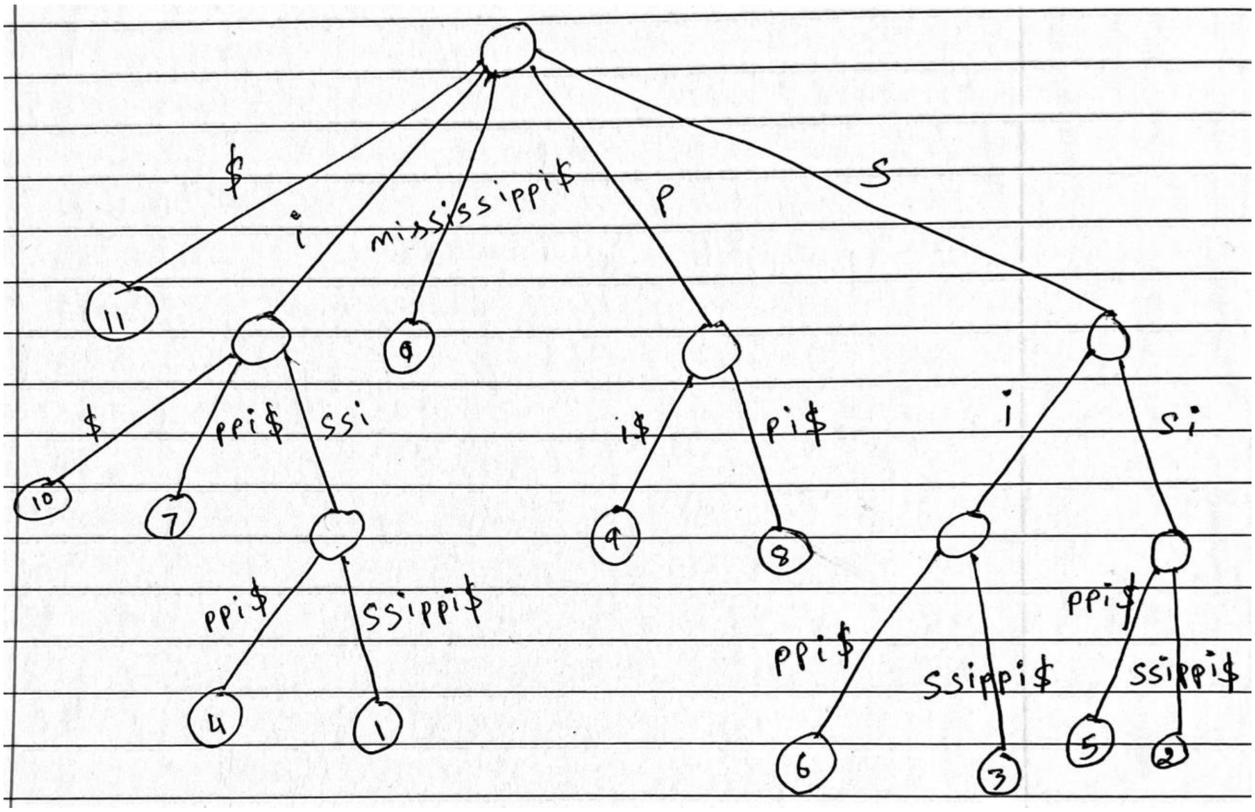
1. [10 marks] Draw the suffix tree for the string `mississippi`. Append a \$ (the end of file symbol) to the end of the string when drawing the suffix tree.

Solution:



2. [15 marks] In class, we defined entropy over a finite set of objects, each associated with a probability.

It is also possible to define entropy over an infinite set of objects. For example, if the set of objects is the set of natural numbers $\{1, 2, 3, \ldots\}$, and, in the probability distribution D, p_i is the probability associated with integer i, then the entropy of D is

$$H(D) = \sum_{i=1}^{\infty} p_i \lg(1/p_i)$$

Now, let us define a discrete distribution D as follows. Consider the following process: We keep tossing a fair coin until the first head occurs; the number of times we toss a coin is a natural number. In the probability distribution D, the probability p_i associated to number i is the probability of tossing the coin exactly i times before we stop this process, i.e., the first $i-1$ tosses all result in a tail, and the i-th toss gets a head.

Your tasks is to compute $H(D)$. Show your steps.

The entropy of D is

$$H(D) = \sum_{i=1}^{\infty} P_i \log(1/P_i)$$

For the discreate distribution D, defined in the question,

P_i = probability of getting head on the i-th toss.

The probability of getting i-1 tails followed by 1 head is $(1/2)^i$, since each toss of a fair coin is independent and has probability 1/2

$$H(D) = \sum_{i=1}^{\infty} (1/2)^i \log(2^i)$$

$$H(D) = \sum_{i=1}^{\infty} (1/2)^i \left(i \log(2)\right)$$

$$= \log 2 \sum_{i=1}^{\infty} i \, (1/2)^i \quad \text{—— ①}$$

$\sum_{i=1}^{\infty} i\,(1/2)^i$ is a Sum of an arithmetic geometric Series.

Further Simplying this ①

The Standard geometric Series formula

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$$

for $|x| < 1$, we can differentiate on both sides

R.H.S $\quad \frac{d}{dx} = \frac{1}{1-x} \qquad \therefore \frac{d}{dx}\left(\frac{u}{v}\right) = \frac{v(u') - u(v')}{v^2}$

$u = x \quad v = 1 - x$

$$= \frac{(1-x)(1) - x(-1)}{(1-x)^2}$$

$$= \frac{1 - x + x}{(1-x)^2}$$

$$= \frac{1}{(1-x)^2} \quad \text{—— ②}$$

L.H.S

$$\frac{d}{dx}\sum_{i=1}^{\infty} x^{i}$$

$$= \sum_{i=1}^{\infty} \frac{d}{dx} x^{i} \qquad \because \frac{d}{dx} x^{n} = n x^{n-1}$$

$$= \sum_{i=1}^{\infty} i x^{i-1} \quad \text{——— } (3)$$

From (2) & (3)

$$\sum_{i=1}^{\infty} i x^{i-1} = \frac{1}{(1-x)^{2}} \quad \text{——— }$$

multiplying x on both sides

$$\sum_{i=1}^{\infty} i x^{i} = \frac{x}{(1-x)^{2}} \quad \text{——— } (4)$$

Substitute (4) in (1) here $x = \frac{1}{2}$

$$= \log 2 \left[\frac{1}{2} \left(\frac{1}{\left(1 - \frac{1}{2}\right)^2} \right) \right]$$

$$= \log 2 \left(\frac{1}{\cancel{2}} \times \cancel{4} \right)$$

$$= 2 \log 2$$

$$= 2 \times 1$$

$$= 2$$

$H(D) = 2$

3. [10 marks] Let T be an arbitrary splay tree storing n elements $A_1, A_2, \ldots, A_n$, where $A_1 \leq A_2 \leq \ldots \leq A_n$. We perform n search operations in T, and the ith search operation looks for element A_i. That is, we search for items $A_1, A_2, \ldots, A_n$ one by one.

   (i) [5 marks] What will T look like after all these n operations are performed? For example, what will the shape of the tree be like? Which node stores A_1, which node stores A_2, etc.?

Solution:

For a splay tree, after performing a search operation, the element being searched for is splayed to the root of the tree through a series of tree rotations.

If we search for elements A_1, A_2, …. A_n one by one in ascending order, each search operation will bring A_i to the root. Since each element A_i is less than $A_{i+1}$and splay trees maintain the binary search tree property, after each search operation, A_i will become the left child of A_{i+1}once A_{i+1} is splayed to the root.

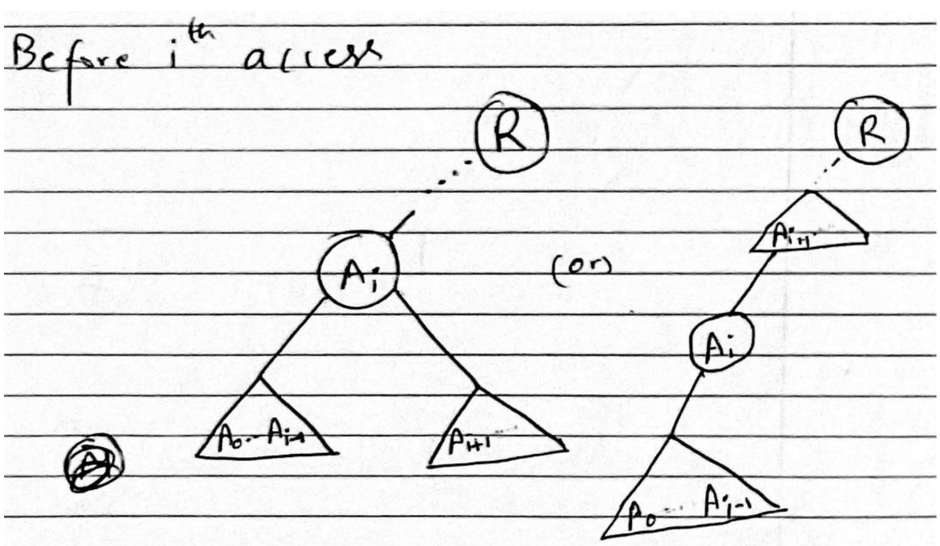


Figure 1: possible splay tree structure before i-th access

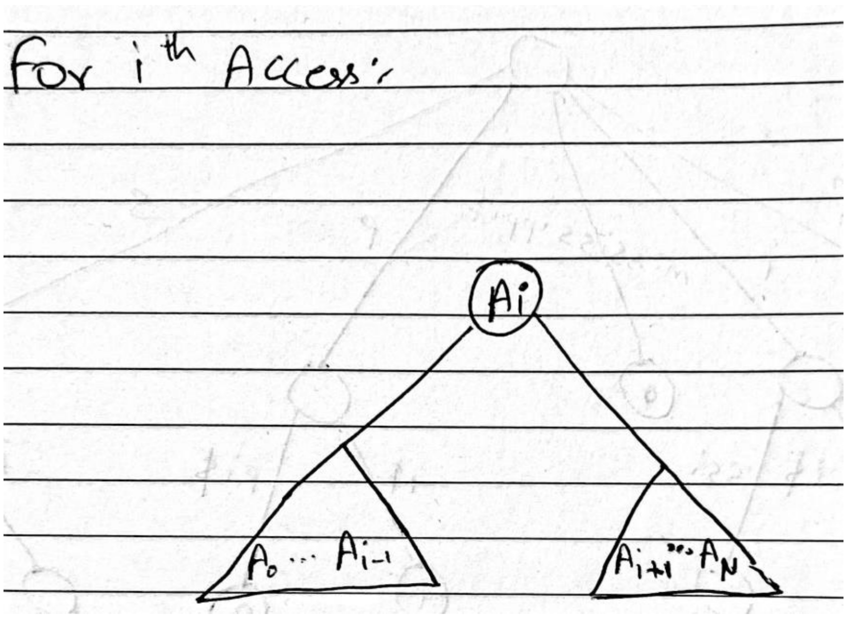


Figure 2: Splay tree structure at i-th access.

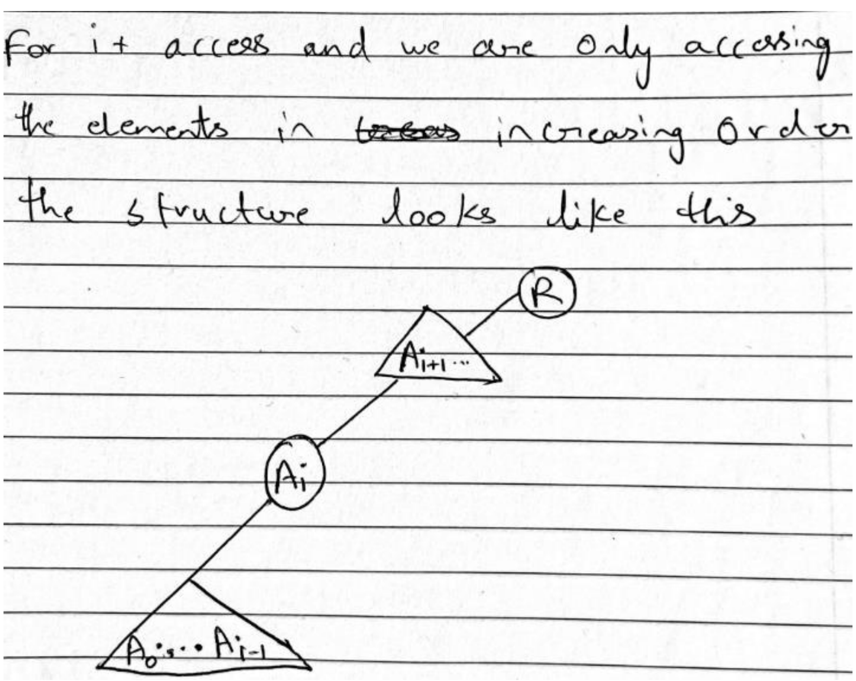


Figure 3: Splay tree structure for all i+ accesses

After searching for all elements in order, the final tree *T* will have A_n as its root, A_{n-1} as the left child of A_n, A_{n-2} as the left child of A_{n-1}, and so on, until A_1 which will be the left child of A_2. The tree will be like a linked list with all nodes having only left children and no right children.

(ii) [5 marks] Prove the answer you gave for (i) formally. Your proof should work no matter what the shape of T was like before these operations.

Hint: It may help to start with some specific examples and try to make some observations to make a guess. Then, construct a proof by induction.

Solution:

Proof:

By mathematical induction

Base case:-

when $n=1$, there's only one element A_1 in the tree T, and its the root of the tree, which is degenerated with only one node

A_1

Inductive Step:-

Assume for some k where $k<n$ after searching for $A_1, A_2, \dots A_k$, the tree is a degenerated tree with A_k as the root A_{k-1} as the left child of A_k and so on, down till A_1.

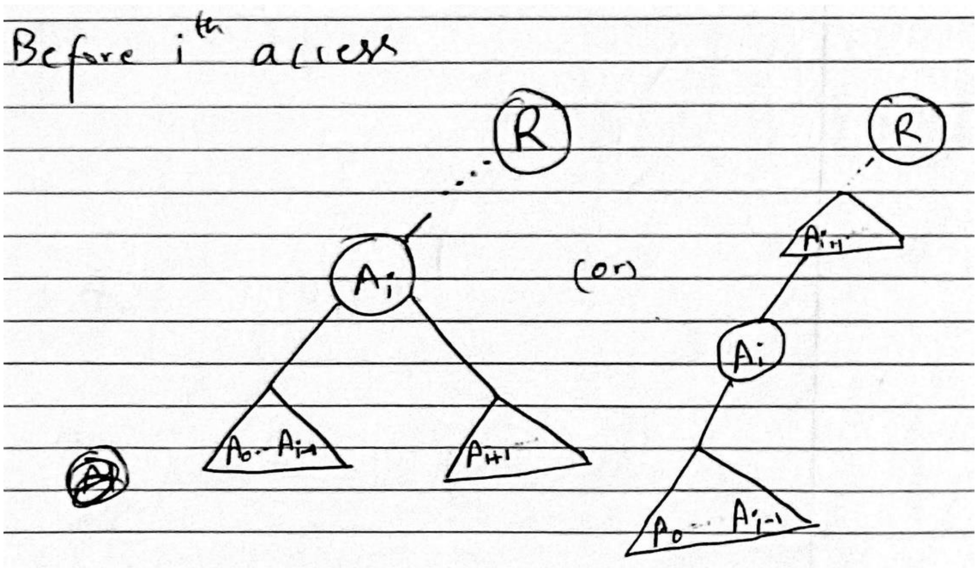


Figure 4: possible splay tree structure before i-th access

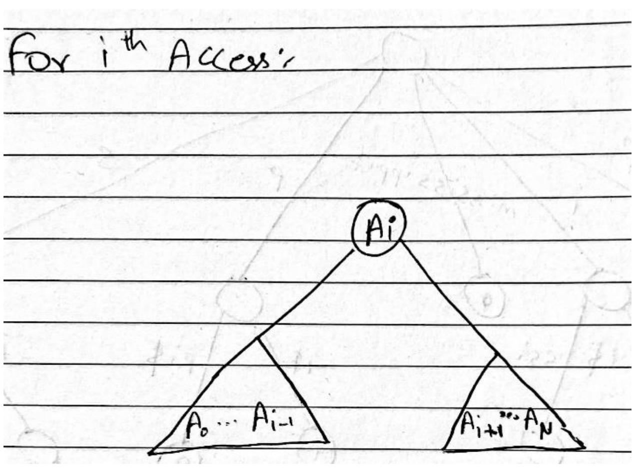


Figure 5: Splay tree structure for i-th accesses

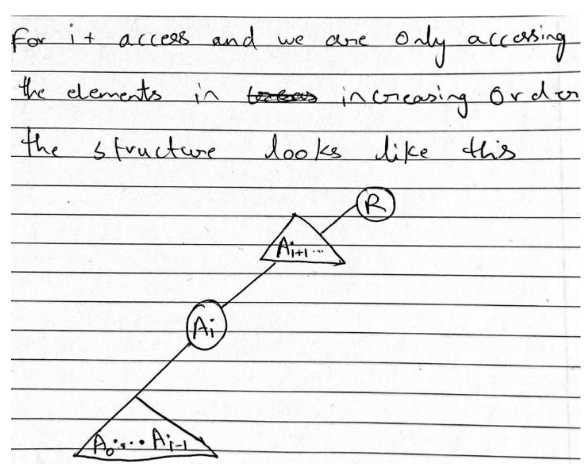


Figure 6: Splay tree structure for all i+ accesses

Now, Consider the Search operation for A_{k+1} Since A_{k+1} is greater than all $A_1, \ldots A_k$ it must be in the right ~~Sequence~~ Subtree of A_k in the tree before the Search. As A_{k+1} is Splayed to the root, all elements that were on the path from the root to A_{k+1} will be moved closer to the root. But ~~bo~~ because of the inductive hypothesis, all A_i for $i \leq k$ are already as close to the root as possible.

Therefore A_{k+1} will become the new root, with A_k as its left child and because of the access sequence the tree will be degenerate tree.

By the principal of Mathematical induction the Statement is true for all n

4. [15 marks] Given a string S of length n over a constant-sized alphabet and a number k, we wish to find the shortest substring of S that occurs in S exactly k times. Design an algorithm to solve this problem in $O(n)$ time. Show your analysis of the running time. You are not required to give pseudocode, but feel free to give pseudocode if it helps you explain your algorithm.

   Hint: You can use the result that a suffix tree for a string of length n over a constant-sized alphabet can be constructed in $O(n)$ time.

Solution:

Pseudo-code:

```
FindShortestkFrequencySubstring(S,k):-

    T ← ConstructSuffixTree(S)

    AnnotateDescendantTree(T.root)

    resultNode ← NULL
    minLength ← ∞
    DFS(T.root, 0)          :: DFS with Depth 0

    if resultNode is not NULL
        return ExtractString(T, resultNode)
    else
        return "No Such Substring"
```

Figure 4: Pseudocode for Find the shortest frequency substring

```
DFS (Node, depth)
    if node is a leaf
        return

    if node.descendantCount = k & depth < minLen
        resultNode ← node
        minLen ← depth

    for each child in node.Children
        DFS (child, depth + EdgeLength(node, child))
```

Figure 5: Pseudocode for DFS

```
ExtractSubstring (T, node)
    Substring ← ""
    while node is not T.root
        Substring ← EdgeLabel (node.parent, node)
                    + Substring
        node ← node.parent
    return Substring
```

Figure 6: Pseudocode for extract substring

Description:

Build a Suffix Tree: Construct a suffix tree T for the string S in O(n) time.

Annotate Suffix Tree: Traverse the suffix tree and annotate each internal node with the number of leaf descendants it has. This count will tell us how many times the substring corresponding to the path from the root to this node appears in S.

Find Eligible Nodes: Perform a depth-first search (DFS) on T to find all internal nodes that correspond to substrings occurring exactly k times. This can be done in O(n) time as each node is visited once.

Determine Shortest Substring: Among all the nodes found in step 3, find the node that corresponds to the shortest substring. This can be done during the DFS by keeping track of the depth of each node and selecting the node with the desired count that has the smallest depth.

Extract Substring: Once the correct node is found, retrieve the substring by traversing the path from the root to this node.

Algorithm Analysis:

Building the suffix tree takes O(n) time.

Annotating each node with the number of descendants can be done in O(n) time by summing the counts during the DFS.

Finding the internal nodes with exactly k descendants also takes O(n) time since each node is visited only once during the DFS.

Determining the shortest substring is done during the DFS without adding extra time complexity.

Extracting the substring is linear with respect to the length of the substring, which is at most n, so it also fits within O(n) time.

Since all these steps are sequential and each takes O(n) time, the overall time complexity of the algorithm remains O(n).