AVL TREE

AVL Tree is a self-balancing binary search tree where the difference between left and right length of root node(and every node in the tree) must not be greater than 1

Left – right <= 1 (Always), if difference is greater then we gonna need rotation to achieve AVL proper

<u>Advantage:</u>
consider the case where all element node are in ascending order now in order to search last node while using BST it gonna take O(n) time complexity compared to O(logn) ideal case , AVL helps to remove this kind of scenario where it make tree self balance by doing rotation and make the time complexity to O(logn)

Operations:

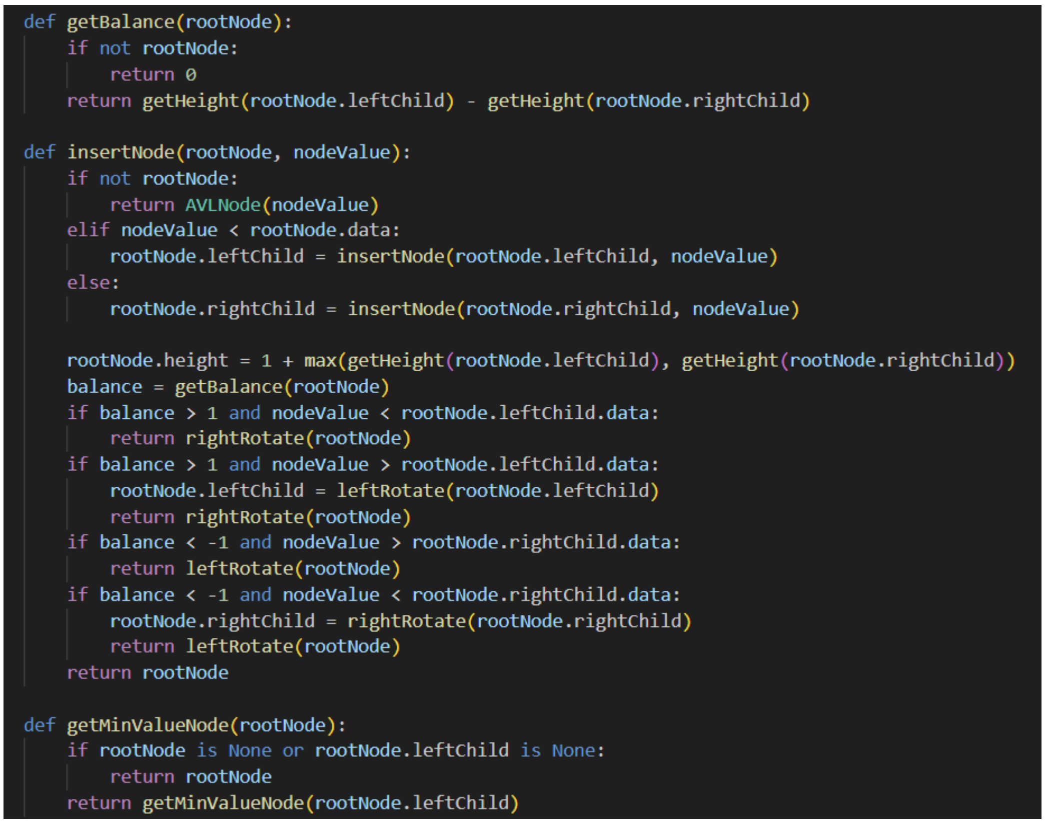
<u>Creation:</u> same as BST (with one extra attribute self.height = 1)
<u>Traversing</u>: same as BST
<u>Searching</u>: same as BST

<u>Insertion:</u>

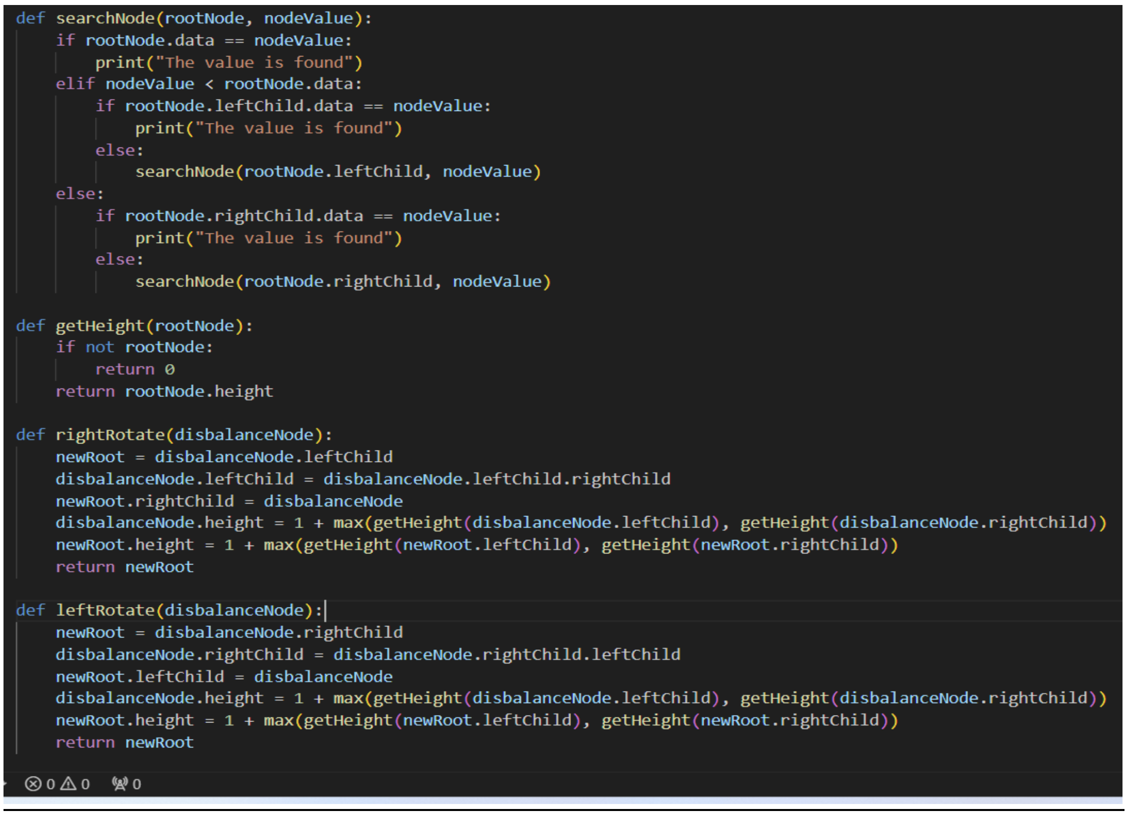
Rotation is Required after insertion: here we have conditions like LL,LR, RL,RR where rotation is required to make it self balanced binary search time after inserting new node

1. **<u>Left Left Condition(disbalanced to leaf)</u>**:
   if the grandchild from root node is LL sequence take the right orientation, while selecting grand child if grand child parent has 2 childs always select the child as grand child which has more height
   - ➔ disbalanced node will have new left child with value of sibling of grand child, if there is no sibling to grand child then this is NONE
2. **<u>Left Right Condition:</u>**
   if the grandchild is LR condition then we want to take the parent and right child of parent and make them left orientation now it becomes LL condition from root node(disbalanced) now we are gonna do right orientation so it is left and right orientation
   - ➔ Edge case of left child of leaf node is also must be considered
3. **<u>Right Right Condition:</u>**
   Same as LL as we did left orientation, here it is RR so we will do right orientation
4. **<u>Right Left Condition:</u>**
   Same kind of LR but here it is RL so we do initially Right orientation and then left orientation

   <u>By combining all these conditions we can make the tree self balanced while creating tree from given set of values take first as root node and after every insertion check tree is balanced if not check which condition we can apply to make it balance</u>



```python
def getBalance(rootNode):
    if not rootNode:
        return 0
    return getHeight(rootNode.leftChild) - getHeight(rootNode.rightChild)

def insertNode(rootNode, nodeValue):
    if not rootNode:
        return AVLNode(nodeValue)
    elif nodeValue < rootNode.data:
        rootNode.leftChild = insertNode(rootNode.leftChild, nodeValue)
    else:
        rootNode.rightChild = insertNode(rootNode.rightChild, nodeValue)

    rootNode.height = 1 + max(getHeight(rootNode.leftChild), getHeight(rootNode.rightChild))
    balance = getBalance(rootNode)
    if balance > 1 and nodeValue < rootNode.leftChild.data:
        return rightRotate(rootNode)
    if balance > 1 and nodeValue > rootNode.leftChild.data:
        rootNode.leftChild = leftRotate(rootNode.leftChild)
        return rightRotate(rootNode)
    if balance < -1 and nodeValue > rootNode.rightChild.data:
        return leftRotate(rootNode)
    if balance < -1 and nodeValue < rootNode.rightChild.data:
        rootNode.rightChild = rightRotate(rootNode.rightChild)
        return leftRotate(rootNode)
    return rootNode

def getMinValueNode(rootNode):
    if rootNode is None or rootNode.leftChild is None:
        return rootNode
    return getMinValueNode(rootNode.leftChild)
```



```python
def searchNode(rootNode, nodeValue):
    if rootNode.data == nodeValue:
        print("The value is found")
    elif nodeValue < rootNode.data:
        if rootNode.leftChild.data == nodeValue:
            print("The value is found")
        else:
            searchNode(rootNode.leftChild, nodeValue)
    else:
        if rootNode.rightChild.data == nodeValue:
            print("The value is found")
        else:
            searchNode(rootNode.rightChild, nodeValue)

def getHeight(rootNode):
    if not rootNode:
        return 0
    return rootNode.height

def rightRotate(disbalanceNode):
    newRoot = disbalanceNode.leftChild
    disbalanceNode.leftChild = disbalanceNode.leftChild.rightChild
    newRoot.rightChild = disbalanceNode
    disbalanceNode.height = 1 + max(getHeight(disbalanceNode.leftChild), getHeight(disbalanceNode.rightChild))
    newRoot.height = 1 + max(getHeight(newRoot.leftChild), getHeight(newRoot.rightChild))
    return newRoot

def leftRotate(disbalanceNode):
    newRoot = disbalanceNode.rightChild
    disbalanceNode.rightChild = disbalanceNode.rightChild.leftChild
    newRoot.leftChild = disbalanceNode
    disbalanceNode.height = 1 + max(getHeight(disbalanceNode.leftChild), getHeight(disbalanceNode.rightChild))
    newRoot.height = 1 + max(getHeight(newRoot.leftChild), getHeight(newRoot.rightChild))
    return newRoot
```

Deletion:

here we have 2 cases like is rotation required after deletion by comparing parent node of deleted child as reference else not required after deletion

Rotation is not required:

Same as BST deletion where :

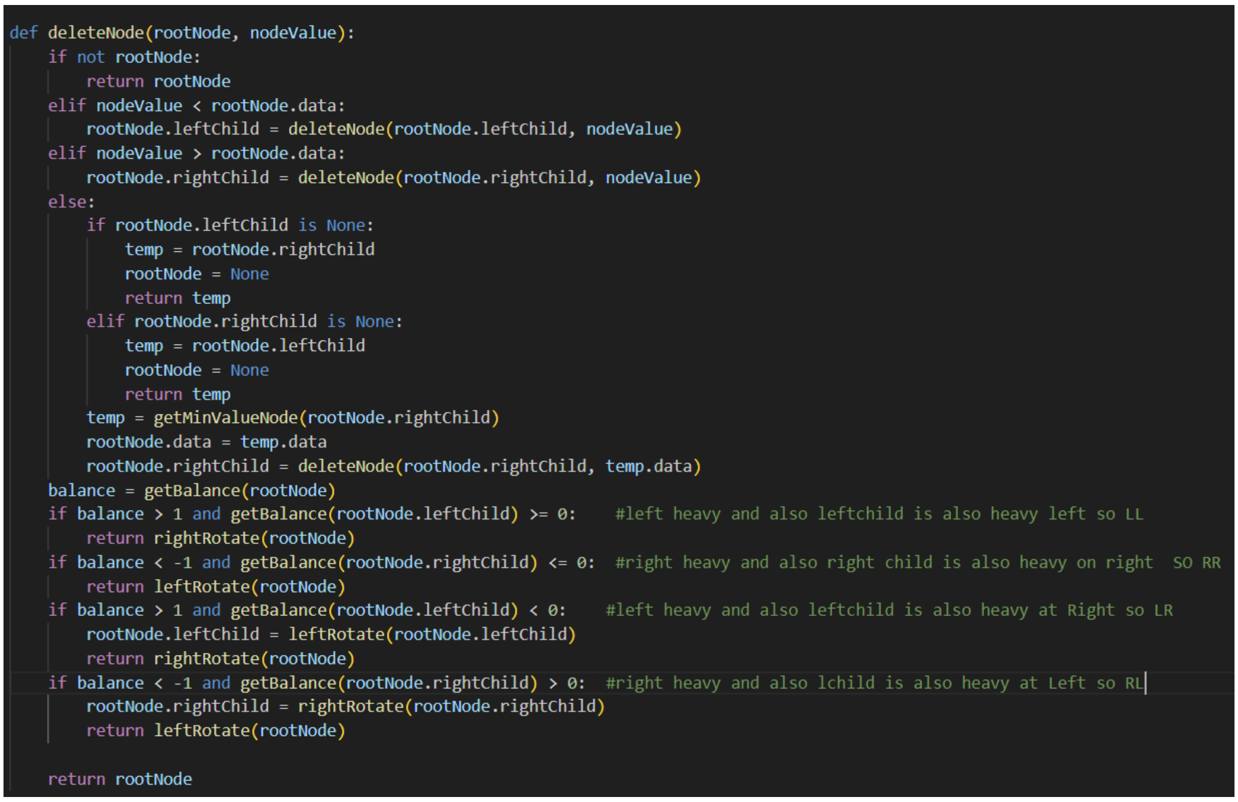
1.leaf node is deleted without any issues

2.node with one child here the child is interlinked with parent node and node is deleted

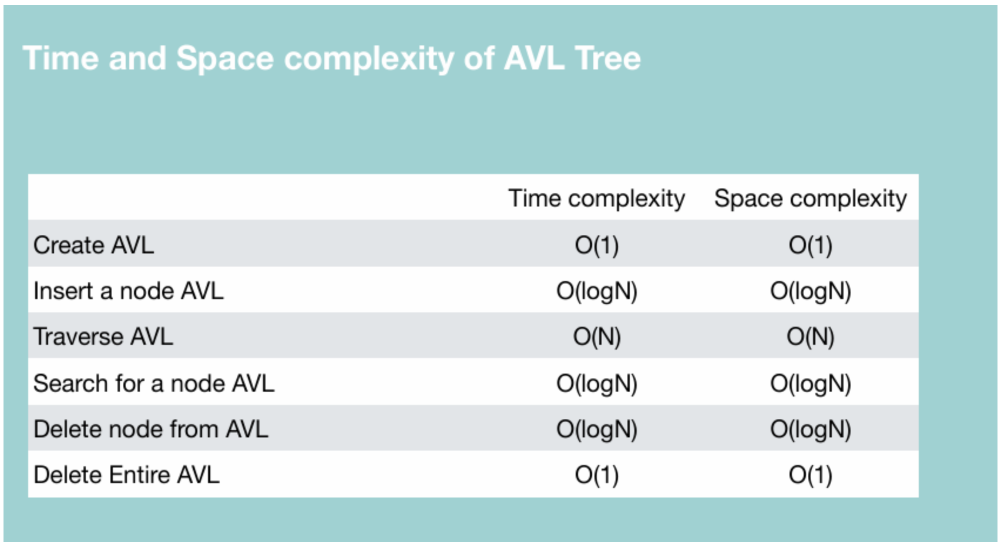
3.node with 2 childs here we need to find immediate successor(this node satisfy the child qualifying requirements) of deleting node and replace it with the successor and now delete the successor node

Rotation is required:

here after deletion we are gonna compare the parent node whether it is balanced or not if not balanced then we are gonna use same condition methods like RR,LL,RL,LR which ever applicable and make it balanced tree



```python
def deleteNode(rootNode, nodeValue):
    if not rootNode:
        return rootNode
    elif nodeValue < rootNode.data:
        rootNode.leftChild = deleteNode(rootNode.leftChild, nodeValue)
    elif nodeValue > rootNode.data:
        rootNode.rightChild = deleteNode(rootNode.rightChild, nodeValue)
    else:
        if rootNode.leftChild is None:
            temp = rootNode.rightChild
            rootNode = None
            return temp
        elif rootNode.rightChild is None:
            temp = rootNode.leftChild
            rootNode = None
            return temp
        temp = getMinValueNode(rootNode.rightChild)
        rootNode.data = temp.data
        rootNode.rightChild = deleteNode(rootNode.rightChild, temp.data)
    balance = getBalance(rootNode)
    if balance > 1 and getBalance(rootNode.leftChild) >= 0:    #left heavy and also leftchild is also heavy left so LL
        return rightRotate(rootNode)
    if balance < -1 and getBalance(rootNode.rightChild) <= 0:  #right heavy and also right child is also heavy on right  SO RR
        return leftRotate(rootNode)
    if balance > 1 and getBalance(rootNode.leftChild) < 0:    #left heavy and also leftchild is also heavy at Right so LR
        rootNode.leftChild = leftRotate(rootNode.leftChild)
        return rightRotate(rootNode)
    if balance < -1 and getBalance(rootNode.rightChild) > 0:  #right heavy and also lchild is also heavy at Left so RL
        rootNode.rightChild = rightRotate(rootNode.rightChild)
        return leftRotate(rootNode)

    return rootNode
```



Time and Space complexity of AVL Tree

	Time complexity	Space complexity
Create AVL	O(1)	O(1)
Insert a node AVL	O(logN)	O(logN)
Traverse AVL	O(N)	O(N)
Search for a node AVL	O(logN)	O(logN)
Delete node from AVL	O(logN)	O(logN)
Delete Entire AVL	O(1)	O(1)

BINARY HEAP

A binary heap is a binary tree with following properties:
1. It is categorized into 2 types max heap and min heap , in Min heap the key at the root node must be minimum than all childrens same applies to every root node in min heap, the same pattern applies to max heap but kind of reverse
2. It is a complete tree(All levels are filled completely except last one—located as left as possible)

Advantage:
Although AVL has many advantage since it is balanced it gives us a logn time complexity for insertion, and deletion…. Binary heap maintains this time efficiency along which it provides O(1) time complexity for maximum and minimum number finding operations

Practical use: heap sort, prims algo, priority queue

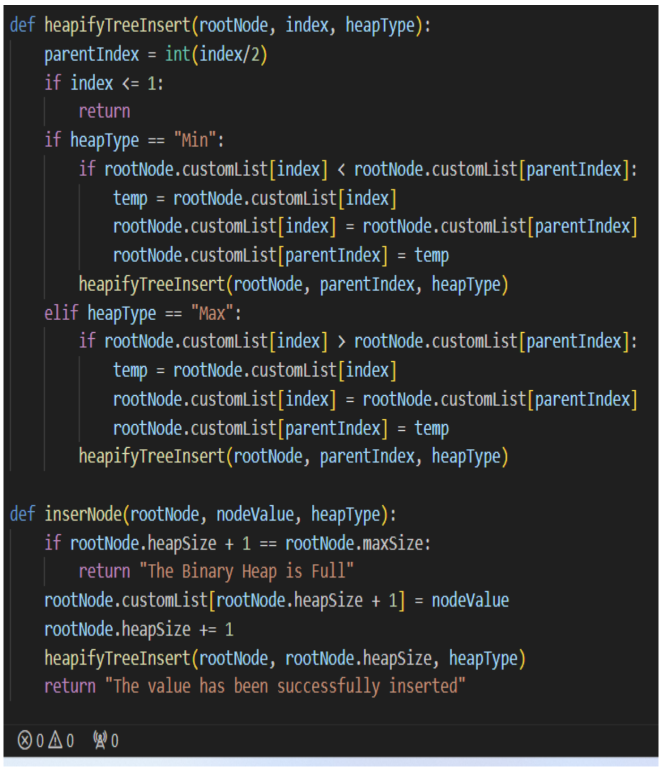
Can be implemented using lists and reference/pointer here we are using list representation because of its simplicity and efficiency.

List representation:
excluding index 0 if x is the parent 2x is the left child and 2x+1 is the right child, same representation as binary tree.

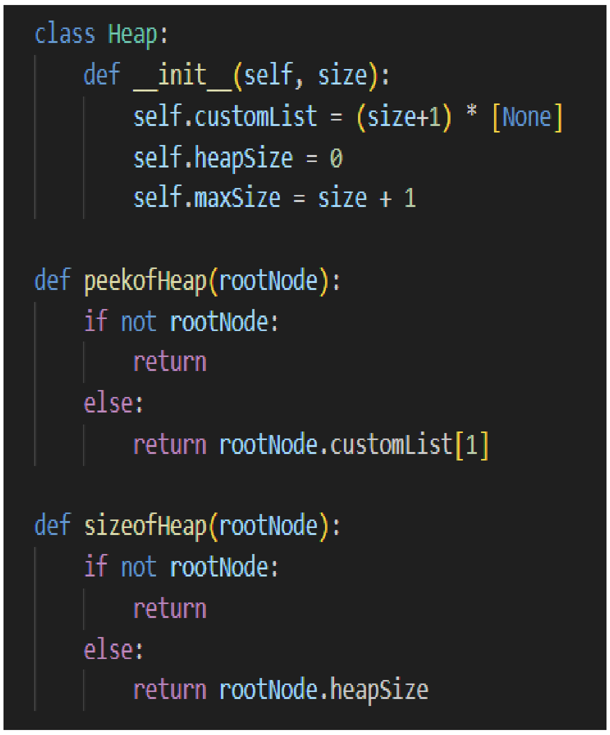
Operations:

Creation: 3 attributes custom_list, Heap_size used to access max or min last element, List_size
Peeking: access the 1st index element in the list that is always the root node
Heapsize: used to identify how many cells we had filled
Traversing: same as we did in BT using list
Insertion:
here first we insert the element at the heap_size index but once done we are gonna use the helper function heapify which helps us to make the tree in sorted order here basically we swap the newly inserted node with parent if it is smaller than parent_node(example considering min heap tree) and we continue to do recursively until we make the tree balance
Extraction:
we can only extract the root node
method: repace last element with root node, then compare rootnode with child and swap them in such a way(example having 2 childs, having one child, no child cases) that they meet the requirements of our Binary heap with given type

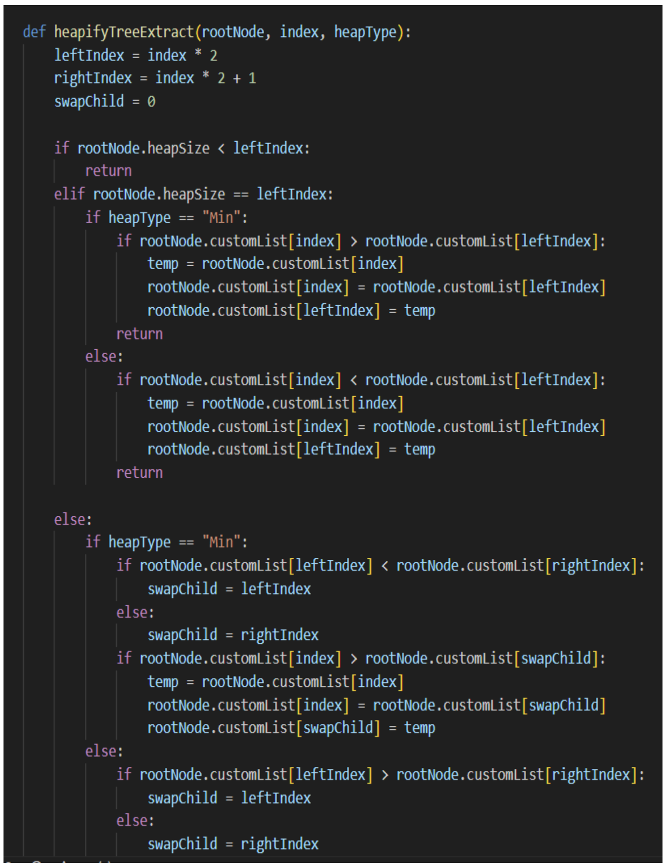
Deletion Entire: make entire to None



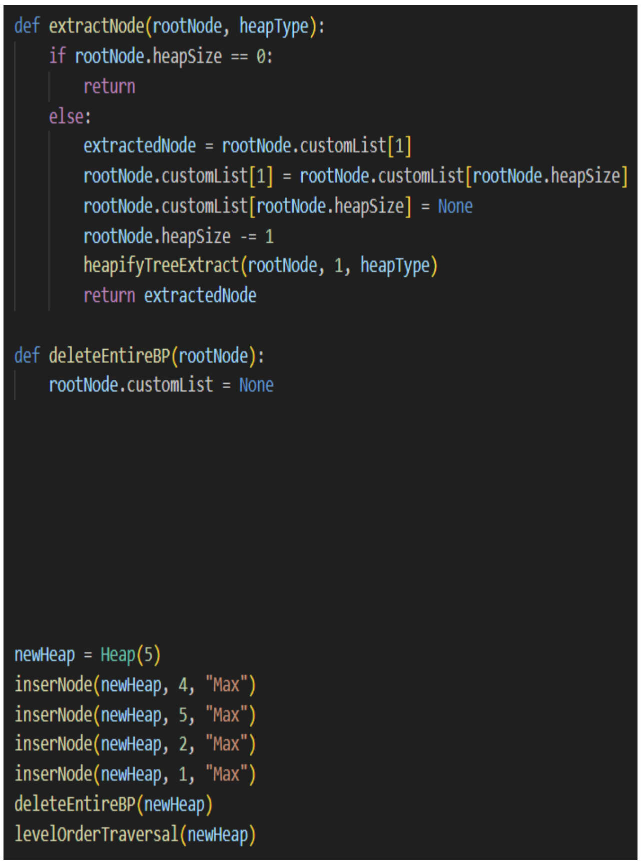
```python
def heapifyTreeInsert(rootNode, index, heapType):
    parentIndex = int(index/2)
    if index <= 1:
        return
    if heapType == "Min":
        if rootNode.customList[index] < rootNode.customList[parentIndex]:
            temp = rootNode.customList[index]
            rootNode.customList[index] = rootNode.customList[parentIndex]
            rootNode.customList[parentIndex] = temp
        heapifyTreeInsert(rootNode, parentIndex, heapType)
    elif heapType == "Max":
        if rootNode.customList[index] > rootNode.customList[parentIndex]:
            temp = rootNode.customList[index]
            rootNode.customList[index] = rootNode.customList[parentIndex]
            rootNode.customList[parentIndex] = temp
        heapifyTreeInsert(rootNode, parentIndex, heapType)

def inserNode(rootNode, nodeValue, heapType):
    if rootNode.heapSize + 1 == rootNode.maxSize:
        return "The Binary Heap is Full"
    rootNode.customList[rootNode.heapSize + 1] = nodeValue
    rootNode.heapSize += 1
    heapifyTreeInsert(rootNode, rootNode.heapSize, heapType)
    return "The value has been successfully inserted"
```



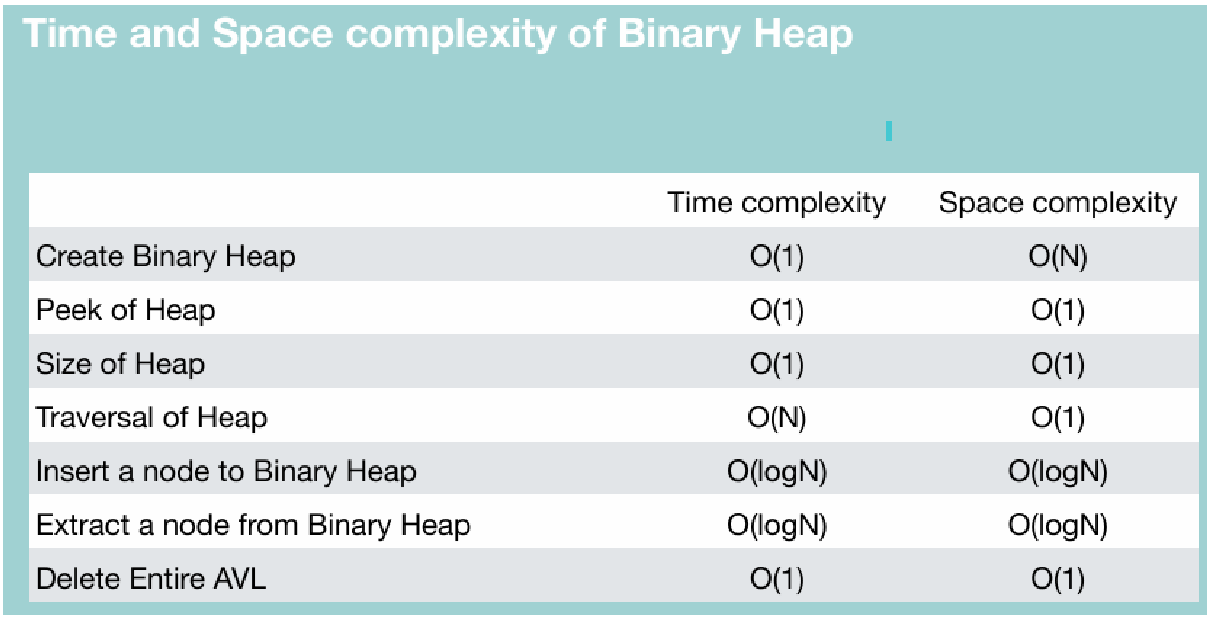
```python
class Heap:
    def __init__(self, size):
        self.customList = (size+1) * [None]
        self.heapSize = 0
        self.maxSize = size + 1

def peekofHeap(rootNode):
    if not rootNode:
        return
    else:
        return rootNode.customList[1]

def sizeofHeap(rootNode):
    if not rootNode:
        return
    else:
        return rootNode.heapSize
```



```python
def heapifyTreeExtract(rootNode, index, heapType):
    leftIndex = index * 2
    rightIndex = index * 2 + 1
    swapChild = 0

    if rootNode.heapSize < leftIndex:
        return
    elif rootNode.heapSize == leftIndex:
        if heapType == "Min":
            if rootNode.customList[index] > rootNode.customList[leftIndex]:
                temp = rootNode.customList[index]
                rootNode.customList[index] = rootNode.customList[leftIndex]
                rootNode.customList[leftIndex] = temp
            return
        else:
            if rootNode.customList[index] < rootNode.customList[leftIndex]:
                temp = rootNode.customList[index]
                rootNode.customList[index] = rootNode.customList[leftIndex]
                rootNode.customList[leftIndex] = temp
            return

    else:
        if heapType == "Min":
            if rootNode.customList[leftIndex] < rootNode.customList[rightIndex]:
                swapChild = leftIndex
            else:
                swapChild = rightIndex
            if rootNode.customList[index] > rootNode.customList[swapChild]:
                temp = rootNode.customList[index]
                rootNode.customList[index] = rootNode.customList[swapChild]
                rootNode.customList[swapChild] = temp
        else:
            if rootNode.customList[leftIndex] > rootNode.customList[rightIndex]:
                swapChild = leftIndex
            else:
                swapChild = rightIndex
```



```python
def extractNode(rootNode, heapType):
    if rootNode.heapSize == 0:
        return
    else:
        extractedNode = rootNode.customList[1]
        rootNode.customList[1] = rootNode.customList[rootNode.heapSize]
        rootNode.customList[rootNode.heapSize] = None
        rootNode.heapSize -= 1
        heapifyTreeExtract(rootNode, 1, heapType)
        return extractedNode

def deleteEntireBP(rootNode):
    rootNode.customList = None




newHeap = Heap(5)
inserNode(newHeap, 4, "Max")
inserNode(newHeap, 5, "Max")
inserNode(newHeap, 2, "Max")
inserNode(newHeap, 1, "Max")
deleteEntireBP(newHeap)
levelOrderTraversal(newHeap)
```



Time and Space complexity of Binary Heap

	Time complexity	Space complexity
Create Binary Heap	O(1)	O(N)
Peek of Heap	O(1)	O(1)
Size of Heap	O(1)	O(1)
Traversal of Heap	O(N)	O(1)
Insert a node to Binary Heap	O(logN)	O(logN)
Extract a node from Binary Heap	O(logN)	O(logN)
Delete Entire AVL	O(1)	O(1)

TRIE

Trie is a Tree based data structure that organize the information in a hierarchy

Properties:
1.mostly used to store or search strings
2.Any node in trie can store non repetative multiple characters
3.each node stores link to the next string and keeps track of end of string

Note: Each node can have multiple characters and each character can have multiple childs

Advantage: spell checker, auto completion trie will be helpful

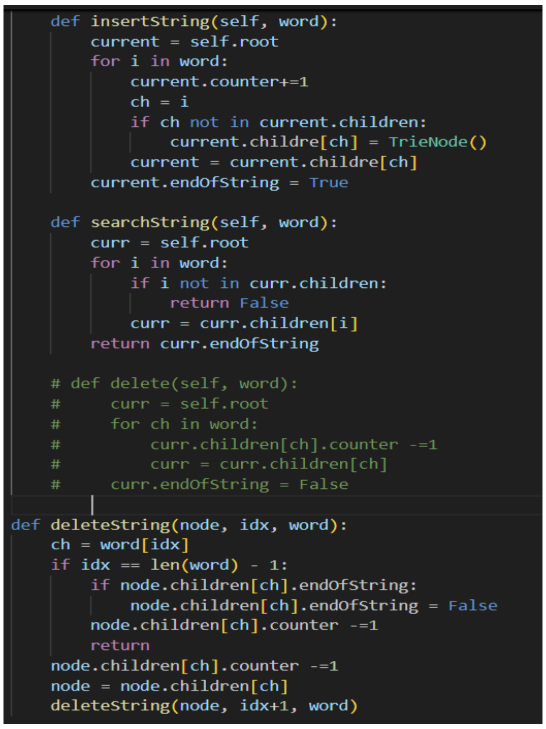
Operations:

<u>Creation:</u>
since we are creating node everytime and updating with character which has reference to the blank new trie node ,,,, we have attributes self.children={}(dictionary), self.endofstring which is False initially, and we have trie class which is just creating instance of trie node.
<u>Insertion:</u>
iterate the word check if character is already present if not update with character as key and blank trie node as value, and now update current node to this created trie node and repeat the process until word is completed
basically we are creating this structure : {A:{p:{k:node}}} (for string APK)
<u>Searching:</u>
here we traverse through the word and check each character by using get method it will return value of another dictionary if key is present else return False
Three cases:

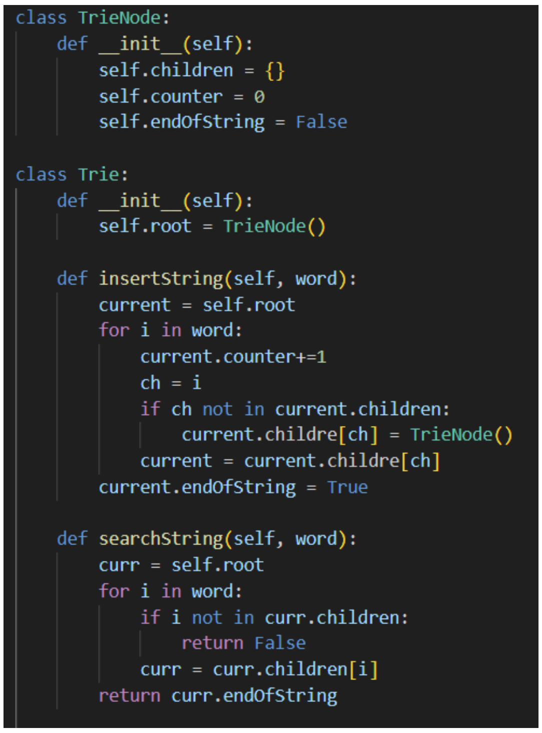
word is completely wrong, word is true, word is prefix which is also wrong

Deletion:

counter is increased if we insert 2 strings with same characters, if index is at end of word if word end of string is true we are making it false and counter is decreased check the code for more clarity



```
    def insertString(self, word):
        current = self.root
        for i in word:
            current.counter+=1
            ch = i
            if ch not in current.children:
                current.childre[ch] = TrieNode()
            current = current.childre[ch]
        current.endOfString = True

    def searchString(self, word):
        curr = self.root
        for i in word:
            if i not in curr.children:
                return False
            curr = curr.children[i]
        return curr.endOfString

    # def delete(self, word):
    #     curr = self.root
    #     for ch in word:
    #         curr.children[ch].counter -=1
    #         curr = curr.children[ch]
    #     curr.endOfString = False

def deleteString(node, idx, word):
    ch = word[idx]
    if idx == len(word) - 1:
        if node.children[ch].endOfString:
            node.children[ch].endOfString = False
        node.children[ch].counter -=1
        return
    node.children[ch].counter -=1
    node = node.children[ch]
    deleteString(node, idx+1, word)
```



```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.counter = 0
        self.endOfString = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insertString(self, word):
        current = self.root
        for i in word:
            current.counter+=1
            ch = i
            if ch not in current.children:
                current.childre[ch] = TrieNode()
            current = current.childre[ch]
        current.endOfString = True

    def searchString(self, word):
        curr = self.root
        for i in word:
            if i not in curr.children:
                return False
            curr = curr.children[i]
        return curr.endOfString
```

HASHING

The idea behind hashing is to allow large amount of data to be indexed using keys commonly created using formulas, Hashing is a method of sorting and indexing data

example: we will have the magic function which will convert string to numbers and in list in that obtained number from magic function we will store our original string

Terminology:

Hash function: It is a function that can be used to map of arbitrary size(string in example) to data of fixed size(list in example)

Key: Input data by a user

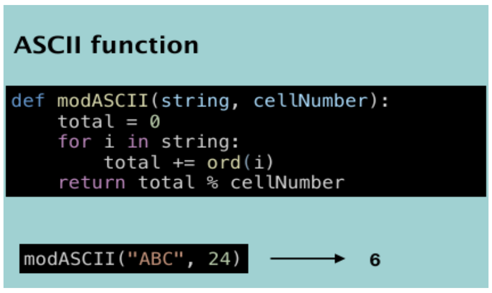
Hash value: A value that is returned by Hash Function(Magic Function in example)

Hash Table: It is a data structure which implements an associative array abstract data type(list in example), a structure that can map keys to hash values

Collision: A collision occurs when two different keys to a hash function produce the same output.

Properties of good hash Function:
1. It distributes hash values uniformly across hash tables(maximum avoiding of collision)
2. It has to use all the input data(every character in input data)

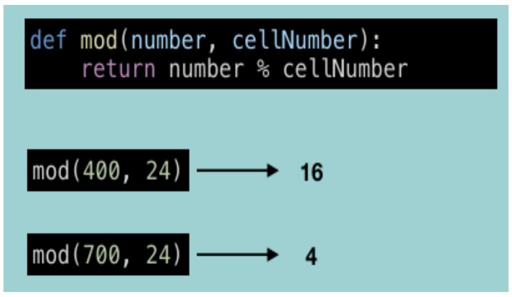
<u>Hash Function:</u>
Mod function(for integers interger % size of array gives the index)
ASCII function(ord() used to convert to ascii value, convert to ascii and then ascii value % size of array for index)



ASCII function

```
def modASCII(string, cellNumber):
    total = 0
    for i in string:
        total += ord(i)
    return total % cellNumber
```

```
modASCII("ABC", 24)  ⟶  6
```



```
def mod(number, cellNumber):
    return number % cellNumber
```

```
mod(400, 24)  ⟶  16

mod(700, 24)  ⟶  4
```

Colliding Resolving Techniques:

<u>1.Direct Chaining:</u>
we can use **linked list** for where we created new node and store physical location of new node in our array, and after this we can update the next reference whenever collision occurs.

<u>2.Open Addressing:</u>
open addressing seeks an alternative location within the table itself
<u>i) Linear Probing:</u>
If a collision occurs, the algorithm checks the next available slot in the List
<u>ii) Quadratic Probing:</u>
If a collision occurs, the algorithm checks the slots at intervals that increase quadratically
Ex: if the initial index is h, it checks h+1^2, h+2^2, h+3^2, h+4^2…etc.,
<u>iii)Double Hashing:</u>
Uses a second hash function to determine the interval between probes(step size for new index if previous index of first hash function is full)
Ex: If a key hashes to index 3 and index 3 is occupied, the second hash function determines the step size. If the second hash function returns 5, the algorithm checks index 8 (3 + 5), then index 13 (3 + 5 + 5), and so on

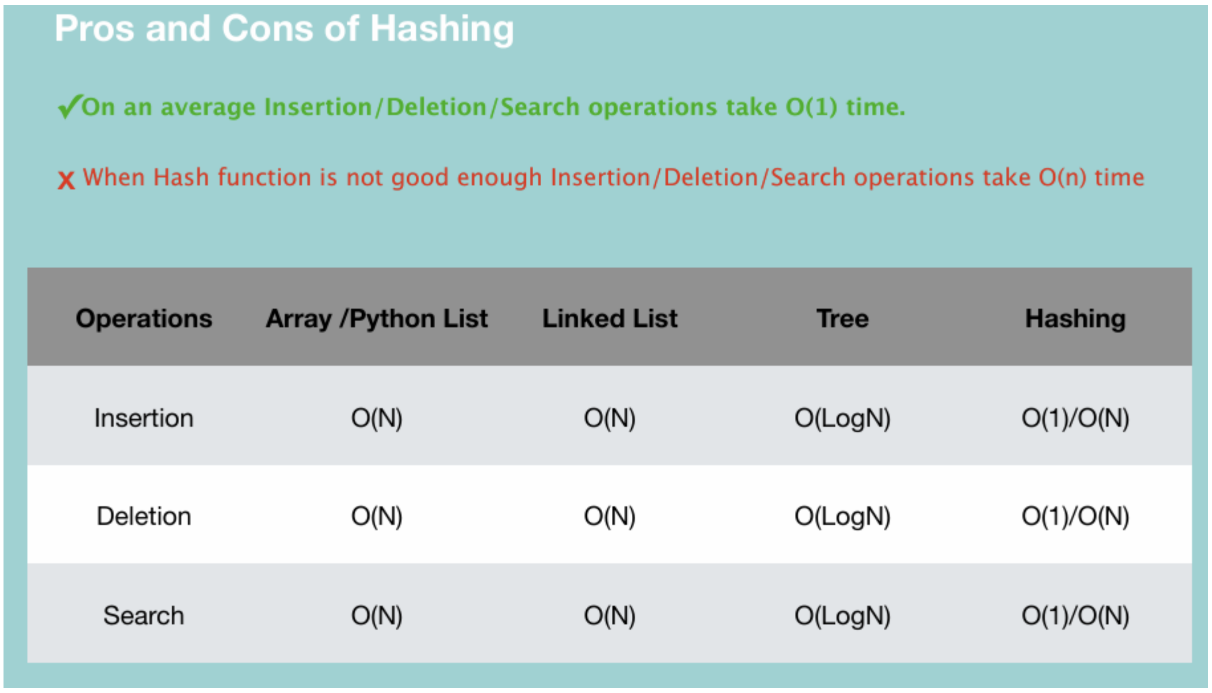
Note: there may be a case where hash table is full at this time we are gonna create new table with 2x size and we are gonna count hash values of entire string this is very time consuming process

Practical use: Password Verification,

Pros and Cons of Collision resolution techniques:

Direct chaining:
i)Hash table never gets full
ii)Huge Linked List causes performance leaks (Time complexity for search operation becomes O(n).)
Open addressing:
i)Easy Implementation
ii)When Hash Table is full, creation of new Hash table affects performance (Time complexity for search operation becomes O(n).)

Summary:
If the input size is known we always use "Open addressing"
If we perform deletion operation frequently we use "Direct Chaining



Pros and Cons of Hashing

✓On an average Insertion/Deletion/Search operations take O(1) time.

x When Hash function is not good enough Insertion/Deletion/Search operations take O(n) time

Operations	Array /Python List	Linked List	Tree	Hashing
Insertion	O(N)	O(N)	O(LogN)	O(1)/O(N)
Deletion	O(N)	O(N)	O(LogN)	O(1)/O(N)
Search	O(N)	O(N)	O(LogN)	O(1)/O(N)