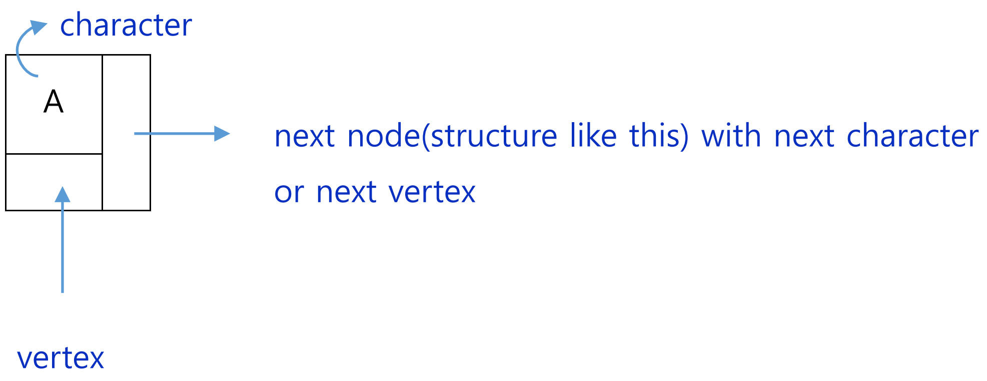
Introduction to Algorithms
Spring Semester 2017
HW11

Prof. Helmut Alt
Team 31. 20140174 Hyungrok Kim

Exercise 1

From class, we learned that in PATRICIA-trees shown as suffix trees,
if each edge has label of $\leq$n characters : space O(n^2)
(n: |w|, w: the main word)
suppose that strings on edges are constructed by <u>linked list</u> of characters.
So assume that labelling on edges of PATRICIA-trees is like this,



vertex will point to this kind of node and if there's more vertices to show in same edge, it will point to next node with next character on it. Last character will point to next vertex.
then the space complexity is O(md)
(we have $\{w_1, \ldots, w_n\}$ strings on the tree and $m = \sum_{1 \leq i \leq n} |w_i|,\ d = |\Sigma|$)

for search(w), we are looking for w, w=aw' ('a' is a character)
starting from the root, check what character it point to, if it is a, get into the node and start search(w') from that node, not root. If there's no pointer to a, then return "search fail". If we found all characters in w, check where the last place is pointing to, if it's pointing to a vertex, then return that vertex, if not, return "search fail"
it's like

```
search(w) is
        ssearch(w, root).
ssearch(w, s) is
        if w==empty string,
        if s is marked, return s.
        else, return "not found"
    check all pointers from s,
        if node(structure above) or vertex s' has a,
            ssearch(w', s') //(w=aw')
        else return "not found"
```

because it is implemented with linked list, checking all neighbors will take O(d) runtime, rest will be constant, and ssearch is done O(|w|) times
so the runtime of search is O(|w|d) ($d = |\sum|$)

for insertion, inserting w in tree.
Do search(w) and if it is found, then w is already in the tree.
else if search fail and ends in a vertex, not in an edge,

    then implement the suffix, which is not found so far, as the structure that is instructed

else if search fails in an edge, in the data structure to show characters in edge,

    then make a new vertex(not marked) pointed by current node, and make that vertex to point where current node was pointing, and construct leftover suffix after that vertex, as the structure that is instructed.

If the search was ended after finding w_1 ($w = w_1w_2$)
It will take O(|w_1|d) for search, and constructing rest w_2 will take O(|w_2|)
So insertion will take O(|w|d) runtime.

For deletion, deleting w in tree.
Do search(w), if found,

    If it has child vertices, then just remove marker on it.
    If it is a leaf, remove all vertices from v upward as long as they have

only one child and are not marked.

Deletion will take (|w|d) runtime.

Exercise 2

(a) Find the longest substrings of a string w that occur more than once.
We can construct the suffix tree of w$ ($: termination character) in $\theta(|w|)$ time by Ukkonen's algorithm.
Any node v are assigned the string of characters on path from root to v.
Assigned string from any internal node is occurring more than once in w, because it means the string in the node is on different positions of w.
Let's say that the depth of a node v is number of characters from root to v.
We can say that this problem is finding string assigned on internal node with deepest depth.
For finding it,

```
Construct suffix tree of w in Ukkonen's algorithm.
result = empty string
Traverse through every node, when current node is v,
    If v is an internal node,
        If (length of result)<(length of string assigned on v)
            result = string assigned on v
return result
```

it will take $\theta(|w|)$ time complexity.

(b) Find the shortest substrings of w that occur only once.
Because of the information from (a)'s answer, this question is finding assigned string from the leaf node with the smallest depth.
To do it,

```
Construct suffix tree of w in Ukkonen's algorithm.
result = empty string
Traverse through every node, when current node is v,
    If v is an leaf node,
        If (length of result)>(length of string assigned on v)
            result = string assigned on v
return result
```

it will take $\theta(|w|)$ time complexity.

(c) Find the longest substrings of w which are palindromes, i.e., read the same forward and backward.

Reverse w and name it w′, this problem can be said like finding longest common substring of w and w′, which is in same position in w.

Assume N=|w|, say that the longest common substring of w and w′ is x, and |x|=L, index of first character from w is i, and j from w′(which is (i+L-1) in w). Then if j=(N-1)-(i+L-1), we can say that x is in same position in w and it is the longest substring of w which is a palindrome.

j=(N-1)-(i+L-1) -> j=N-i-L

to find it,

construct suffix tree for string $w\$_1w'\$_2$,

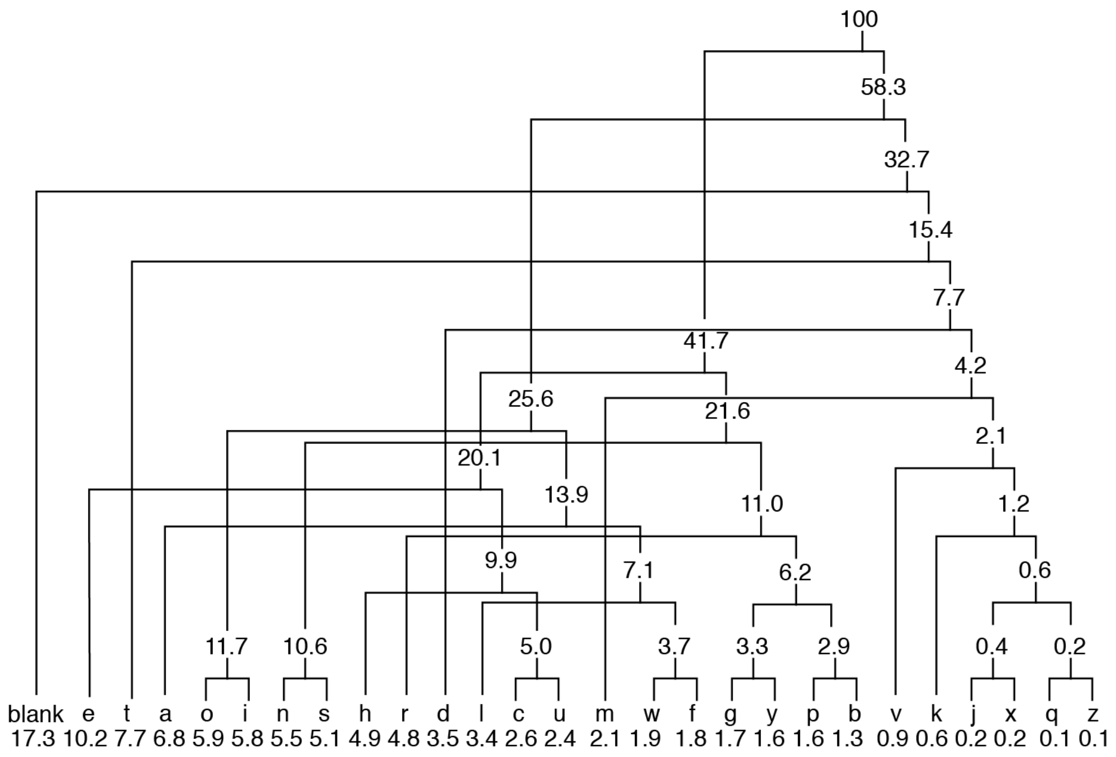
starting from the leaves mark each interior node by 1(2) if its subtree contains a leaf marked (i, 1) [if (j, 2) -> 2] for some, i,(j).

```
result = empty string
Traverse through every node, when current node is v,
    If v is labeled with 1 and 2,
        Try climb up one depth(deleting last character) ->(j++), until j=N-i-L
            If (length of result)<(length of corresponding string)
                result = corresponding string
```

Exercise 3

(a)

with probability distribution data, build heap for Huffman code like below,



suppose that right edge will have 1 and left edge 0,

then the optimum encoded code for all messages will be like

blank	110	r	0110	y	011101
e	000	d	11110	p	011110
t	1110	l	10110	b	011111
a	1010	c	00110	v	1111110
o	1000	u	00111	k	11111110
i	1001	m	111110	j	1111111100
n	0100	w	101110	x	1111111101
s	0101	f	101111	q	1111111110
h	0010	g	011100	z	1111111111

The expected number of bits per letter is

$$\sum(probability\ of\ the\ letter) * (length\ of\ Huffman\ code\ for\ the\ letter)$$

and it is computed to 4.171. the expected number of bits per letter=4.174

(b)

Let's say the characters are $a_1, a_2, \dots, a_{256}$ (numbered in ascending order of frequency)

The maximum character frequency is less than twice the minimum

$f(a_1) \le f(a_2) \le \cdots \le f(a_{256})$ and $f(a_{256}) \le 2f(a_1)$ (let's say f is return the frequency for a character or a node containing characters as descendant)

in the algorithm of building heap for Huffman code,

a_1 and a_2 will be joined to a new node, name it b_1.

Any sum of two $f(a_i) + f(a_j) \ge f(a_1) + f(a_2) = f(b_1) \ge 2f(a_1) \ge f(a_{256})$

none of the b_i nodes will be joined before a_{256} and a_{255} is joined to b_{128} because b_i is always bigger than a_{256}, and after that we have nodes

$b_1, b_2, \dots, b_{128}$

and $f(b_1) \le f(b_2) \le \cdots \le f(b_{128})$

we can see that $f(b_1) = f(a_1) + f(a_2)$ and $f(b_{128}) = f(a_{255}) + f(a_{256})$ and,

$2f(a_1) \ge 2f(a_1) \ge f(a_{256}) \ge f(a_{255})$

$2f(a_1) + 2f(a_2) \ge f(a_{256}) + f(a_{255})$

$2f(b_1) \ge f(b_{128})$,

this means in 128 nodes $b_1, b_2, \dots, b_{128}$ the maximum frequency is less than the twice the minimum frequency. This is the same condition from start with 256 changed to 128.

We can do the same process 7 times until there's the root node.

Than from the heap, we encoded Huffman code length will be 8-bit to all of the characters. This means the Huffman coding doesn't change the expected length of characters.

So it proves that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.