Class groupLinkedList – Lista encadeada simples (unidirecional)

O método protegido **find** tem a complexidade O(n) no pior caso, pois é necessário percorrer toda a lista. Os métodos **delete** e **search,** que utilizam o **find,** tem a mesma complexidade O(n), já no melhor caso a complexidade é O(1), porque o método **isEmpty** avalia se a lista está vazia. Tem há um método push, que insere os dados em fila, portanto a complexidade é O(1).

```
def __find(self, goal):
    current = self.head
    parent = self
    while (current and goal != current.data.id):
        parent = current
        current = current.next
    return parent, current
```

```
def search(self, goal):
    if self.isEmpty(): print("Empty list")
    else: return self.__find(goal)
```

```
def __delete(self, goal):
    parent, node = self.search(goal)
    if parent is self:
        if node.next is None:
            self.head.data = None
            self.head.save()
        else:
            self.head = node.next
            self.save()
    else:
        parent.next = node.next
        parent.save()

def delete(self, goal):
    if self.isEmpty(): print("Empty list")
    else: return self.__delete(goal)
```

Traverse:

Traverse é um método recursivo que percorre a lista e retorna nextData (o nó) e node.data (o dado).

```
def __traverse(self, node):
    if node is None:
        return
    for nextData in self.__traverse(node.next):
        yield(nextData)
    yield(node.data)

def traverse(self):
    if self.isEmpty(): print("Empty list")
    else: return self.__traverse(self.head)
```

Class scheduleList – Lista duplamente encadeada (bidirecional)

O método **append** tem complexidade O(n) no pior caso, pois percorre a lista até que **current** aponte para **None,** já o melhor caso ocorre quando a lista está vazia, em que o método **isEmpty** retorna “Empty List”.

```
def append(self, new_data):
    if self.isEmpty():
        self.head.data = new_data
        self.head.save()
    else:
        parent = self
        current = self.head
        while current:
            parent = current
            current = current.next
        new_node = scheduleListNode(previous=parent, data=new_data, next=None)
        new_node.save()
        parent.next = new_node
        parent.save()
    self.save()
```

Traverse:

Tem a mesma função do primeiro traverse.

```
def __traverse(self, node):
    if node is None:
        return
    for nextData in self.__traverse(node.next):
        yield(nextData)
    yield(node.data)

def traverse(self):
    if self.isEmpty(): print("Empty list")
    else: return self.__traverse(self.head)
```

Class userBinarySearchTree – Árvore Binária não balanceada

Diferente do que ocorre em uma lista encadeada, em uma árvore binária o método **find** compara se a informação procurada "**goal**" tem valor menor ou maior que a informação presente no nó, após essa comparação, ele descerá um nível. Portanto, a complexidade no melhor caso é O(1), se a informação procurada estiver na raiz ou a lista estiver vazia e no pior caso é O(n).

```
def isEmpty(self):
    return self.root.data is None

def __find(self, goal):
    current = self.root
    parent = self

    while (current and goal != current.data):
        parent = current
        current = (current.left if goal.name < current.data.name else current.right)
    return parent, current

def search(self, goal):
    if self.isEmpty(): print("Empty tree")
    else: return self.__find(goal)
```

O método **insert** utiliza o **find** e o **isEmpty,** com isso a complexidade se mantém.

Traverse:

O método **traverse** é recursivo, ele “desce” tá o ultimo nó e “sobre” retornando em ordem, ou seja, LEFT – ROOT – RIGHT. A complexidade é O(n), pois depende do tamanho da árvore.

O mesmo acontece com o método traverseNode.

```
def traverse(self):
    if self.isEmpty(): print("Empty tree")
    else: return self.__traverse(self.root)

def __traverse(self, node):
    if node is None:
        return

    for leftData in self.__traverse(node.left):
        yield(leftData)

    yield(node.data)

    for rightData in self.__traverse(node.right):
        yield(rightData)
```

Promote:

O método **promote** tem complexidade O(n) no pior caso, já que a condição de busca é enquanto sucessor.left existir.

```
def __promote_sucessor(self, node):
    sucessor = node.right
    parent = node

    while sucessor.left:
        parent = sucessor
        sucessor = sucessor.left

    node.data = sucessor.data
    self.__delete(parent, sucessor)
```