NOTES

Parent Pointers

In computer science, an in-tree or parent pointer tree is an N-ary tree data structure in which each node has a pointer to its parent node, but no pointers to child nodes.

Advantages

It gives you the huge benefit that you can jump "up" one level (i.e. from a node to its parent) without remembering the parent node's address. Several algorithms (for example, getting the number of nodes between to two values) can be implemented very effectively and simple if the parent node of a node is known.

Disadvantage

The trade-off is the redundancy: if you modify the structure of the tree (for example by balancing the tree) you must remember to update both the left/right and the parent pointers to keep the consistency of the tree.

```
class newNode:
    def __init__(self, item):
        self.key = item
        self.left = self.right = None
        self.parent = None

def inorder(root):
    if root != None:
        inorder(root.left)
        print("Node :", root.key, ", ", end = "")
        if root.parent == None:
            print("Parent : NULL")
        else:
            print("Parent : ", root.parent.key)
        inorder(root.right)

def insert(node, key):
    if node == None:
        return newNode(key)
    if key < node.key:
        lchild = insert(node.left, key)
        node.left = lchild
        lchild.parent = node
    elif key > node.key:
        rchild = insert(node.right, key)
        node.right = rchild
        rchild.parent = node
    return node

root = None
root = insert(root, 50)
insert(root, 30)
insert(root, 20)
insert(root, 40)
insert(root, 70)
insert(root, 60)
insert(root, 80)
inorder(root)
```

Pattern matching

Problem: Given a text of n characters and a pattern of m characters such that n > m, write a function that prints all occurrences of pattern in text.

Input:

```
text = "THIS IS A PLAIN TEXT"
pattern = "TEXT"
Output:
Pattern found at index 16

Input:
text =  "AABAACAADAABAABA"
pattern =  "AABA"
Output:
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
```

```
def search(pat, txt):
    M = len(pat)
    N = len(txt)

    #iterate loop from 0 to N - M
    for i in range(0,N-M+1):
        for j in range(M):
            if txt[i + j] != pat[j]:
                #mismatch so break
                break;

        #if j is equal to M, mean pattern found
        if j == M-1:
            print(pat,"found at index",i)

txt = "AABAACAADAABAAABAA";
pat = "AABA";
search(pat, txt)
```

Complexity of above approach is O(m(n-m+1))

Better Approach: using Z-array
For a string (str) of n characters, Z array is of n element. An element Z[i] of Z array stores length of the longest substring starting from str[i] which is also a prefix of str[0..n-1]. The first entry of Z array is meaningless as complete string is always prefix of itself.

Example:

Index	0	1	2	3	4	5	6	7	8	9	10	11
Text	a	a	b	c	a	a	b	x	a	a	a	z
Z values	X	1	0	0	3	1	0	0	2	2	1	0

str = "aaaaaa"
Z = [x, 5, 4, 3, 2, 1]

str = "aabaacd"
Z = [x, 1, 0, 2, 1, 0, 0]

str = "abababab"
Z[] = [x, 0, 6, 0, 4, 0, 2, 0]

Main Concept
The idea is to concatenate pattern and text, and create a string "P$T" where P is pattern, $ is a special character should not be present in pattern and text, and T is text. Build the Z array for concatenated string. In Z array, if Z value at any point is equal to pattern length, then pattern is present at that point.

Example:
Pattern P = "aab", Text T = "baabaa"

The concatenated string is = "aab$baabaa"

Z array for above concatenated string is {x, 1, 0, 0, 0, 3, 1, 0, 2, 1}.
Since length of pattern is 3, the value 3 in Z array indicates presence of pattern.

```
def getZarr(str, Z):
    n = len(str)
    L,R = 0,0;
    for i in range(1,n):
        if i > R:
            L = R = i;
            while R<n and str[R-L] == str[R]:
                R+=1
            Z[i] = R-L;
            R-=1
        else:
            k = i-L
            if Z[k] < R-i+1:
                Z[i] = Z[k]
            else:
                # else start from R and check manually
                L = i;
                while R<n and str[R-L] == str[R]:
                    R+=1
                Z[i] = R-L
                R-=1

def search(text, pattern):
    concat = pattern + "$" + text
    l = len(concat)

    Z = [0]*l;
    getZarr(concat, Z);

    for i in range(0,l):
        if Z[i] == len(pattern):
            print("Pattern found at index",i - len(pattern) - 1 )

text = "All is well All is unwell";
pattern = "All";
search(text, pattern)
```

Edit Distance () Problem

Given two character strings s1 and s2, the edit distance between them is the minimum number of edit operations required to transform s1 into s2. Most commonly, the edit operations allowed for this purpose are:
(i) insert a character into a string
(ii) delete a character from a string and
(iii) replace a character of a string by another character

It is also called Levenshtein distance. In other words the Levenshtein distance between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. Each of these operations has unit cost.

For example, the Levenshtein distance between kitten and sitting is 3. A minimal edit script that transforms the former into the latter is:

kitten -> sitten (substitution of s for k)
sitten -> sittin (substitution of i for e)
sittin -> sitting (insertion of g at the end)

The Edit distance problem has an optimal substructure. That means the problem can be broken down into smaller, simple subproblems which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial.

Problem:Convert string X[1..m] to Y[1..n] by performing edit operations on string X.
Sub-problem: Convert substring X[1..i] to Y[1..j] by performing edit operations on substring X.

Case 1: We have reached the end of either substring
If substring X is empty, then we insert all remaining characters of substring Y to X and the cost of this operation is equal to number of characters left in substring Y.
('', 'ABC') --> ('ABC', 'ABC') (cost = 3)
If substring Y is empty, then we delete all remaining characters of X to convert it into substring Y. The cost of this operation is equal to number of characters left in substring X.
('ABC', '') --> ('', '') (cost = 3)

Case 2: Last characters of substring X and substring Y are same
If last characters of substring X and substring Y matches, nothing needs to be done. We simply recur for remaining substring X[0..i-1], Y[0..j-1]. As no edit operation is involved, the cost will be 0.
('ACC', 'ABC') --> ('AC', 'AB') (cost = 0)

Case 3: Last characters of substring X and substring Y are different
If the last characters of substring X and substring Y are different, then we return minimum of below three operations:

Insert last character of Y to X. The size of Y reduces by 1 and size of X remains the same. This accounts for X[1..i], Y[1..j-1] as we move in on target substring, but not in source substring.

('ABA', 'ABC') --> ('ABAC', 'ABC') == ('ABA', 'AB') (using case 2)

Delete last character of X. The size of X reduces by 1 and size of Y remains the same. This accounts for X[1..i-1], Y[1..j] as we move in on source string, but not in target string.

('ABA', 'ABC') --> ('AB', 'ABC')

Substitute (Replace) current character of X by current character of Y. The size of both X and Y reduces by 1. This accounts for X[1..i-1], Y[1..j-1] as we move in both source string and target string.

('ABA', 'ABC') - > ('ABC', 'ABC') == ('AB', 'AB') (using case 2)
It is basically same as case 2 where the last two characters matches and we move in both source string and target string except it costs edit operation.

$$\text{dist}[i][j] = \begin{cases} \max(i, j) & \text{when } \min(i, j) = 0 \\ \text{dist}[i - 1][j - 1] & \text{when } X[i-1] == Y[j-1] \\ 1 + \text{minimum } \{ \text{dist}[i - 1][j], \text{dist}[i][j - 1], \text{dist}[i - 1][j - 1] \} & \text{when } X[i-1] != Y[j-1] \end{cases}$$

```
def dist(X, Y):
        (m, n) = (len(X), len(Y))
        """
                for all i and j, T[i,j] will hold the Levenshtein distance between
    the first i characters of X and the first j characters of Y
    note that T has (m+1)*(n+1) values
        """
        T = [[0 for x in range(n + 1)] for y in range(m + 1)]

        # source prefixes can be transformed into empty by dropping all characters
        for i in range(1, m + 1):
                T[i][0] = i                    # (case 1)
    # target prefixes can be reached from empty source prefix by inserting every character
    for j in range(1, n + 1):
        T[0][j] = j             # (case 1)

        # fill the lookup table in bottom-up manner
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if X[i - 1] == Y[j - 1]:            # (case 2)
                    cost = 0                        # (case 2)
                else:
                    cost = 1                        # (case 3c)

                T[i][j] = min(T[i - 1][j] + 1,T[i][j - 1] + 1,T[i - 1][j - 1] + cost)

    return T[m][n]

X = "kitten"
Y = "sitting"
print("The Levenshtein Distance is", dist(X, Y))
```