal nodes of T.

(c) If an internal node x has exactly two children — a ***first child***, which is the root of a ***first subtree*** of the subtree with root x, and a ***second child***, which is the root of a ***second subtree*** of the subtree with root x — then each of the values stored at the leaves of the first subtree is ***less than*** each of the values stored at the leaves of the second subtree.

(d) If an internal node x has exactly three children — a ***first child***, ***second child*** and ***third child***, which are the roots of the ***first subtree***, ***second subtree*** and ***third subtree*** of the subtree with root x, respectively — then each of the values stored at the leaves of the first subtree is ***less than*** each of the values stored at the leaves of the second subtree, and

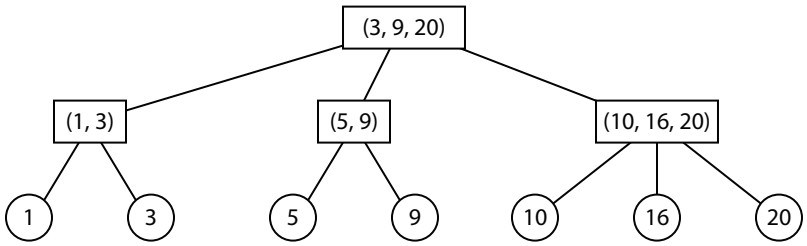

Figure 1: A 2-3 Tree

each of the values stored at the leaves of the second subtree is ***less than*** each of the values stores at the leaves of the third subtree.

(e) If an internal node x of T has exactly two children then x stores a 2-tuple (m_1, m_2) — where m_1 is the largest value stored at any leaf of the *first* subtree, and m_2 is the largest value stored at any leaf of the *second* subtree of the subtree with root x.

(f) If an internal node x of T has exactly three children then x stores a 3-tuple (m_1, m_2, m_3) — 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, and m_3 is the largest value stored at any leaf of the *third* subtree of the subtree with root x.

(g) Every leaf of T is at the same level of the tree — that is, the number of edges from the root down to each leaf is the same.

(h) Each node x of T is 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.

For example, the tree shown in Figure 1 is a 2-3 tree representing the set of integers

$$\{1, 3, 5, 9, 10, 16, 20\}.$$

Exercise: Confirm that this tree satisfies the "2-3 Tree Properties" that have now been given.

Note: While a variety of references (that are available online) describe "2-3 Trees", they *do not* all describe the same kind of tree and, in particular, they do not necessarily describe the kind of tree that is to be considered in this assignment!

It will be useful to measure costs of operations on 2-3 trees in terms of the sizes of the ***sets of elements of*** E that they represent, so that it will be helpful to bound the number of ***internal nodes*** in this way too.

1. Let T be a nonempty 2-3 tree, so that it includes at least one node. Prove that if T represents a subset $S \subseteq \mathtt{E}$ such that $|S| = n \in \mathbb{N}$, then T has at most $n - 1$ internal nodes.

It follows that (if the uniform cost criterion is used to define this) the amount of storage space needed to represent a non-empty 2-3 tree is at most linear in the size of the subset of E that it represents.

The ***depth*** of an empty 2-3 tree will be defined to be -1, just like the depth of an empty binary search tree. Otherwise it is defined to be the number of edges in a longest path from the root to a leaf in T — which is the number of edges in *any* path from the root down to *any* leaf in T, by 2-3 Tree Property (g). Useful relationships between the depth of a 2-3 tree and the size of the subset of E it represents can now be established.

2. Let T be a non-empty 2-3 tree with depth $d \in \mathbb{N}$ and let n be the size of the subset of E represented by T. Prove that $2^d \leq n \leq 3^d$.

3. Let $d \in \mathbb{N}$ and suppose that $|\mathtt{E}| \geq 2^d$. Prove that there exists a non-empty 2-3 tree with depth d, representing a subset of E with size exactly 2^d.

The proof used to complete Problem #3 can be modified to prove that if $d \in \mathbb{N}$ and $|\mathtt{E}| \geq 3^d$ then there exists a non-empty 2-3 tree with depth d, representing a subset of E with size exactly 3^d, as well. Thus the bounds on sizes of a 2-3 tree, given in Problem #1, are *tight*. That is, they cannot be improved.

2-3 Trees: Searches

If a ***search*** algorithm is to be implemented as a method, provided by some 2-3 tree T, then it can be considered as an algorithm that solves the following problem[1].

Searching in This 2-3 Tree

Precondition:

(a) This 2-3 tree, T, satisfies the 2-3 Tree Properties given above.

(b) A non-`null` `key`, of type E, is given as input.

[1] It is assumed here, and in the rest of the assignment, 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.

Postcondition:

(a) If the `key` is stored at a leaf in T then the leaf storing this `key` is returned as output. A `NoSuchElementException` is thrown, otherwise.

(b) This 2-3 tree has not been changed, so it still satisfies the 2-3 Tree Properties.

Consider the following problem (which might also be solved using a method provided by this 2-3 tree) as well.

Searching in a Subtree of This 2-3 Tree

Precondition:

(a) This 2-3 tree, T, satisfies the 2-3 Tree Properties given above.

(b) `key` is a non-`null` input of type `E`.

(c) `x` is a non-`null` in T that is also given as input.

Postcondition:

(a) 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.

(b) This 2-3 tree has not changed, so it still satisfies the 2-3 Tree Properties.

Suppose that `get` is an algorithm that correctly solves the *second* of these problems. Then the algorithm shown in Figure 2 correctly solves the *first* of these problems, assuming that `T.root` is a reference to the root of this 2-3 tree (so that `T.root` is `null` if and only if this is an empty tree).

It is possible to modify a corresponding algorithm, for searching in a subtree of a *binary* tree, to obtain a simple ***recursive*** algorithm that solves the second problem. Since this is a 2-3 tree and the input code `x` is not `null`, such an algorithm is as shown in Figure 3. It is assumed here that if `x` is not a leaf then `x.firstChild` is its first child and `x.firstMax` is the largest element stored in the first subtree. Similarly, if `x` is not a leaf then `x.secondChild` is its second child and `x.secondMax` is the largest element stored in its second subtree — and if `x` has three children then `x.thirdChild` is its third child and `x.thirdMax` is the largest element stored in its third subtree.

4. Prove that the depth of the subtree with root `x` is a ***bound function*** for the recursive algorithm in Figure 3.

```
node search(E key) {
1. if (T.root == null) {
2.   Throw a NoSuchElementException
   } else {
3.   return get(key, root)
   }
}
```

Figure 2: An algorithm to Search in this 2-3 Tree

5. Prove that this algorithm correctly solves the "Searching in a Subtree of This 2-3 Tree" problem that is given above.

6. Now let $T_{\mathsf{get}}(k)$ be an upper bound for the number of steps used by the algorithm when the depth of the subtree with the input node `x` as root is k, for $k \in \mathbb{N}$. Use the uniform cost criterion to define this.

   Write a ***recurrence*** for $T_{\mathsf{get}}(k)$ as a function of k. Explain why your answer (which you should make as precise as possible) is correct.

   Note: You should be able to use your recurrence to show that $T_{\mathsf{get}}(0) \leq 2$, $T_{\mathsf{get}}(1) \leq 7$, and $T_{\mathsf{get}}(2) \leq 12$.

7. Use your recurrence to give an upper bound in closed form (that is, not involving a summation or recurrence) for $T_{\mathsf{get}}(k)$ and prove that your bound is correct. Once again, this should be a function of k and you should try to make it as precise as possible.

2-3 Trees: Insertions

Consider, now, the problem of ***inserting*** an element into the set stored by a 2-3 tree. If this is to be implemented as a method supplied by this tree then it can be considered to solve the following computational problem.

Insertion into This 2-3 Tree

Precondition:

(a) This 2-3 tree, T, satisfies the 2-3 Tree properties given above.

(b) `key` is a non-`null` input of type `E`.

```
node get(E key, node x) {
 1.  if (x is a leaf) {
 2.    if (key is equal to the element stored at x) {
 3.      return x
       } else {
 4.      Throw a NoSuchElementException
       }
 5.  } else if (x has two children) {
 6.    if (key ≤ x.firstMax) {
 7.      return get(key, x.firstChild)
       } else {
 8.      return get(key, x.secondChild)
       }
     } else {
 9.    if (key ≤ x.firstMax) {
10.      return get(key, x.firstChild)
11.    else if (key ≤ x.secondMax) }
12.      return get(key, x.secondChild)
       } else {
13.      return get(key, x.thirdChild)
       }
     }
}
```

Figure 3: An Incomplete Algorithm to Search in a Subtree of this 2-3 Tree

Postcondition:

(a) If the input `key` is not already in the subset of `E` represented by T then it is added to this set (with this set 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 properties given above.

There are a few cases where this problem is easy, even though a node has to be added:

8. *Briefly* describe how to solve the above problem in the following cases.

   (a) T is an empty tree, so that its root is a `null` node.

   (b) T is not empty, but the set it represents has size one (so that its root is a leaf),

Suppose, for the rest of this assignment, that `addFirstElement` is a simple algorithm (based on your answer for part (a)) that takes an input `key` and can be used to complete the insertion operation when T is empty, and that `addSecondElement` is another algorithm (based on your answer for part (b)) that also takes an input `key` and can be used to complete the insertion operation when the root of T is a leaf.

A problem that arises, in other cases when 2-3 trees are modified, is that *internal nodes temporarily have illegal numbers of children*.

With that noted, consider the following ***Modified Tree Properties***, which might be satisfied by a rooted tree T when an operation is in progress.[2]

(a) 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 either (exactly) one or four children — all other internal nodes of T have either exactly two children or exactly 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, `x` stores a 1-tuple (m_1) — where m_1 is the largest value stored at any leaf of the first subtree of the subtree with root `x`.

(d) If an internal node `x` of T has exactly 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 with root `x`, respectively — then each of the values stored at the leaves of the first subtree is ***less than*** each of the values stored at the leaves of the second subtree, each of the values stored at the leaves of the second subtree is ***less than*** each of the values stored at the leaves of the third subtree, and each of the values stored at the leaves of the third subtree is ***less than*** each of the values stored at the leaves of the fourth subtree.

(e) If an internal node `x` of T has exactly four children then `x` 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

[2] As for the "2-3 Tree Properties", some of these introduce ***terminology*** that will be used in the rest of the assignment.

value stored at any leaf of the *second* subtree, m_3 is the largest value stored at any leaf of the *third* subtree, and m_4 is the largest value stored at any leaf of the *fourth* subtree of the subtree with root `x`.

(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 every node `x` in T.

Consider the following computational problem. It might not be clear why this is useful as described here (but it will be).

Insertion into Subtree of This 2-3 Tree

Precondition:

(a) This 2-3 tree, T, satisfies the 2-3 Tree properties given above.

(b) `key` is a non-`null` input of type `E`.

(c) `x` is a non-`null` node in T.

(d) Either `key` is not stored in any leaf of T at all, or it is stored in a leaf of the subtree of T with root `x`. If it is not stored at any of these leaves then it is possible to add a new leaf storing this `key`, as a new child of one of the internal nodes in the subtree with root `x` (without making other changes) in such a way that T satisfies the "Modified Tree" properties.

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 of 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.

Suppose, now, that `insertIntoSubtree` is a method that solves this computational problem.

```
void insert(E key) {
1.  if (T.root == null) {
2.    addFirstElement(key)
3.  } else if (T.root is a leaf) {
4.    addSecondElement(key)
    } else {
5.    insertIntoSubtree(key, T.root)
6.    if (T.root has four children) {
7.      fixRoot()
      }
    }
}
```

Figure 4: An Algorithm for the "Insertion into This 2-3 Tree" Problem

9. Suppose that the root of T is an internal node and the `insertIntoSubtree` method is called with an input `key` $\in$ `E` and with the root of T as input. Suppose, as well, that the input `key` was not initially stored at any leaf of T — and that the root of T has four children when the execution of this method ends.

   Briefly describe how to modify T so that the 2-3 Tree properties are restored (that is, T is a 2-3 tree, once again) — without changing the subset of `E` represented by T, and using only a constant number of steps.

Suppose, now, that `fixRoot` is a method (with no inputs) that can be used to carry out the computation described in the above question — and that this method is only applied when T satisfies the above "Modified Tree" properties and its root is an internal node with four children.

If the `addFirstElement`, `addSecondElement`, `insertIntoSubtree` and `fixRoot` methods are as described above, then an `insert` method that solves the "Insertion into This 2-3 Tree" problem is as shown in Figure 4 — where "`T.root`" is a reference to the root of T.

The only method that has not been described is the `insertIntoSubtree` method that solves the above "Insertion into Subtree of This 2-3 Tree" problem, and which is used at line 5. It will be useful to describe another pair of methods that will be used by this one.

- Suppose that T satisfies the above "Modified Tree" properties and `x` is an internal node of T, whose children are also internal nodes, such that either
  - one of the *children* of `x` has four children, or

– there is no internal node of T with either one or four children at all, so that T is a 2-3 tree.

Suppose, as well, that if it called with input `x`, then the `raiseSurplus` method modifies T, without changing the subset of `E` represented by T or changing the position of `x` in T (that is, without changing the distance or path from `x` to the root), so that T still satisfies the above "Modified Tree" properties and either

– `x` has four children, or
– there is no internal node of T with either one or four children, so that T is a 2-3 tree.

One way to perform this computation is to create an `ArrayList` of nodes, and another `ArrayList` of the largest elements of `E` stored at the subtrees with these nodes as roots. If none of the children of `x` has four children then the first `ArrayList` will simply store these children. On the other hand, if some child of `x` has four children then this should be replaced by *two* internal nodes, that each have two children, namely, two of the four children of this child. In effect this child of `x` should be ***split*** into a pair of internal nodes, increasing the number of children of `x` by one.

With a bit of work, it can be confirmed that the number of steps needed for an execution of this method is bounded by a constant.

- Suppose, instead, that T is a 2-3 tree, `x` is an internal node of T whose children are leaves, `key` is a non-`null` element of `E` that is not stored in any of the leaves of T, and the the "Modified Tree" properties would still be satisfied if `x` was given another child, namely, a new leaf storing the `key`.

  Suppose that, if called with the above `key` and internal node `x` as inputs, the method `addLeaf` adds another leaf, storing the input `key`, as another child of `x` — so that T still satisfies the "Modified Tree" properties, the subset of `E` represented has been changed by including `key` (and making no other changes), and either `x` has four children or T is a 2-3 tree.

  Once again, an `ArrayList` — of elements of `E`, to be stored at the children of `x` — can be useful as part of an implementation this method, and this method can also be implemented so that the number of steps used by an execution of the method is at most a constant.

Suppose, now, that the `insertIntoSubtree` method is as shown in Figure 5 on page 11. It is assumed that `x.firstChild`, `x.secondChild` and `x.thirdChild` are the first, second and third children of `x`, respectively, and that `x.firstMax`, `x.secondMax` and `x.thirdMax` are the largest elements of `E` stored that the leaves of the subtrees of T with these children of `x` as roots (when these children exist).

```
void insertIntoSubtree(E key, node x) {
 1. if (x is a leaf) {
 2.   Set e to the be the element of E stored at x
 3.   if (e is equal to the input key) {
 4.     Throw an ElementFoundException
      } else {
 5.     Throw a NoSuchElementException
      }
 6, } else {
 7.   try {
 8.     if (x has two children) {
 9.       if (key ≤ x.firstMax) {
10.         insertIntoSubtree(key, x.firstChild)
          } else {
11.         insertIntoSubtree(key, x.secondChild)
          }
        } else {
12.       if (key ≤ x.firstMax) }
13.         insertIntoSubtree(key, x.firstChild)
14.       } else if (key ≤ x.secondMax) {
15.         insertIntoSubtree(key, x.secondChild)
          } else {
16.         insertIntoSubtree(key, x.thirdChild)
          }
        };
17.     raiseSurplus(x);
18.   } catch (NoSuchElementException ex) {
19.     addLeaf(key, x)
      }
    }
}
```

Figure 5: The `insertIntoSubtree` Method

10. Suppose that 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` and a node `x` that is a leaf in T. Tracing through the execution of the pseudocode shown in Figure 5, as needed, explain why this method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied.

11. Suppose, again, that the precondition for the "Insertion into Subtree of This 2-3 Tree" problem is satisfied and the `insertIntoSubtree` method is executed with the `key` and with an *internal node* `x` whose children are leaves, as inputs. Tracing through the execution of the pseudocode shown in Figure 5, as needed, explain why this method would terminate with the postcondition for the "Insertion into Subtree of This 2-3 Tree" problem satisfied.

    Note: It might be helpful to compare the algorithms shown in Figures 3 and 5, and make use of similarities in their structure and analysis.

12. Describe how it can be proved that the `insertIntoSubtree` algorithm correctly solves the "Insertion into Subtree of This 2-3 Tree" problem. Your answer does not need to be a complete proof. However, you should identify the proof technique that is used, describe what you are inducting on and give both an "Inductive Hypothesis" and "Inductive Claim" whenever induction is to be used, and briefly describe any cases that must be handled (including, but not limited to, situations considered when solving the above problems).

13. Briefly describe how to prove that the number of steps executed by this method (when it starts with its problem's precondition satisfied) is in $O(\text{depth}(\mathtt{x}))$ where `x` is the node given as input — and where the Uniform Cost criterion is used to define this number of steps.

    It is not necessary to give a complete proof. However, a bound function for this method should be identified. and you should explain *how* the upper bound for the number of steps, given above, is established.

Once again, an examination of the `insert` method shown in Figure 4 should confirm that this correctly solves the "Insertion into This 2-3 Tree" problem. Furthermore, the number of steps used by an execution of this method is at most linear in the depth of this tree T.

2-3 Trees: Deletions

Now consider the problem of ***deleting*** a value from the subset of E represented by a 2-3 tree. This problem can be defined as follows.

Deletion from This 2-3 Tree

Precondition:

(a) This 2-3 tree, T, satisfies the 2-3 Tree properties given above.

(b) `key` is a non-`null` input of type `E`.

Postcondition:

(a) If the input `key` belongs to the subset of `E` represented by T then the `key` is removed from this set, which is otherwise unchanged. A `NoSuchElementException` is thrown and T is not changed if the `key` does not belong to this subset of `E`.

(b) T satisfies the 2-3 Tree properties given above.

14. Describe related problems to be solved, algorithms that solve these problems, and sketches of proofs of correctness, as needed to describe a solution for this computational problem too.

    It will probably be helpful to consider the development of the solution for the "Insertion into This 2-3 Tree" problem — that you have completed if you have solved the problems before this — and describe significant differences between these two problems and their solutions.

    - In both case some sort of "problem node" has probably been identified and then moved from the area near a leaf up to the root of the tree. Say what kind of "problem node" is involved for the "Deletion" problem instead of the "Insertion" one.
    - You may find that lots of the related problems are similar to the ones needed for "Insertion" problem. Make sure that you clearly ***identify*** which problems are similar, as you go, and describe the differences in enough detail so that a reader can understand this.

      One difference *might* be that *grandchildren* of a node should be considered, at a place where the *children* a node got considered by the corresponding part of the "Insertion" algorithm.

      Please allow lots of time to answer this question! It will probably required more work (and a different kind of activity) than the questions before it.

Implementation as a Java Program

15. An incomplete Java program, `TwoThreeTree.java`, is now available. Details of the implementation of a node for a 2-3 tree are consistent with the pseudocode given in this assignment.

    Please complete this, by supplying code for the methods whose content currently consists of the phrase

    ```
    // FOR YOU TO COMPLETE
    ```

    (or something very similar, if there are typographical errors causing similar comments to be used too).

    Do not change the code that has already been supplied: This may cause tests used to evaluate your submission to fail. A complete solution has been prepared and tested, so it has already been confirmed that no such changes are needed: If you feel that a change is needed then this is probably evidence that you do not understand the problem to be solved, or that you need to change the code you are including.

    However, you will need to introduce additional methods. It might be helpful to add one or two more methods to reduce repetitiveness in the code that you would otherwise need to provide to complete an implementation of an `insert` algorithm. You should also include methods solving the additional computational problems you discussed when answering the previous question, in order to complete the implementation of a `delete` algorithm.

    Additional code for testing — with information about how to use it — will be made available online very soon (and, in plenty of time for you to use it).