1.

(a)

see the python script name hw3-pb1.py

(b)

Run the program and see the output. It will be printed out in the console.

(c)

There will m points be performed for the estimation. In this case, m = 300.

2.

(a)

Pseudo

```
def Tree-Find:
    arr = [a1 a2….am]
    Set root as the first p. (to the a1)
    Word_len = len(arr)
    for i in range(len - 1):
        if arr[i + 1] belongs to next pointer:
            go to the next char.
        elif isword:
            return the pointer.
        else:
            return NULL
```

Explanation:

I fist put the character we look for into the numpy array and find the length of it. Run a for loop. If there's the next pointer point to the next character in the array, we go further and if isword, which means that we have found the word we are looking for. Simply return the pointer of the current one. Note that this is the ending pointer unlike common one, which will point in the first element of an array. However, since it's a doubly linked list where there's always one pointer point to its parent, thus we can still find a way back to the root. If we can't find anything, return NULL meaning that the word is not in this tree.

2

(b)

Pseudo

```
def Tree-Delete:
    if Tree-Find(T,x) == NULL:
        return
    else:
        if IS-EMPTY: # meaning that it's leaf
            Delete(T,x)
            Search upward from the parent pointer
            if (parent node does not have white child) and (grey node)
                Delete
            else:
                return
        else: # internal node
            mark the node grey
```

Explanation:

In deletion, there will be divide into two cases. First is we delete the node in the leaf. We need to find it first, if there's no such node inside this tree, there is also no way for us to delete it. Otherwise, we use IS-EMPTY function to check if the node we want to delete is a leaf. If it is, we delete the node. Later we need to check if the node's parent need to be deleted, by checking if it's grey node and without any white child. If it is, there's no point for us to keep the parent node as well, thus delete it too.

The second case is that we find the inter node, since it's still useful to look for it's descendent, therefore all we need to do is to mark it grey.

2

(c)

The worst case will be when I got the parent node and search for the children that match my string. This will take $O(n)$ for my algorithm.

The worst case of my tree delete will be it's a super unbalanced tree which only have one path. In other words, every node only has one child. And say we want to delete the node in the leaf. We will need $O(n)$ runtime to find the node we want to delete and delete it. Since all the ascendant is useless right now. We will also need $O(n)$ runtime to delete them all. Therefore, $O(2n) => O(n)$

(d)

Store at the parent will be more efficient. For the internal node case, there's no difference since all we need to do is to mark it grey. But for the leaf case, we can use IS-EMPTY function, which is constant time to find if we have next node. Therefore, it won't waste our time finding it empty if it's leaf. Plus, when we need to search back to see what we need to delete, we can simply use parent pointer to traceback the path.

(e)

Since efficiency is w/n.

w= # words stored

n= # nodes

For the smallest efficiency, we will have a large number of n, but limited number of words. Therefore, the most inefficient tree will be we have n nodes but only till the leaf will form a word. The efficiency will be (1/n). In other words, it takes n node to form a word and there's no other internal node will form a word.

For the largest efficiency, we need every node can form a word. In other word is that every node is white node and has its meaning. Therefore the efficiency will be (n/n) = 1