CSCI 6057 Advanced Data Structures - Assignment 4

Yansong Li
B00755354
yansong.li@dal.ca

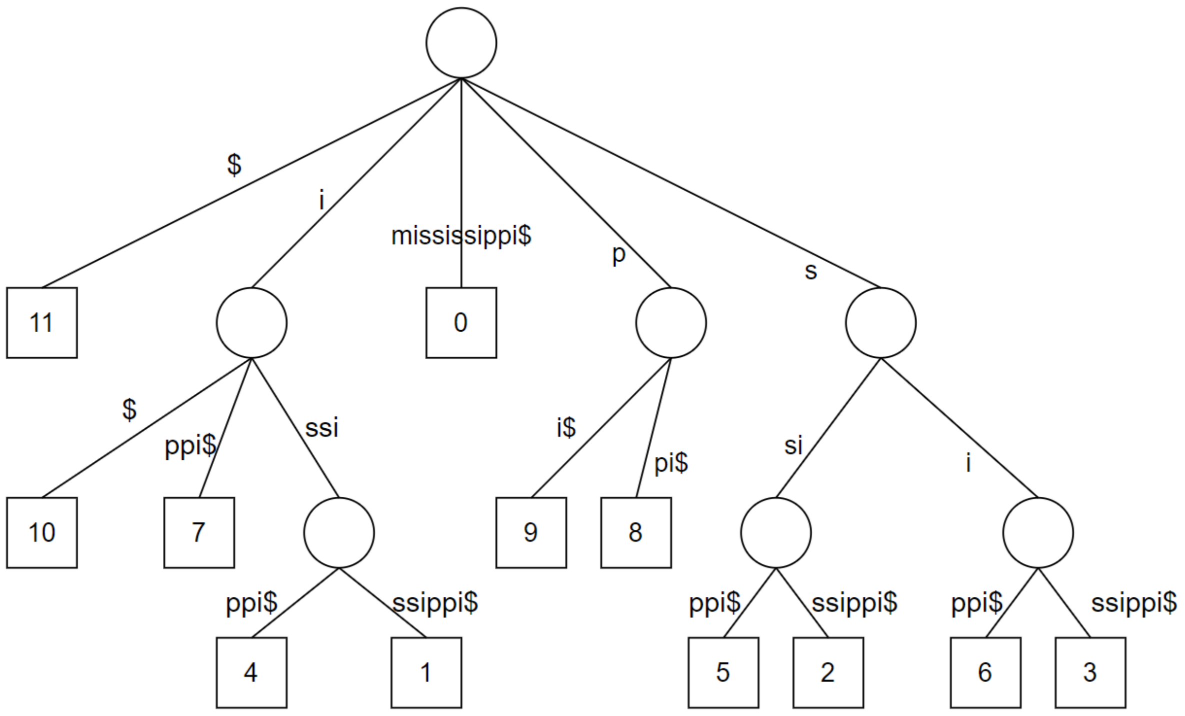
Question 1. Draw the suffix tree for the string Mississippi

T$: mississippi$

Suffixes:

0	1	2	3	4	5	6	7	8	9	10	11
mississippi$	ississippi$	ssissippi$	sissippi$	issippi$	ssippi$	sippi$	ippi$	ppi$	pi$	i$	$

Suffix Tree:



Question 2. Prove that $T(n) = O(\log_2 \min\{i, n-i+1\}) + T(i-1) + T(n-i) = O(n)$

We know that $T(0)$ is undefined because a tree must have at least one node. Also $T(1) = T(2) = 0$ because any binary search tree with 1 or 2 nodes is an optimal binary search tree, so the constructing time is 0. Hence, we use a base case of $T(3)$.

Also, we notice that the worst case of $O(\log_2 \min\{i, n-i+1\})$ is when $i = \frac{n}{2}$; therefore, $O(\log_2 \min\{i, n-i+1\}) \le O(\log_2 \frac{n}{2}) \le O(\log_2 n) \le c(\log_2 n)$. For base case we choose $n = 3$ and $i = n/2 = 2$.

Base case:

$$T(3) = O(\log_2 \min\{2,3-2+1\}) + T(2-1) + T(3-2)$$
$$T(3) = O(\log_2 2) + T(1) + T(1)$$
$$T(3) = 1 + 0 + 0$$
$$T(3) = 1$$

Therefore, it holds $T(n) = O(n)$.

Induction step:
Next, we guess $T(n) = O(n)$, which means $T(n) \leq cn$ for $\exists c > 0, \forall n \geq n_0$. Next, we assume that it is true for all $m < n$.

Therefore, we are assuming $T(m) \leq cm$ for $\forall m < n$. Then, let $n - i = m_1$ and assume $T(n-i) \leq c(n-i)$ where d is a constant, which holds the inequality $m < u$ (since $i \geq 1$, $n - i < n$). Meanwhile, let $i - 1 = m_2$ and assume $T(i-1) \leq c(i-1)$, which holds the inequality $m < u$ (since $i \leq n$, $i - 1 < n$). Hence,

$$\begin{aligned} T(n) &= O(\log_2 \min\{i, n-i+1\}) + T(i-1) + T(n-i) \\ &\leq O(\log_2 n) + c(i-1) + c(n-i) \\ &\leq c(\log_2 n) + c(i-1) + c(n-i) \\ &\leq c(\log_2 n) + ci - c + cn - ci \\ &\leq c(\log_2 n) + cn - c \\ &\leq c(n + \log_2 n - 1) \\ &= O(n + \log_2 n - 1) \end{aligned}$$

Since n dominates $\log_2 n$ and 1. Hence $O(n + \log_2 n - 1) = O(n)$. Therefore, we have proved that $T(n) = O(n)$.

Question 3.
Algorithm: This algorithm is based on the longest repeated substring problem that we talked about in class and building a generalized suffix tree that includes both A and B. Specifically, we add a delimiter # for A and a delimiter \$ for B, then concatenate them together. Hence, we would have a new string S that formed as $A\#B\$$ with length $m + n + 2$. Next, we build a suffix tree T of $A\#B\$$. After we have built T, we travel T from leaf to root and mark each internal node of T as A, B, or AB, which indicate it has leaf node only belong to A, only belong to B, or belong to A and B both in its subtree respectively. Next, we perform a depth first search on T to find the internal node that marked as AB and has the maximum "letter" depth (the number of characters on the edges). Then the characters on the path from root to the internal node with the largest "letter" depth that marked as AB is the longest common substring of A and B.

Analysis:
First, we know that we can constructure a suffix tree of a string with length n over a constant-size alphabet in $O(n)$ time. Hence, we can build the suffix tree T of $A\#B\$$ in $O(m+n)$ time. Second, T has $m + n + 2$ leaf nodes, and its number of internal nodes is $\leq m + n + 1$ since suffix tree is a compressed tris. Therefore, T has $\leq 2(m + n + 1)$ edges (one edge for each

node except the root). Hence, the time complexity of marking the internal nodes in T by a leaf to root traversal is $O(m+n)$. Third, we know that the time complexity of depth-first search on a graph with V nodes is $O(V)$. For the suffix tree T of $A\#B\$$, the total number of nodes is $\leq 2(m+n)+3$. Hence, the time complexity of depth-first search on T is also $O(m+n)$. Therefore, the total time we used to find the common substring of A and B is $O(m+n)+O(m+n)+O(m+n)=O(m+n)$. As a result, the time complexity of this entire algorithm is $O(m+n)$.

Question 4.

(i) Show that E occupies $O(\sqrt{n}\log_2\log_2 n)$ bits.

The length of each possible bit vector is $\frac{1}{2}\log_2 n$ and each entry of a bit vector can be either 1 or 0. Therefore, the number of bit vectors of length $\frac{1}{2}\log_2 n$ is $2^{\frac{1}{2}\log_2 n}$, which is

$$2^{\frac{1}{2}\log_2 n} = 2^{\log_2 n^{\frac{1}{2}}} = n^{1/2} = \sqrt{n}$$

Hence, there are $\sqrt{n}$ possible bit vectors of length $\frac{1}{2}\log_2 n$, which means there are $\sqrt{n}$ rows of lookup table E. Also, table E only has one entry for each possible bit vector, which means there is only one column of E. Meanwhile, it takes $\log_2\frac{1}{2}\log_2 n$ bits to store each entry in E because it records the number of 1s in a bit vector of with length $\frac{1}{2}\log_2 n$. Hence, the table E occupies $\sqrt{n} * 1 * \log_2\frac{1}{2}\log_2 n$ bit in total, if we represent it using big-O notation, it is $O(\sqrt{n}\log_2\log_2 n)$ bits.

(ii) Show how to use the resulting set of data structures to support rank in constant time.

For a rank query r of element i, we first get the cumulative rank of the super blocks r_1 before the superblock that i is located, which can be found in the first data structure that stores the cumulative ranks of superblocks in the starting position of each superblock. hence, it takes $O(1)$ time.

Next, we get the relative cumulative rank of blocks to this superblock r_2 before the block that i is located, which can be found in the second data structure that stores the cumulative ranks of blocks in the starting position of each block. hence, it also takes $O(1)$ time.

Then, we get the number of 1s within the block r_b that i is located, which can be found in the lookup table E. Hence, it takes $O(1)$ time. Next, we use the size of E times $\frac{1}{2}\log_2 n$ then modulus $\frac{1}{2}\log_2 n$ to obtain x, which is $\left(\sqrt{n} * \frac{1}{2}\log_2 n\right) mod\ \frac{1}{2}\log_2 n = x$. Then the number of 1s of the first x bits of r_b is the relative rank r_3 of i to this block, which also takes $O(1)$ time since it just bit operations.

Finally, the rank of i, $r = r_1 + r_2 + r_3$. The total time that we used to find $r = O(1) + O(1) + O(1) + O(1) = O(1)$. Therefore, the resulting set of data structures to support rank in constant time.