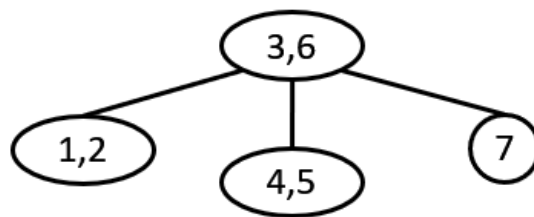


CSCI-3103
Data Structures and Algorithm

Obscure Binary Search Tree

Applications and Advantages of 2-3 Trees

Ze Sen Tang and Zhen Ze Ong



Contents

Acknowledgments:	3
Abstract.....	4
Introduction	4
Problem Statement.....	5
Project Objectives	5
Code	7
Time Complexity:	18
Real-World Applications:	22
References	23

Acknowledgments:

Team members:

Ze Sen Tang and Zhen Ze Ong

Mentor:

Dr. Ahmad Al Shami

Organization:

Southern Arkansas University

Abstract:

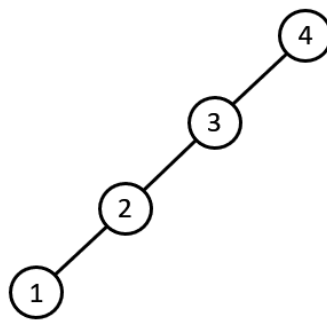
Items, such as names, numbers, etc. can be stored in memory in a sorted order called Binary Search Trees or BSTs. And some of these data structures can automatically balance their height when arbitrary items are inserted or deleted. Therefore, they are known as self-balancing BSTs. Further, there can be different implementations of this type, like the B-Trees, AVL trees, and red-black trees. But there are many other lesser-known executions that you can learn about. Some examples include AA trees, 2-3 trees, splay trees, and scapegoat trees. This project is based on the 2-3 tree and how it can outperform the regular BST in different circumstances.

Introduction:

The main target of establishing a binary search tree is to reduce the time complexity of operations and balance problems in the data structure. To achieve this target, computer scientists designed several different binary search trees that can maximize the efficiency of operations on the data in different circumstances. The most famous Binary Search Trees they designed are the B-trees, AVL trees, and red-black trees, used in many different areas. However, some lesser-known Binary Search Trees may outperform in other areas of utilization. Our target is to research the lesser-known binary search tree, the 2-3 tree, and study the benefits of using it in certain circumstances where it may outperform other binary trees.

Problem Statement:

A common problem that occurs in Binary Search Trees is the unbalance in the structure. The strength of a BST comes from its ability to divide and conquer, significantly cutting down the time required to locate a specific node and its data. Most BSTs have the time complexity of $O(\log N)$, and a BST is only able to achieve this by being balanced, meaning that both sides of the tree have a similar height or contain a similar number of nodes. If this balance is not accomplished, the time complexity of a BST begins to deteriorate. Thus, to tackle this problem, any BST must have a function to check and balance its structure.

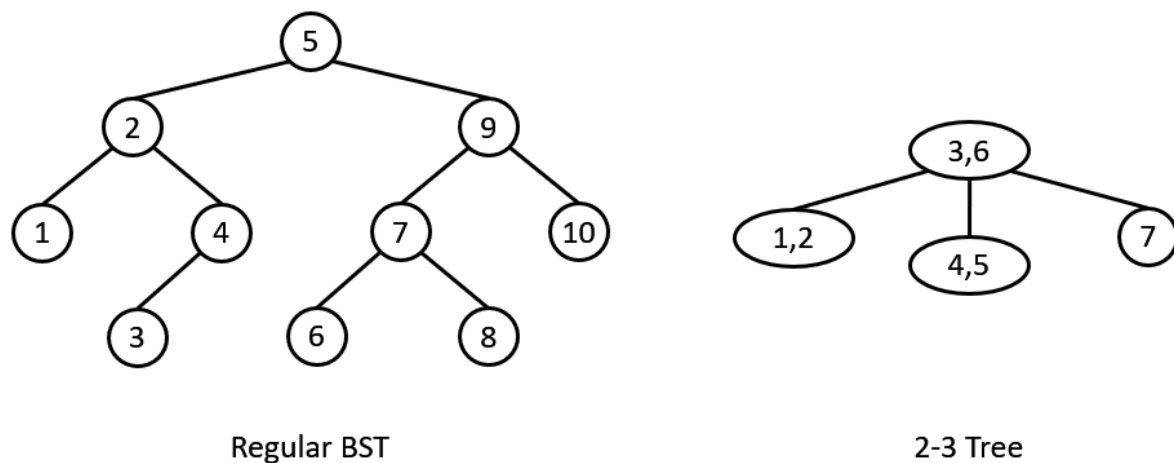


An unbalanced BST with $O(n)$ time complexity

Project Objectives:

To solve this problem, we created a self-balancing function in the binary search tree to manage its structure into a balanced form. One way to implement this self-balancing function is by increasing the number of data and children a node can hold. Furthermore, we maximize the number of children (if it has children) in each node of the tree such that no node holds numbers of children less than its maximum capacity. Another advantage of this implementation is that the height of the tree will be minimized because some values are

stored in the same node. This advantage reduced the time complexity of all functions (search, insert, remove) and increased the efficiency of the code. A special kind of these self-balancing trees is called 2-3 trees. Below is an example of 2-3 trees in comparison to the regular binary search tree.



A 2-3 Tree has two kinds of nodes. The first kind is called the "2 node", which contains only one value and exactly two children (or no child at all if it is a leaf node); the second kind is called the "3 node", which contains two values and exactly three children (or no child at all if it is a leaf node). With these two kinds of nodes, we can implement a special code in each function of the tree and give the tree itself the ability to check and decide when or how each node must be modified when a new value is inserted or removed. This structure makes it possible for the tree to check and balance itself whenever an insert or remove function causes the tree to become an unbalanced form.

The data arrangement in 2-3 trees is similar to what a regular tree does. The left side child is always smaller than the values of the parent node, and the right side is always larger. The only difference shows up when a node holds 2 values. In this case, the middle child node

must be present, and the value held by the middle node must fall between the range of the two-parent values.

With these advantages, the data structure will be balanced all the time and limits the time complexity for all functions around $O(\log N)$ as the data becomes larger and larger.

Furthermore, it also makes the BST a very stable data structure by having the same time complexity in either the average or worst-case scenarios. A 2-3 Tree is very strict in keeping a balanced structure whether an add or remove function is called.

Code:

```
1 class Node:
2     def __init__(self, value): ...
10    def is_leaf(self): ...
14    def is_full(self): ...
18    def get_child(self, value): ...
31    def brother(self): ...
```

The node class

As for any BST, a node class must be defined. A node can be defined as the storage for the values we wish to store in the BST. A difference between the nodes of a 2-3 tree and a regular binary search tree is that a 2-3 tree's node can hold up to two values. As shown below, we define `value1` as well as `value2`. The default value of `value2` will be null until the program determines that `value1` is occupied and adds the new value to `value2`.

The value in `value2` cannot be smaller than that of `value1`. A 2-3 tree can also allow nodes to have up to three children nodes, and we define the reference values to those children nodes with `left`, `mid`, and `right` for the left, middle, and right children

respectively. The `parent` holds a reference to the node for which the node in question is a child. All these values are initially set to null.

```
2 v      def __init__(self,value):
3          self.value1 = value
4          self.value2 = None
5          self.left = None #left child
6          self.mid = None #mid child
7          self.right = None #right child
8          self.parent = None #parent
```

The `__init__` function

The `is_leaf` function checks if the node is a leaf node. A leaf node is a node that has no children. This function can easily determine this by checking if all children's reference values are empty.

The `is_full` function checks if the node has two values. As mentioned before, a 2-3 tree's node can only hold up to two values. If it exceeds two, the node must split to maintain the 2-3 tree structure. The `is_full` function must be called before adding a new value to make sure that this problem does not arise.

We can traverse the tree using the `get_child` function. This function determines whether the value we are searching for is smaller, between, or larger than the parent node, and returns a reference to the left, middle, or right child respectively. This function is usually called multiple times to locate certain nodes.

The `brother` function is used to determine the other child (or children) of the parent node. This function is only used in remove functions which we will discuss later. This

brother function can only be run if there is a parent node and typically prioritizes returning the right child of the parent node before the rest.

```
48 v class TwoThreeTree:
49 >     def __init__(self): ...
52 >     def search(self, value): ...
58 >     def search_node(self, root, value): ...
65 >     def insert(self, value): ...
72 >     def insert_node(self, root, value): ...
80 >     def put_item(self, leaf, value): ...
87 >     def put_item_full(self, leaf, value): ...
123 >     def split(self, leaf, value): ...
139 >     def remove(self, value): ...
166 >     def get_predecessor(self, root): ...
171 >     def remove_leaf(self, node, value): ...
191 >     def remove_item(self, node, value): ...
197 >     def leaf_case1(self, node, brother): ...
249 >     def leaf_case2(self, node, brother): ...
280 >     def merge(self, node, brother): ...
302 >     def in_order(self, root): ...
323 >     def traversal(self): ...
```

The TwoThreeTree class

The class for a 2-3 tree, however, is much more complicated than one of a regular tree.

Because of this, some cases of removal must be solved in separate functions, as seen in

leaf_case1 and leaf_case2.

The `__init__` function initializes the root to null, since the tree will be empty initially. The

`search` function checks if the tree is empty, if not, it calls the `search_node` function.

This function searches through the tree until it either finds the node with the value or

reaches the end of the tree. Notice in the code below that it calls the `get_child` function multiple times to determine whether it should traverse to the left, middle, or right child.

```
49 v     def __init__(self):
50         self.root = None
51
52 v     def search(self,value):
53 v         if not self.root:
54             print('The tree is null!')
55             return
56         return self.search_node(self.root,value)
57
58 v     def search_node(self,root,value):
59 v         if not root:
60             return None #return None if node is not found
61 v         if value == root.value1 or value == root.value2:
62             return root #node is found
63         return self.search_node(root.get_child(value),value)
        #search from the node's children
```

The `__init__`, `search`, and `search_node` functions

Inserting an item into a 2-3 tree may become complicated in certain situations. The simplest case is when the tree is empty. The `insert` function checks this, and simply initializes the new value as the root. If not, it calls the `insert_node` function. The `insert_node` function can only insert a new value into a leaf node. This is a unique characteristic of 2-3 trees. A 2-3 tree grows by pushing values upwards, while a regular tree grows downwards. The `insert_node` function calls a while loop until it reaches the correct leaf node. Then it checks if the leaf node contains one or two values. If it does not contain two values, the node is not considered full and `put_item` is called. Otherwise, `put_item_full` is called.

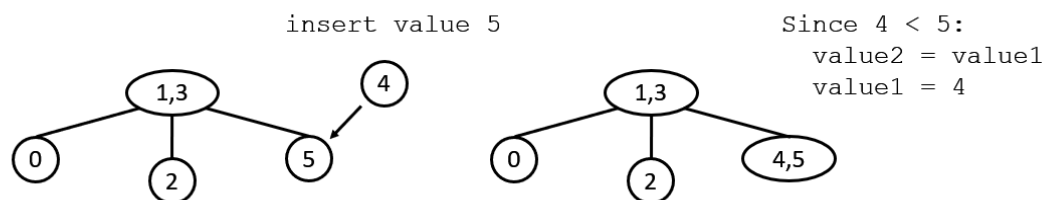
```

65 ▼ def insert(self,value):
66     print('Insert', value)
67 ▼     if not self.root: #if the tree is null
68         self.root = Node(value)
69         return
70     self.insert_node(self.root,value)
71
72 ▼ def insert_node(self,root,value):
73 ▼     while not root.is_leaf(): #insert only at leaf node
74         root = root.get_child(value)
75 ▼     if not root.is_full(): #if the node we insert is not full
76         self.put_item(root,value)
77 ▼     else: #if the node we insert is full
78         self.put_item_full(root,value)

```

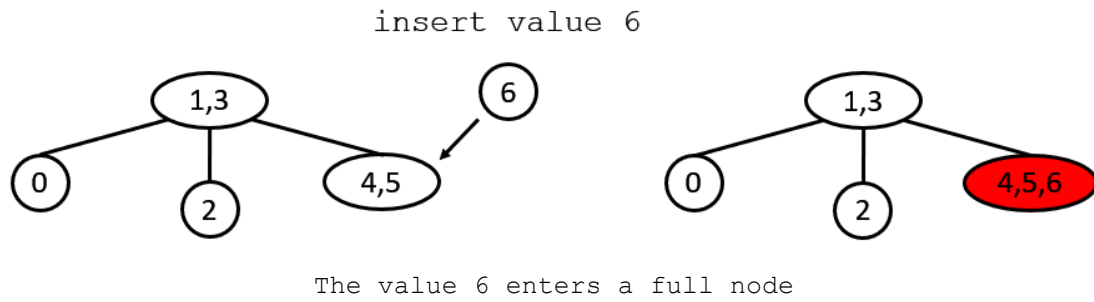
The insert and insert_node functions

The `put_item` function occurs when the node is not full. It simply inserts the new value into the node either into `value1` or `value2` depending on whether the new value is smaller or greater than the existing value.

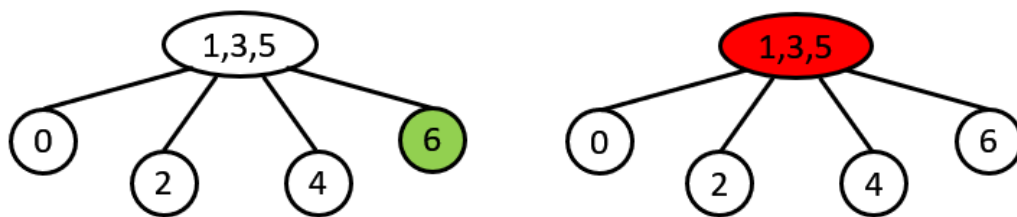


A visual representation of a `put_item` function

However, the `put_item_full` function is more complicated. Below is an example of when `put_item_full` is called.

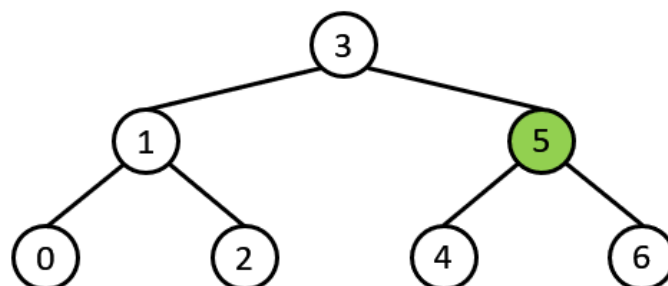


When a node is full, the values are split through the `split` function and placed into different nodes. The largest value on the right is placed in a new node indicated by green, and the middle value gets pushed up into the parent node. The smallest value on the left stays in the existing node.



The full node splits and value 5 is pushed into the parent node

Notice that in this case, the value 5 is pushed into a parent node that is also full. The `put_item_full` function is called once more as shown below.

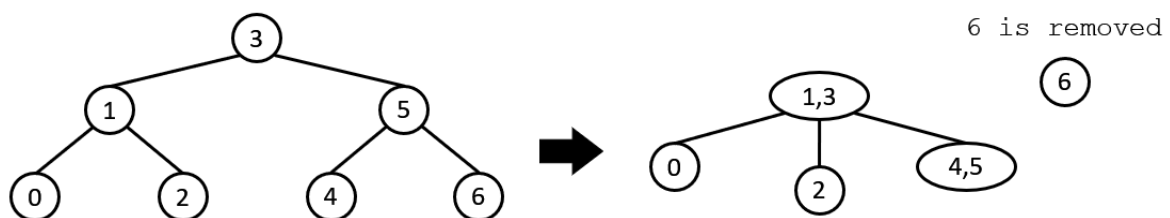


The full node splits and value 3 is pushed into a new parent node

The `split` function is called, and the middle value 3 gets pushed up into a parent node.

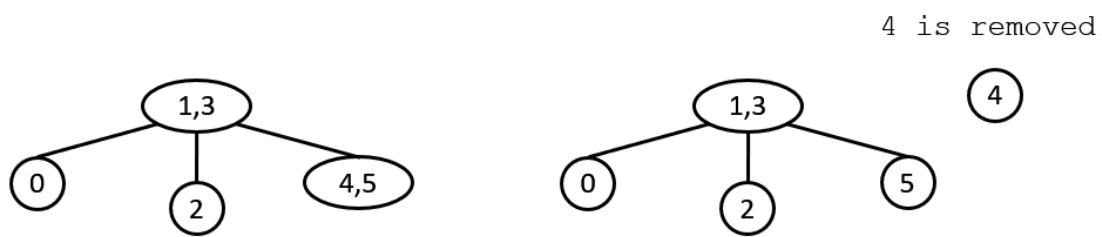
Since there is no existing parent node, a new one must be created for value 3. The right value of 5 enters a new node and value 1 stays in the existing node. The four children nodes are assigned to the two new children nodes. Notice how the 2-3 tree grows upwards by creating a new parent node. This is opposed to the downward growth of regular trees.

Removing is much more complicated than inserting in 2-3 trees. Although removing a value does not cause much work, the necessity to maintain the balance of the tree takes extra steps. For instance, if we want to remove the value 6 from the previous example, we will have to merge values 4 and 5, then 1 and 3, to revert to what we had initially. In a similar case in a regular tree, the leaf node containing 6 would simply be removed. However, this would be considered unbalanced in a 2-3 tree, which is not allowed.



Removal of 6 in the previous example

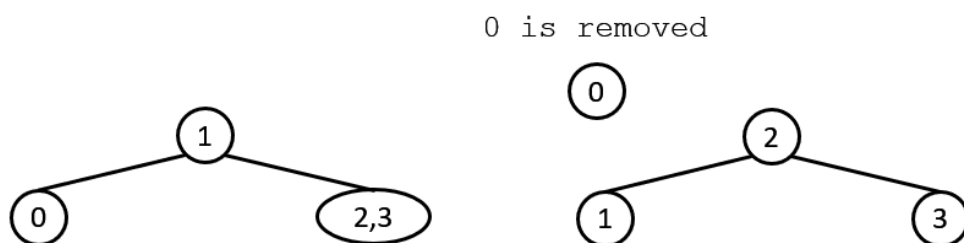
The `remove` function first checks that the tree is not empty and the value we wish to remove exists in the tree. If both are true, it then determines which case the removal falls into. If the value is in a leaf node, the `remove_leaf` function is called. If the node is full, it will simply call the `remove_item` function which removes the value, since the balance of the tree will be maintained.



Removing a value of a full node

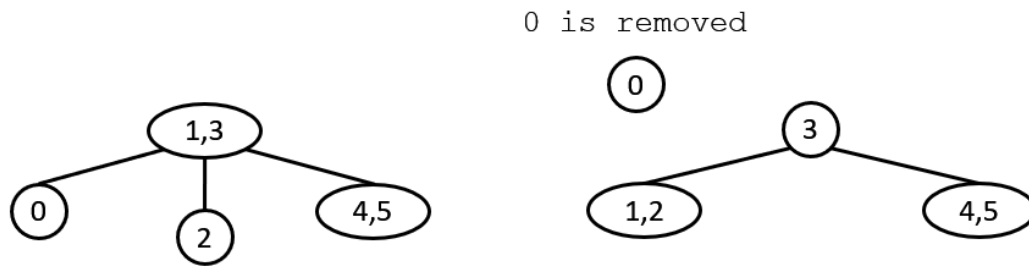
If the leaf node is not full, it has to further determine which case the leaf removal falls under. It then calls either the `leaf_case1` or `leaf_case2` function.

The first case is shown below, where the parent node has to be changed to keep the balance of the tree. First, the parent value replaces the value that is to be removed, then `value2` of the brother node is pushed into the parent node, using the previously defined `brother` function.



leaf_case1

In the second case, a value from the parent node has to enter the correct child node using the `merge` function. The `merge` function reverses the actions of the `split` function. It is much more complicated since it has to locate the correct node to merge using the `brother` function. This allows the 2-3 tree to stay true to its structure, where a parent node cannot have two values without having three children.



leaf_case2

```

280 ▼ def merge(self,node,brother):
281     #put parent's value1 into brother
282     self.put_item(brother,node.parent.value1)
283     #if the node is in the left
284 ▼   if node is node.parent.left:
285 ▼       if node.mid:
286           brother.mid = brother.left
287           brother.left = node.mid
288           node.mid.parent = brother
289           node.parent.right = None
290 ▼   else: #if the node is in the right
291 ▼       if node.mid:
292           brother.mid = brother.right
293           brother.right = node.mid
294           node.mid.parent = brother
295           node.parent.left = None
296       node.parent.mid = brother
297       temp = node
298       node = node.parent
299       del temp
300       return node

```

The merge function

If the `remove` function determines that the value is not in a leaf node, it calls the `get_predecessor` function and exchanges the predecessor with the node to be removed, then we remove the node as a leaf node.

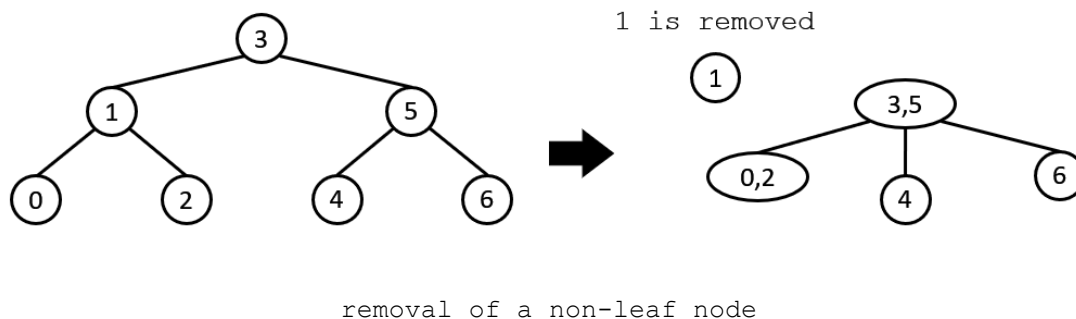
```

166 ▼ def get_predecessor(self, root):
167 ▼     if root.right:
168         return self.get_predecessor(root.right)
169     return root

```

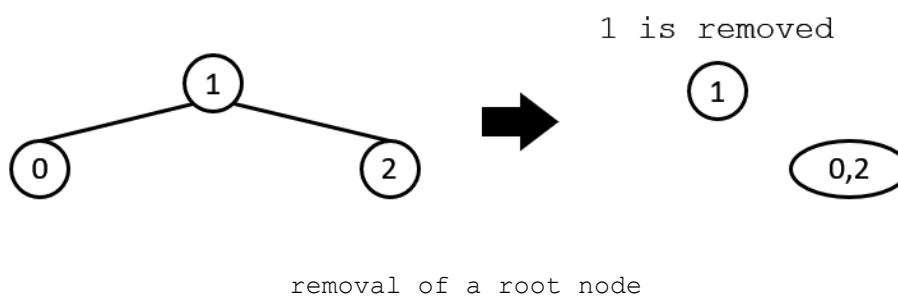
The `get_predecessor` function

It then removes the node and makes adjustments accordingly, so that the structure of a 2-3 tree maintains its balance. There are many different cases of non-leaf node removals, which we will not cover completely in this paper. One example is the removal in the tree below.



This removes the non-leaf node with the value of 1 but multiple adjustments must be made. Its two children are merged, its brother node is merged with the parent node, and finally, the children of the brother node are distributed to the middle and right nodes, respectively.

Another case of a non-leaf node removal is a root that is not full. If both children of the root only have one value, they simply merge to become the new root, as seen below.



For this 2-3 tree we used an in-order transversal as defined by the `in_order` function. This function will traverse the tree starting from the left node, followed by the root node, and then the right node.

```
302 ▼ def in_order(self, root):
303 ▼     if not root.parent:
304         print('-----In Order Traversal-----')
305 ▼     if root.is_full():
306 ▼         if root.left:
307             self.in_order(root.left)
308             print(root.value1, end=' ')
309 ▼         if root.mid:
310             self.in_order(root.mid)
311             print(root.value2, end=' ')
312 ▼         if root.right:
313             self.in_order(root.right)
314 ▼     else:
315 ▼         if root.left:
316             self.in_order(root.left)
317             print(root.value1, end=' ')
318 ▼         if root.right:
319             self.in_order(root.right)
320 ▼     if not root.parent:
321         print('')
```

The `in_order` function

The `traverse` function simply checks that the tree is not empty and calls the `in_order` traversal function.

```
323 ▼ def traversal(self):
324 ▼     if not self.root:
325         print('The tree is null!')
326         return
327         self.in_order(self.root)
```

The `traversal` function

Time Complexity:

In 2-3 trees, the self-balancing ability is guaranteed whenever the data are changed. Thus, compared to regular trees, 2-3 trees guarantee that the time complexity of all functions in the tree will not exceed $O(\log N)$. In this paper, we will focus on four main functions in 2-3 trees: search, insert, remove, and traversal.

Search Functions:

```
51     def search(self,value):
52         if not self.root:
53             print('The tree is null!')
54             return
55         return self.search_node(self.root,value)
56
57     def search_node(self,root,value):
58         if not root:
59             return None #return None if node is not found
60         if value == root.value1 or value == root.value2:
61             return root #node is found
62         return self.search_node(root.get_child(value),value)
```

The 2 search functions

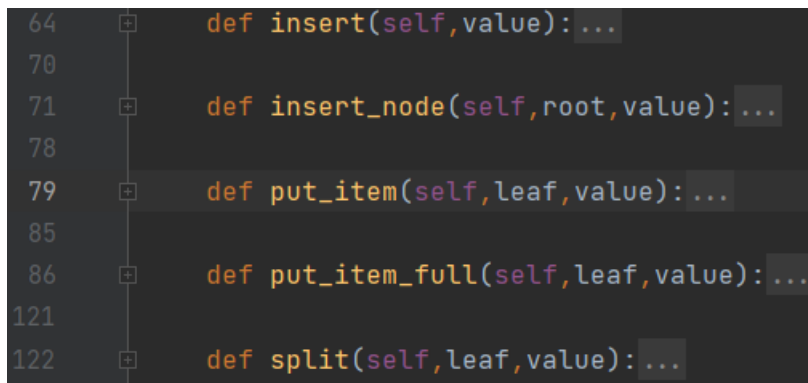
The search function is separated into two divisions: search and search_node. The search function only has 1 operation; thus, its time complexity is $O(1)$. The search_node function is recursive, and the number of its recursive calls depends on the height of trees h (the maximum call of search_node is h). Now, we must decide the relationship between h and N , where N is the total number of values. For the worst case, all nodes only hold one value, and the height of the tree is equal to $\log_2 N$. Thus, $h \leq \log_2 N$ for all time. Furthermore, each time the search_node function is called, 2 logical operations (lines 58 and 60) come into scope, so the total operation of search_node is

$2 \log_2 N$. Then, the time complexity of `search_node` is $O(\log N)$. Then, the overall time complexity of the search function is:

$$O(\log N) + O(1) \sim O(\log N)$$

Insert Functions:

Due to its complexity, we separate the insert code into 5 different parts.



```

64      def insert(self, value): ...
70
71      def insert_node(self, root, value): ...
78
79      def put_item(self, leaf, value): ...
85
86      def put_item_full(self, leaf, value): ...
121
122     def split(self, leaf, value): ...

```

The 5 insert functions

The `insert` function has a total of 2 operations, the `put_item` function has, and the `split` function has a total of 11 operations. All of them have the time complexity of $O(1)$. The `put_item_full` function has a while loop with a maximum loop of time h and a total of 32 operations in the loop, thus:

$$total\ operations \leq 1 + (1 + \log_2 N) + 32(\log_2 N) + 6 = 2 + 33 \log_2 N$$

The time complexity of the `put_item_full` function is $O(\log N)$. The `insert_node` function also has a while loop with maximum loop time h and 1 operation in the loop, thus:

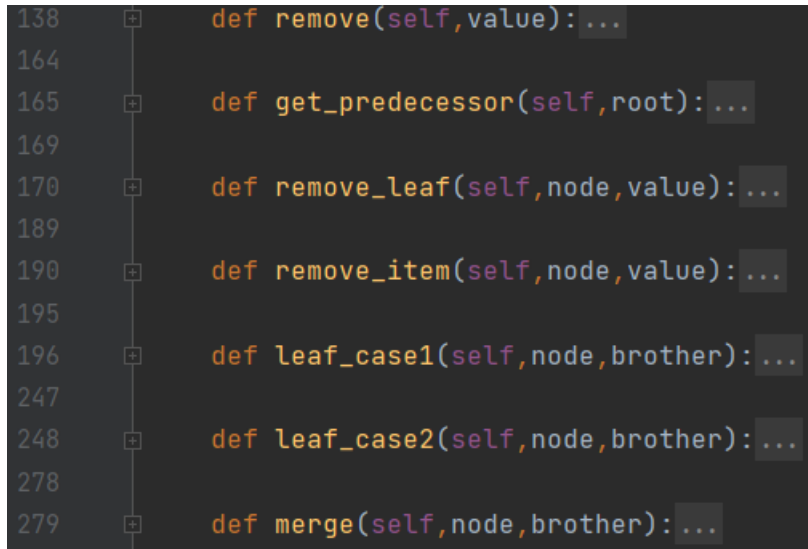
$$total\ operations \leq (1 + \log_2 N) + 1(\log_2 N) + 1 = 2 + 2 \log_2 N$$

The time complexity of the `insert_node` function is $O(\log N)$. The overall time complexity of all insert functions is:

$$O(1) + O(1) + O(1) + O(\log N) + O(\log N) \sim O(\log N)$$

Remove Functions:

The remove function is the most complex function in 2-3 trees. Because of its complexity, we divide it into 7 functions, shown in the picture below.



```

138 def remove(self, value): ...
164
165 def get_predecessor(self, root): ...
169
170 def remove_leaf(self, node, value): ...
189
190 def remove_item(self, node, value): ...
195
196 def leaf_case1(self, node, brother): ...
247
248 def leaf_case2(self, node, brother): ...
278
279 def merge(self, node, brother): ...

```

The 7 remove functions

There are 13 total operations in the `remove` function, 3 in `remove_item`, 37 in `leaf_case1`, 18 in `leaf_case2`, and 14 in `merge`. All the time complexities are $O(1)$.

The `get_predecessor` function is a recursive function with maximum h calls and 1 operation in the function. Thus, the total operations of it are $1(\log_2 N)$, and the time complexity is $O(\log N)$. The `remove_leaf` function has a while loop with maximum loop h and 4 operations in the loop. Outside the while loop, it also has 3 operations. Then, the total operations of `remove_leaf` are:

$$3 + (1 + \log_2 N) + 4(\log_2 N) = 4 + 5 \log_2 N$$

Its time complexity is $O(\log N)$. Finally, the overall time complexity of all the remove functions is:

$$5(O(1)) + 2(O(\log N)) \sim O(\log N)$$

Traversal Functions:

```

301     def in_order(self, root): ...
321
322     def traversal(self): ...

```

The 2 traversal functions

The `traversal` function has only one operation, so its time complexity is $O(1)$. The `in_order` function is a recursive function with maximum h calls and a total of 8 operations in each call. Adding all the operations in this function makes $5(8(\log_2 N - 1)) + 8$, and the time complexity is $O(\log N)$. Finally, the overall time complexity of all traversal functions is:

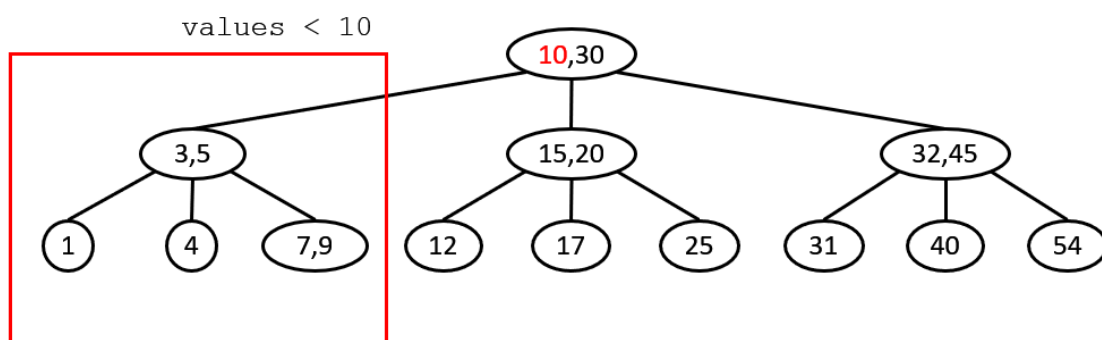
$$O(1) + O(\log N) \sim O(\log N)$$

Search	$O(\log N)$
Insert	$O(\log N)$
Remove	$O(\log N)$
Traversal	$O(\log N)$

Time Complexity Table

Real-World Applications:

The 2-3 tree can practically replace a regular tree in any situation. The benefit of using a 2-3 tree over a regular one is its balance in structure at all times. A good application of this advantage is processing a live stream of data, such as that of an online store. A 2-3 tree will always be balanced even though online orders are constantly being placed, which allows the database to function at its optimal speed at all times. Binary Search Trees overall are excellent in databases that must be sorted, for example, searching for items above or below a given cost. This comes from the basic structure of a tree, where all smaller values are located on the left and the larger values on the right.



Searching for values smaller than 10

References

- Boris. (n.d.). *2-3 TREE (PYTHON RECIPE)*. Retrieved from Active State:
<https://code.activestate.com/recipes/577898-2-3-tree/>
- Fu, H. (n.d.). *Algorithm-and-Complexity*. Retrieved from <https://github.com/infinityglow/Algorithm-and-Complexity/blob/master/Transform%20and%20Conquer/Two-Three%20Tree/2-3%20tree.py>
- Joe, J. (n.d.). *Python: 2-3 Trees Tutorial*. Retrieved from YouTube:
<https://www.youtube.com/watch?v=vSbBB4MRzp4>
- joeyajames. (n.d.). *2-3_tree.py*. Retrieved from GitHub:
https://github.com/joeyajames/Python/blob/master/Trees/2-3_tree.py
- Reddy, V. S. (n.d.). *Different Self Balancing Binary Trees*. Retrieved from OpenGenus IQ:
<https://iq.opengenus.org/different-self-balancing-binary-trees/>
- strictlysimplifiedesign. (n.d.). *23tree*. Retrieved from GitHub:
<https://github.com/strictlysimplifiedesign/23tree/blob/master/23tree.py>