String Matching With Suffix Trees

Sagar Kumar (MA24BTECH11020)
Krish Agarwal (MA24BTECH11014)

Table of Contents

Problem Statement

Suffix Trees and DNA Matching

Given a text string T and a set of text patterns $P = \{p^1, p^2, \ldots, p^r\}$, often it is important to find all occurrences of any pattern P in T.

For instance, in molecular biology, where a relatively stable library of interesting or distinguishing DNA or protein substrings have been constructed, when a new DNA sequence S is generated, one wishes to see if, how many and where the occurrences of the known substrings occur in S.

- Let us adopt the convention:
  - n be the combined length of all the patterns in P.
  - Let there be k occurrences of any of the patterns in the string.
  - Let length of string T be m.

Naïve String Matching

Naïve String Matching

Pseudocode:

- **Input:** A text string *T* of length *n* and a pattern string *P* of length *m*.
- **Iterate:** Loop through the text *T* from index *i* = 0 to *i* = *n* - *m*.
- **Compare:** For each index i:
  - Check if the first character matches (T[i] == P[0]).
  - If it matches, run an inner loop to check the remaining *(m - 1)* characters.
- **Mismatch:** If a mismatch is found at any point in the inner loop, break the loop immediately and advance to the next starting position *(i + 1)*.
- **Match:** If the inner loop completes without mismatch, store/print the index *i* (Pattern found).

Naïve Algorithm Time Complexity

- **Exact Complexity:** The exact number of comparisons is proportional to (n - m + 1) x m.
- **Big-O Notation:** Therefore, the worst-case time complexity is O((n - m + 1) x m).
- **Approximation:** Since typically n>>m (the text is much larger than the pattern), we approximate the complexity as **O(m x n)**.



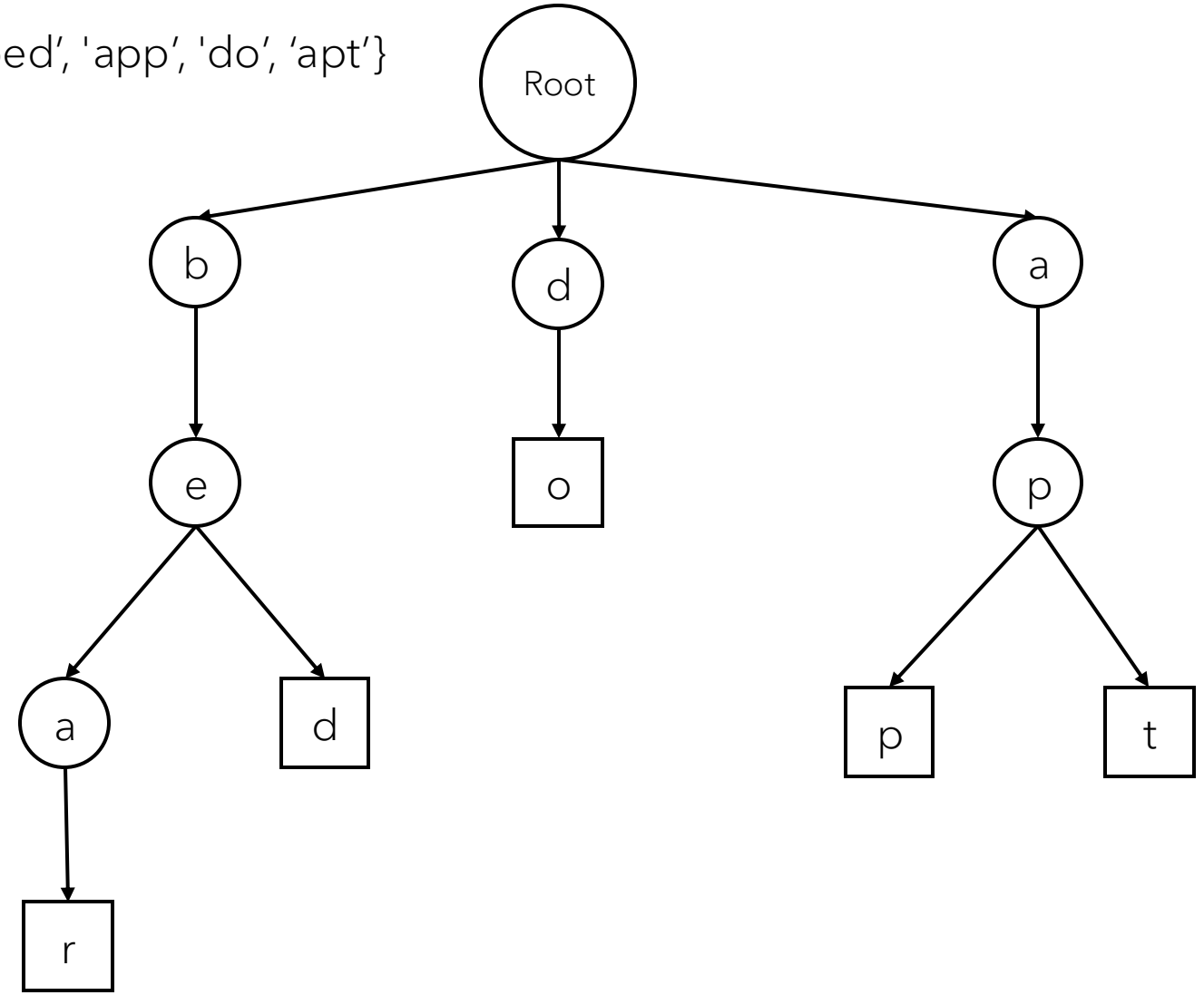
Tries
Compressed Tries

Tries (from re’trie’val):

- A Trie is an ordered tree that stores a set of keys represented as strings. It maintains the strings of words.
- The pattern matching algorithm looks for the i^{th} letter of the substring on the i^{th} child of the tree along the path.
- That way, we conclude by at most searching the number of nodes equal to the length of the pattern *m*.

Example:

List of words = {‘bear’, ‘bed’, 'app’, 'do’, ‘apt’}

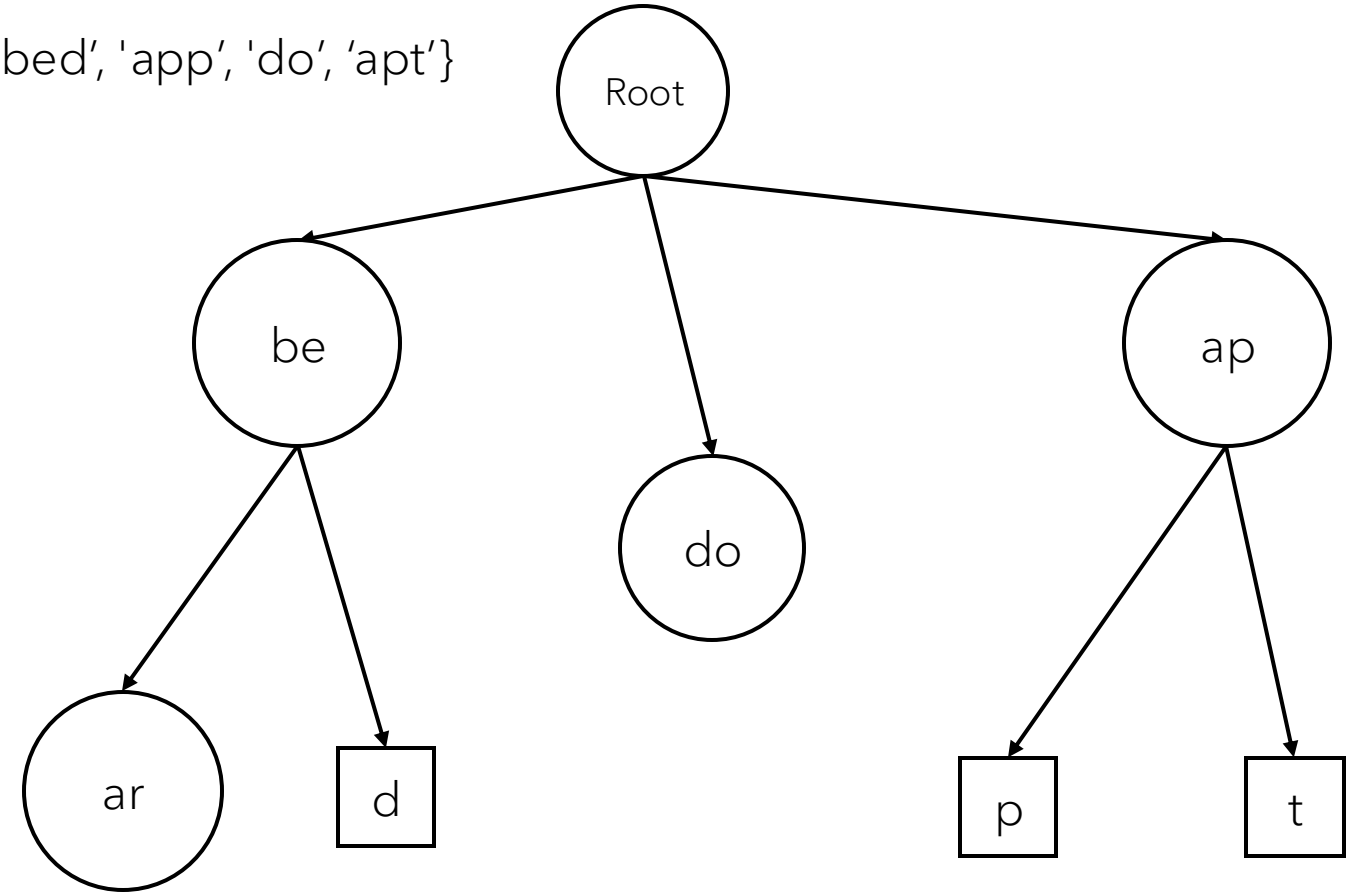


Compressed Tries:

- A compressed trie removes **chains of single-child nodes**.
- A compressed trie is a trie where **chains of single-child nodes are merged**
- Keep merging until a node has **multiple children** or is a **leaf**.
- Edges store **substrings**, not single characters.

Example:

List of words = {‘bear’, ‘bed’, 'app’, 'do’, ‘apt’}



Ukkonen's Algorithm

We want to build a suffix tree for the string-matching problem, in O(m) time.

For this, we utilize Ukkonen's Algorithm, presented by Esko Ukkonen in 1995.

The presentation of Ukkonen's Algorithm will be done iteratively, building as we receive the string. This makes it easily extensible (online).

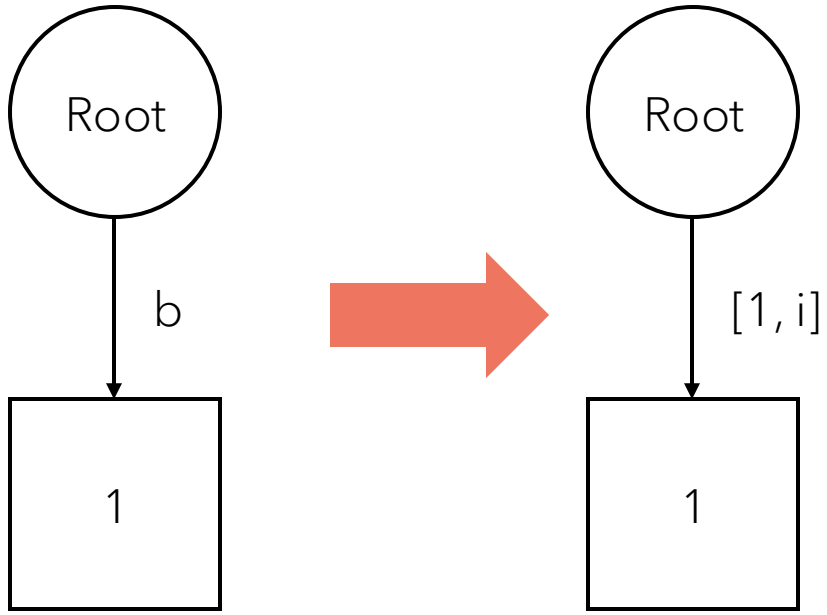
Step 1: Add first character to the tree

Insert the first character of the string into the tree. Treat it like it comprises the entire string for now, so sketch it as a root node.

Like in the case of other compressed tries, we are going use pointers / indices. Here, we label using start and stop indexes.

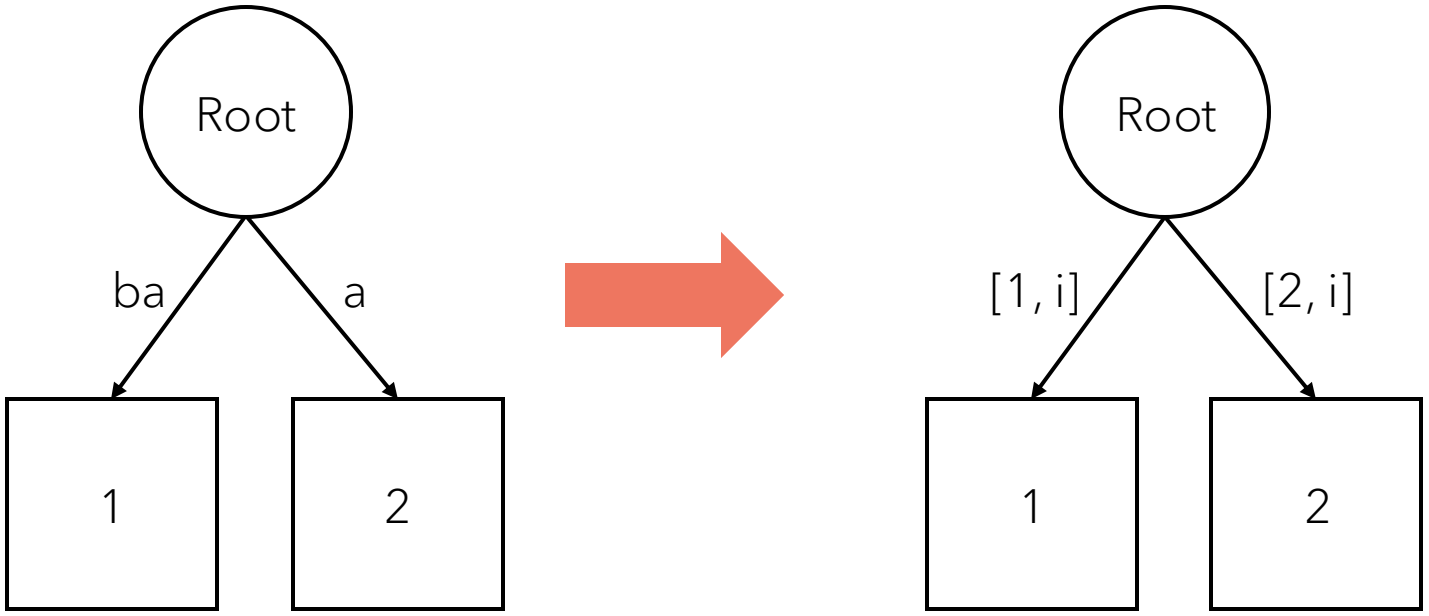
Start index here will be 1. Stop index is a variable at this stage in the algorithm (set to 1) that represents the current iteration that we're on.

(running example) : bananasna$

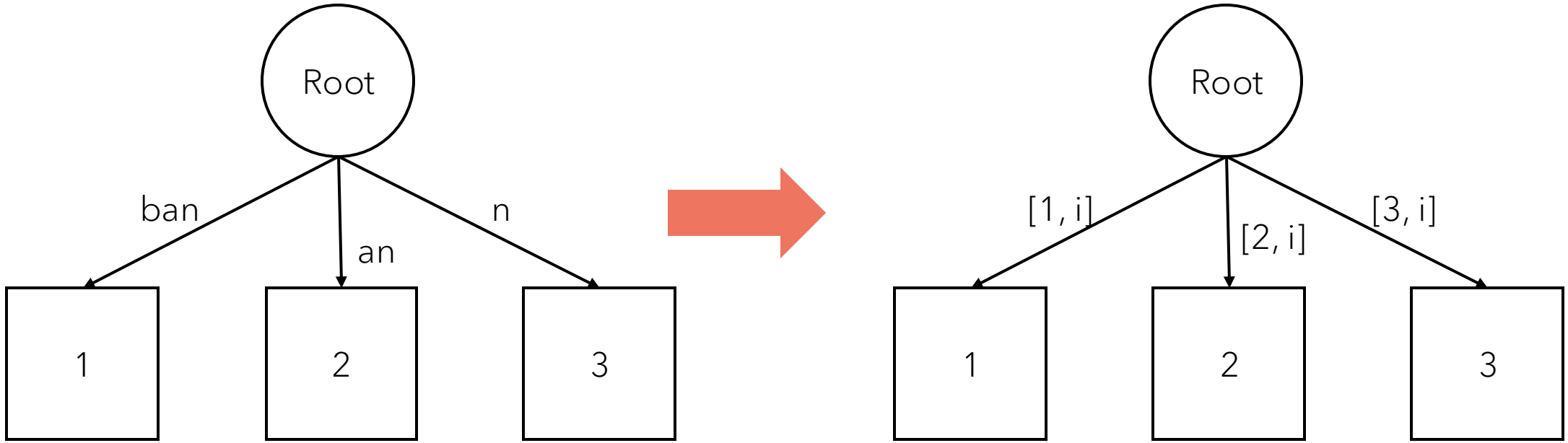


Step 2: Iterating over the rest of the string:

The first step in the iteration is to append the next character to each existing edge. Now we can see the real use of holding a variable ‘i’. By incrementing ‘i’, the existing edges automatically update themselves! We can actually repeat this as long as we don’t encounter the cases we discuss next.



and Step 3:



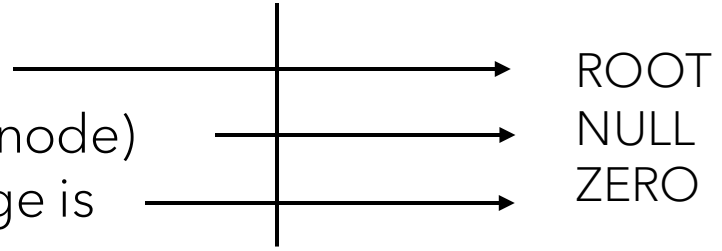
Step 3.5: The First ‘Case’ – Matching characters

If we see that the new character we receive is already there, we must now do something about it. Here, we introduce the variables of ‘remainder’ and ‘active_point’.

- Remainder: Stores how many suffixes we are waiting to implement. Increases by 1 at each step. At the end of each successful suffix insertion, it is decremented.

- Active Point: Useful to know where to insert a suffix. It’s a struct that stores:

  - active_node : current node of interest
  - active_edge : edge of interest (leaves the active_node)
  - active_length : how far inside the string active_edge is



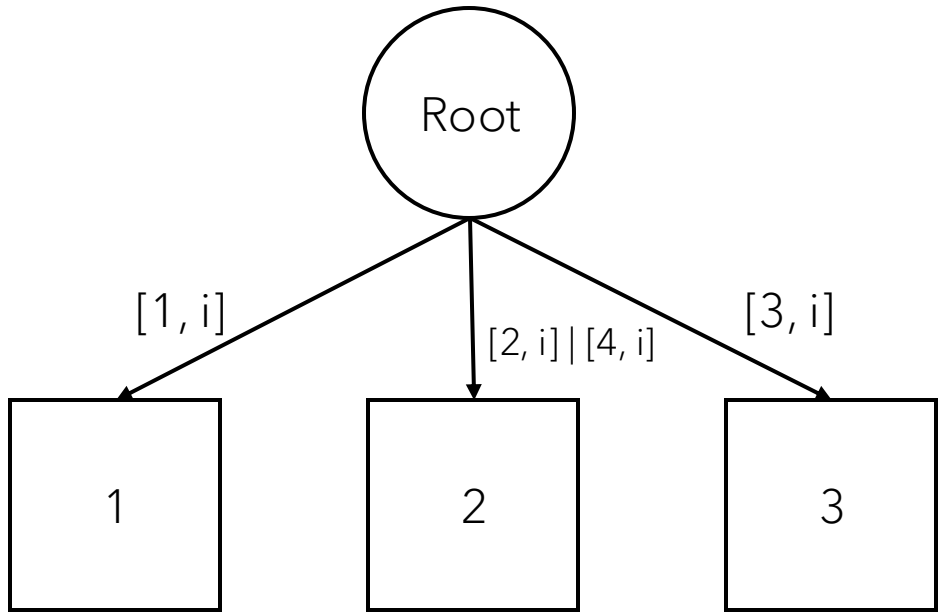
Step 4: The First ‘Case’ – Matching characters

Increment happens. So i = 4 at this stage.

From the active points, we go on searching for the current character in the suffix from the active node. To do this in O(1) time, which is critical to reducing the time complexity of construction to O(m), we use a map.

If we find that the new letter is already there, we do not insert a new edge. The colliding edge is the active_edge. There is no decrement to remainder either. active_length is incremented to 1.

The active_node becomes (root, ‘a’, 1).



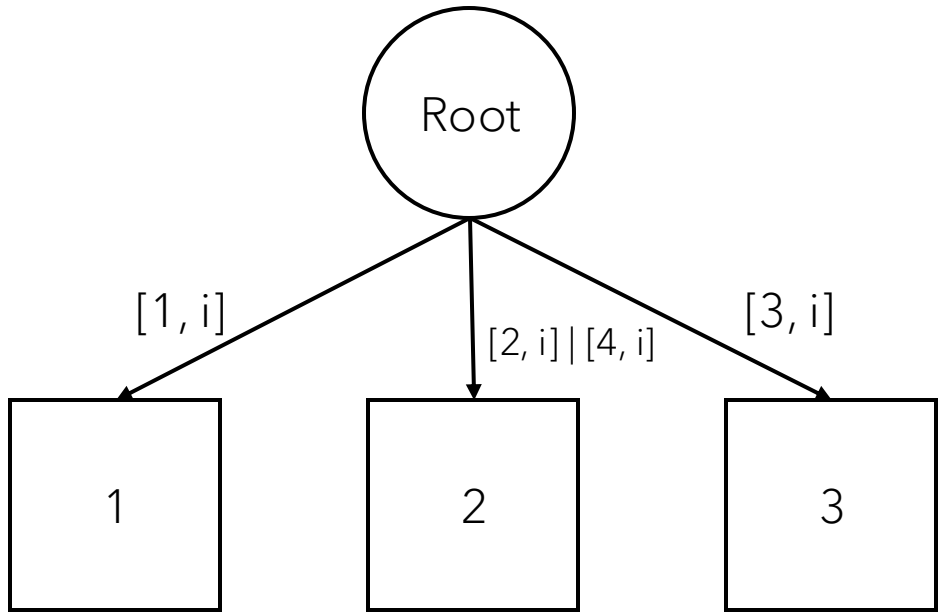
Step 5: The First ‘Case’ – Matching characters

Increment happens. So i = 5 at this stage. Remainder increases to 2 at the beginning of this step.

We want to insert both ‘n’ (to the remaining suffixes :– note this happens automatically when we increase i) and ‘an’ since we didn’t insert ‘a’ from the previous step.

We repeat the same steps as last time (keep in mind that active point is (root, ‘a’, 1) at the moment. The letter ‘n’ is again a colliding edge (determined using the map) at the active_node.

Do not insert again. active_length becomes 2. active_point is now (root, ‘a’, 2).



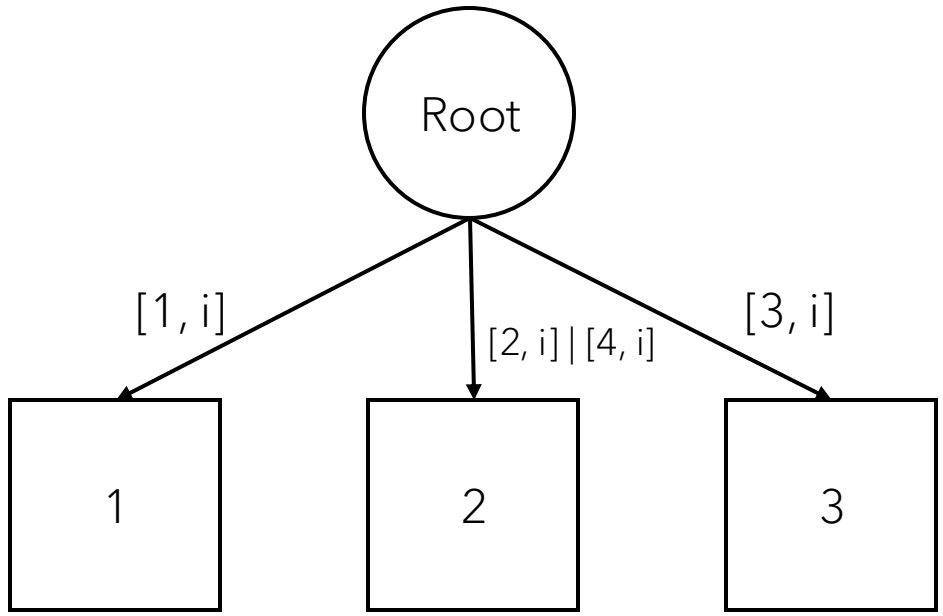
Step 6: The First ‘Case’ – Matching characters

Increment happens. So i = 6 at this stage. Remainder increases to 3 at the beginning of this step.

We want to insert ‘a’ (to the remaining suffixes :– note this happens automatically when we increase i) and ‘ana’, ‘na’ since we didn’t insert ‘n’ from the previous step.

We repeat the same steps as last time (keep in mind that active point is (root, ‘a’, 2) at the moment. The letter ‘a’ is again a colliding edge (determined using the map) at the active_node.

Do not insert again. active_length becomes 3. active_node is now (root, ‘a’, 3).



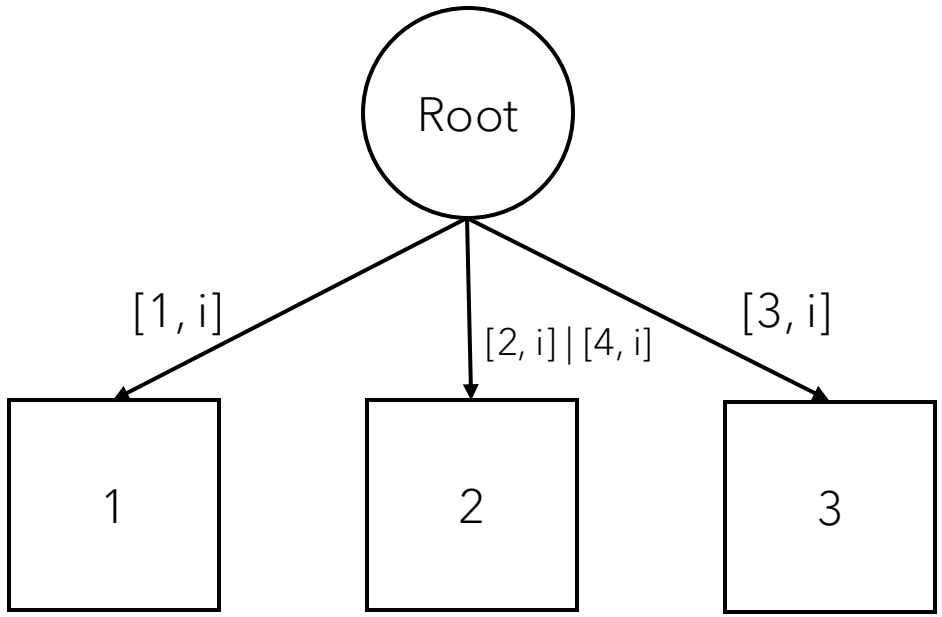
Step 7: The First ‘Case’ – Matching characters

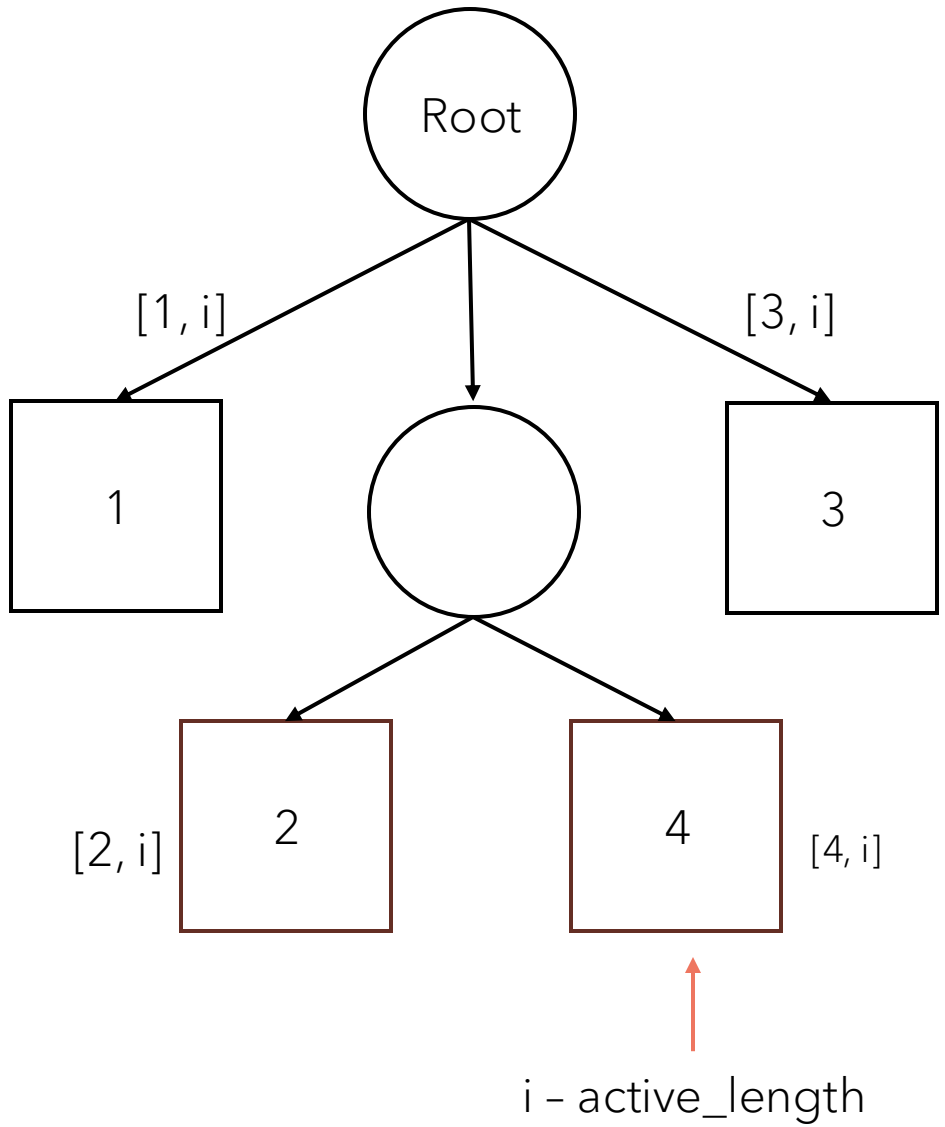
Increment happens. So i = 7 at this stage. Remainder increases to 4 at the beginning of this step.

We want to insert ‘s’ (to the remaining suffixes :– note this happens automatically when we increase i) and ‘anas’, ‘nas’, ‘as’ since we didn’t insert ‘a’ from the previous step.

We repeat the same steps as last time (keep in mind that active point is (root, ‘a’, 3) at the moment. Finally! No collision. We can insert some of the remaining nodes.

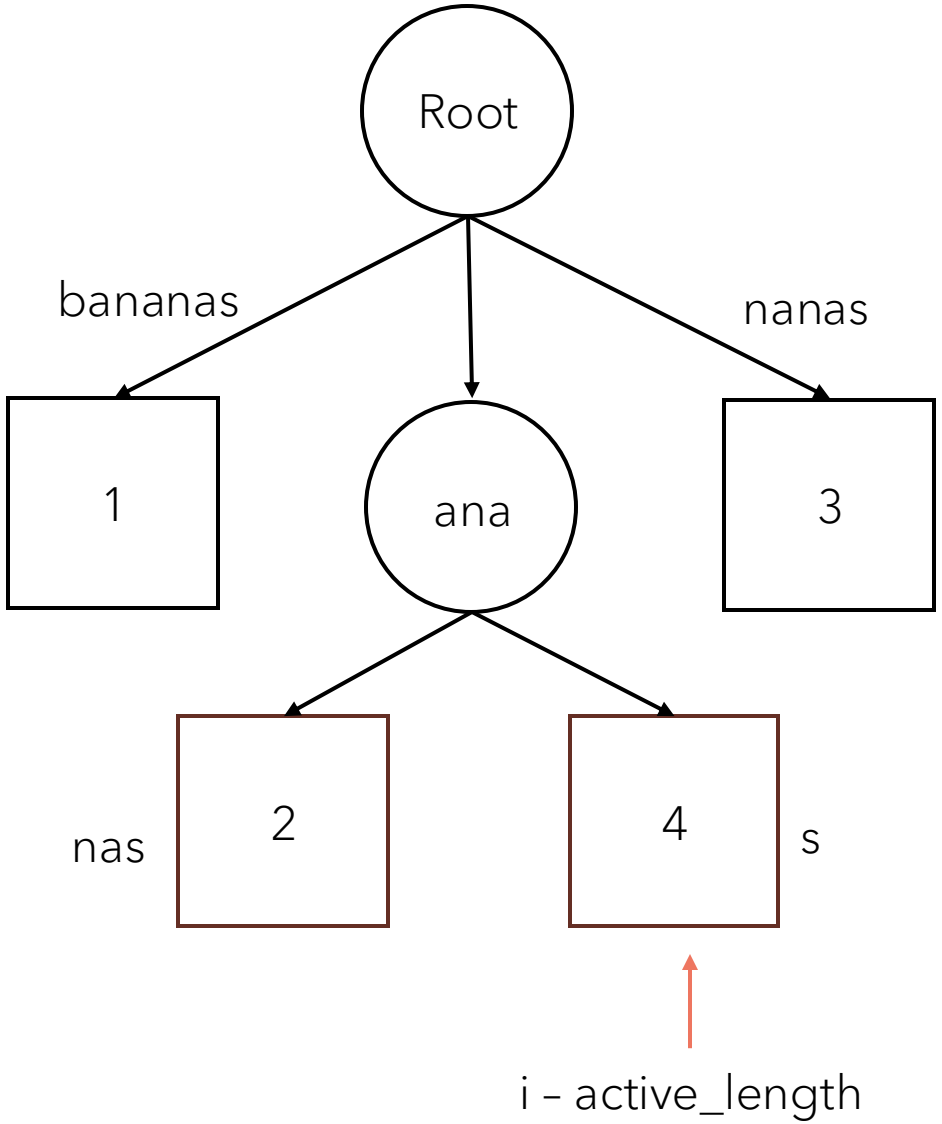
We have to split edges (we’ll see how) whenever we have a remainder > 0 and active_length > 0.





Create an internal node that splits the active point (which is (root, 'a', 3)).
It has two outgoing edges: the remaining characters after active point and the inserted / current character.
Successful insert! Decrement remainder to 3.

A Quick Check for the Running Algorithm:

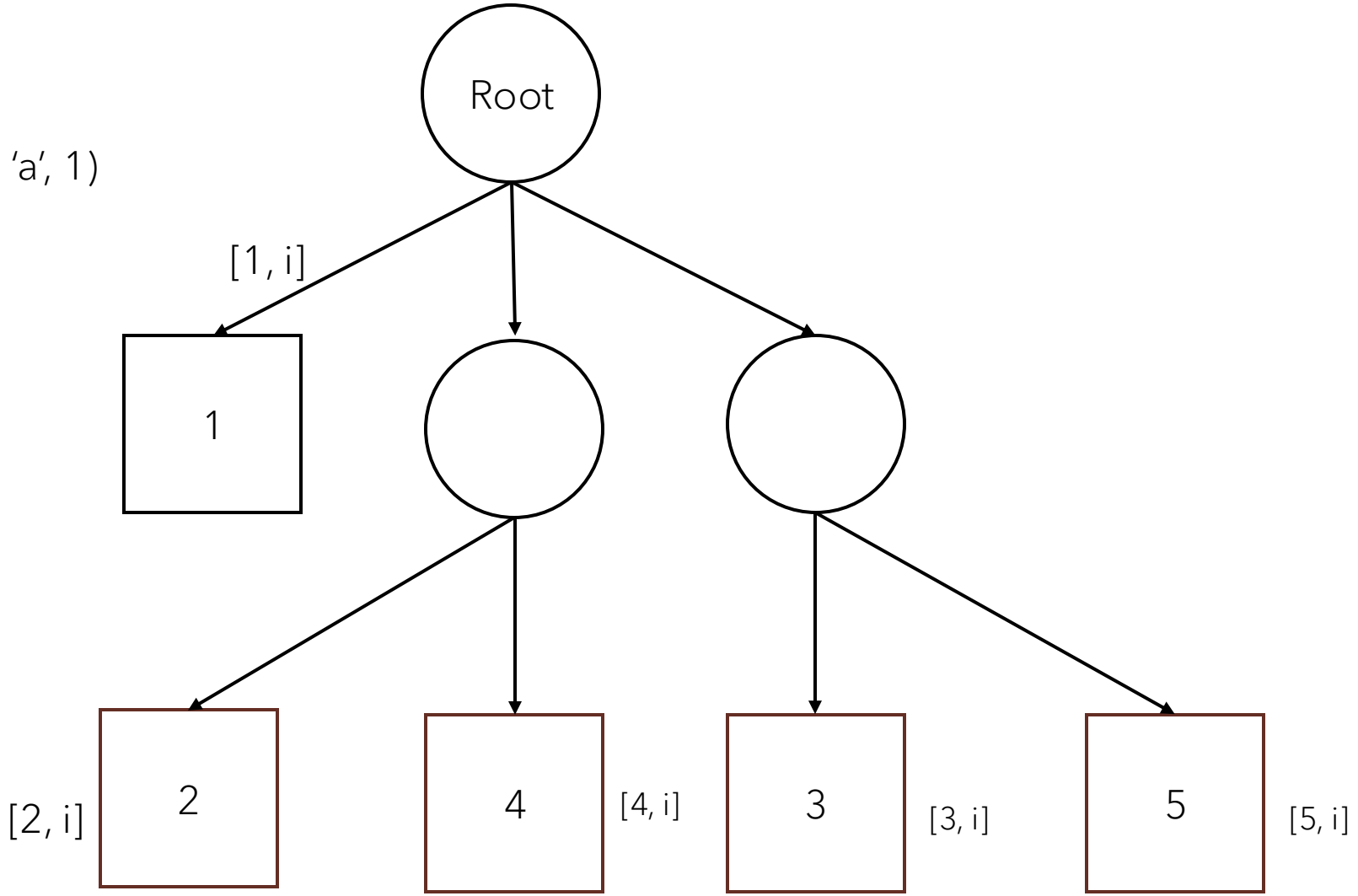


We update the active node according to this insertion. If we are at the root, the active node remains the same. The active_edge becomes the first character of the suffixes yet to be successfully inserted. Here, we are left with (in the order): ‘nas’, ‘as’, and ‘s’. So active_edge becomes ‘n’. Also, active_length is decremented.

This becomes the problem at i = 7, trying to insert ‘s’ as the new letter and the suffixes ‘nas’, ‘as’, and ‘s’ are to be inserted, with the active_node as (root, ‘n’, 2). Great! We can try to wrap this up using the same steps as we just performed on this smaller problem.

After the next step, we get an active_point as (root, ‘a’, 1). We’ll see that there’s a new step we have to take to ensure correctness.

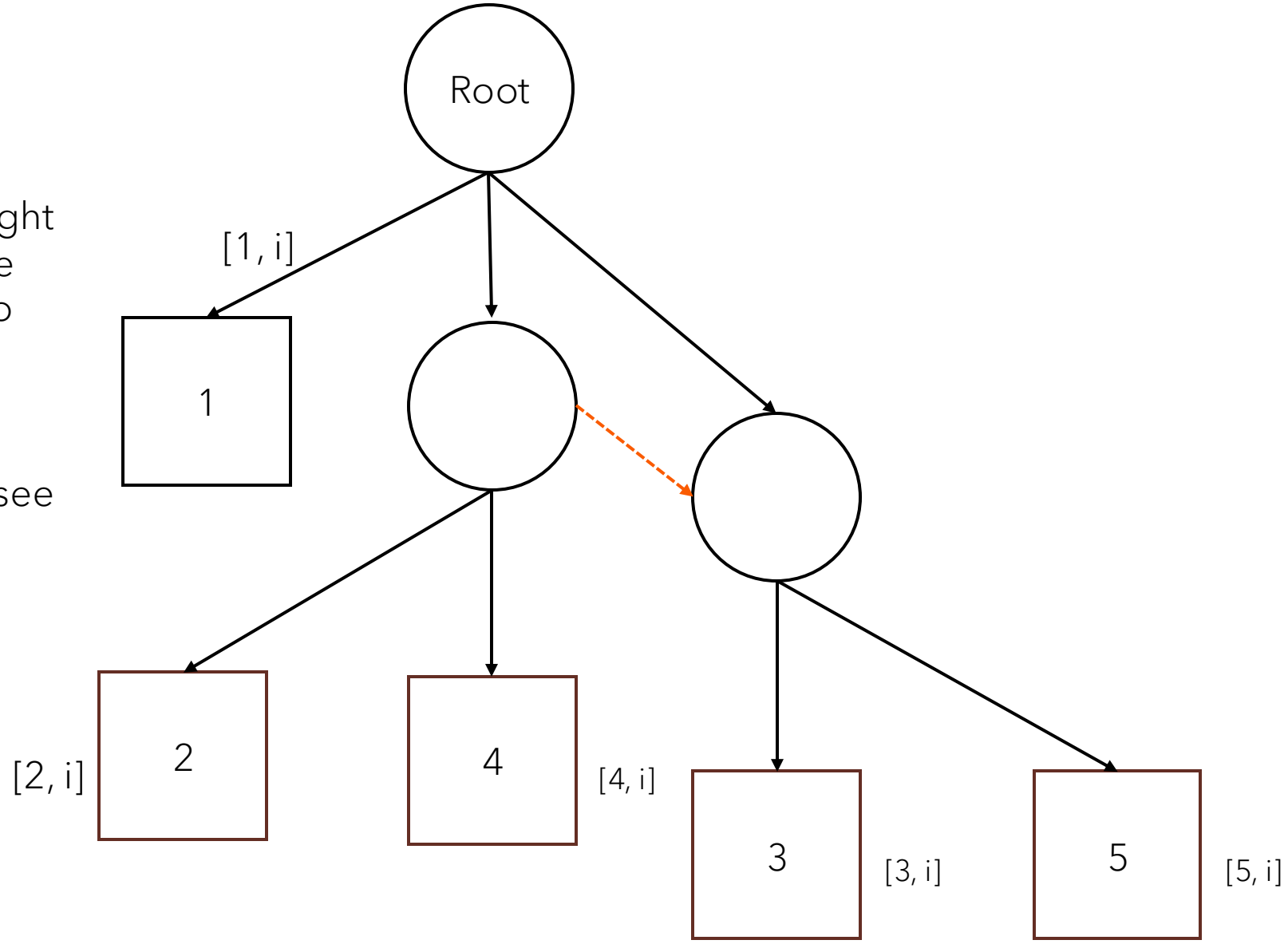
active_node = (root, ‘a’, 1)
Remainder = 2

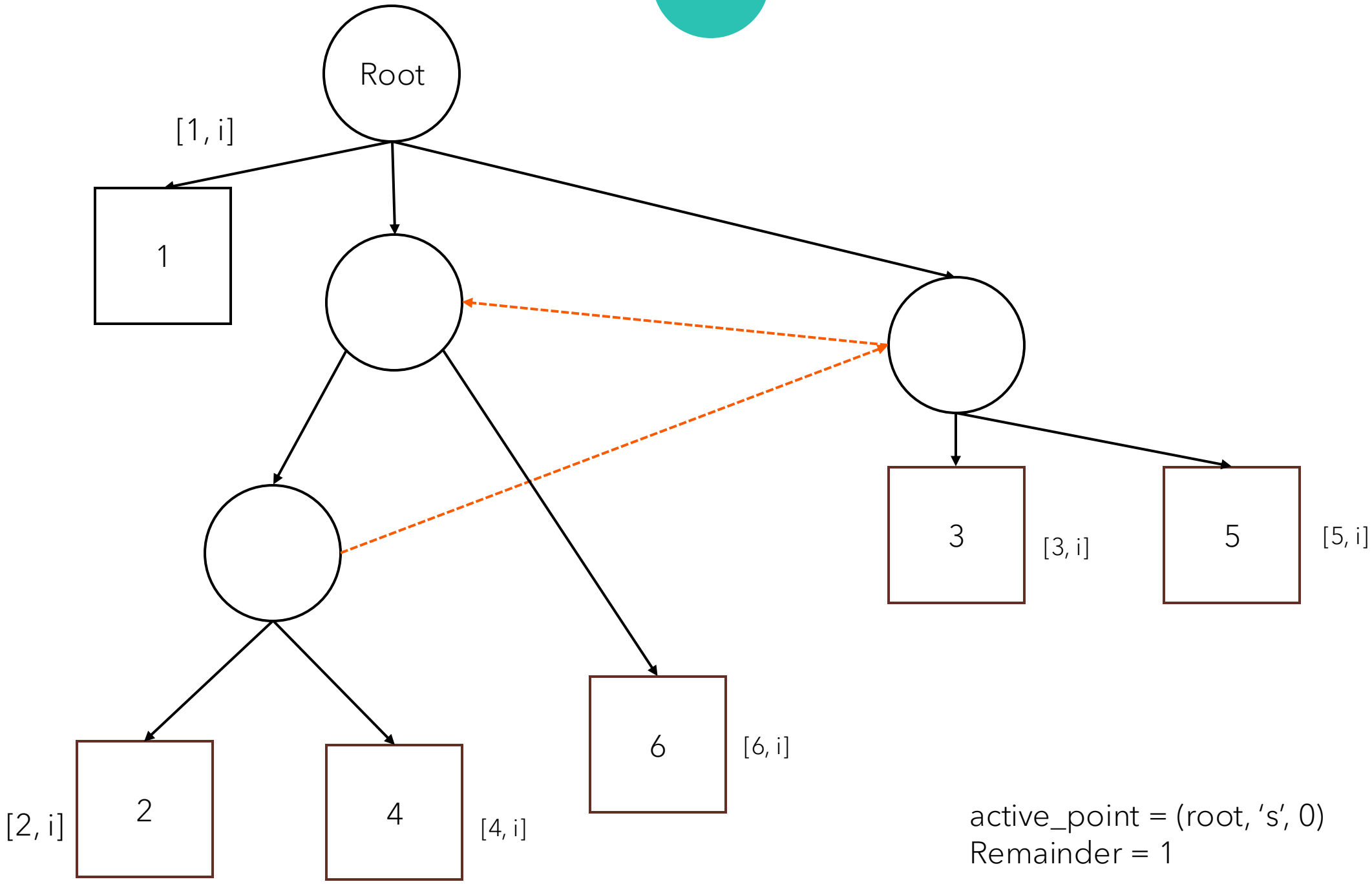


The additional step: Suffix Links!

When we insert a new internal node right after inserting an internal node (like we did here) on the same step, we have to add a link between the two internal nodes.

This link is called a Suffix Link. We will see why this is important in just a bit.

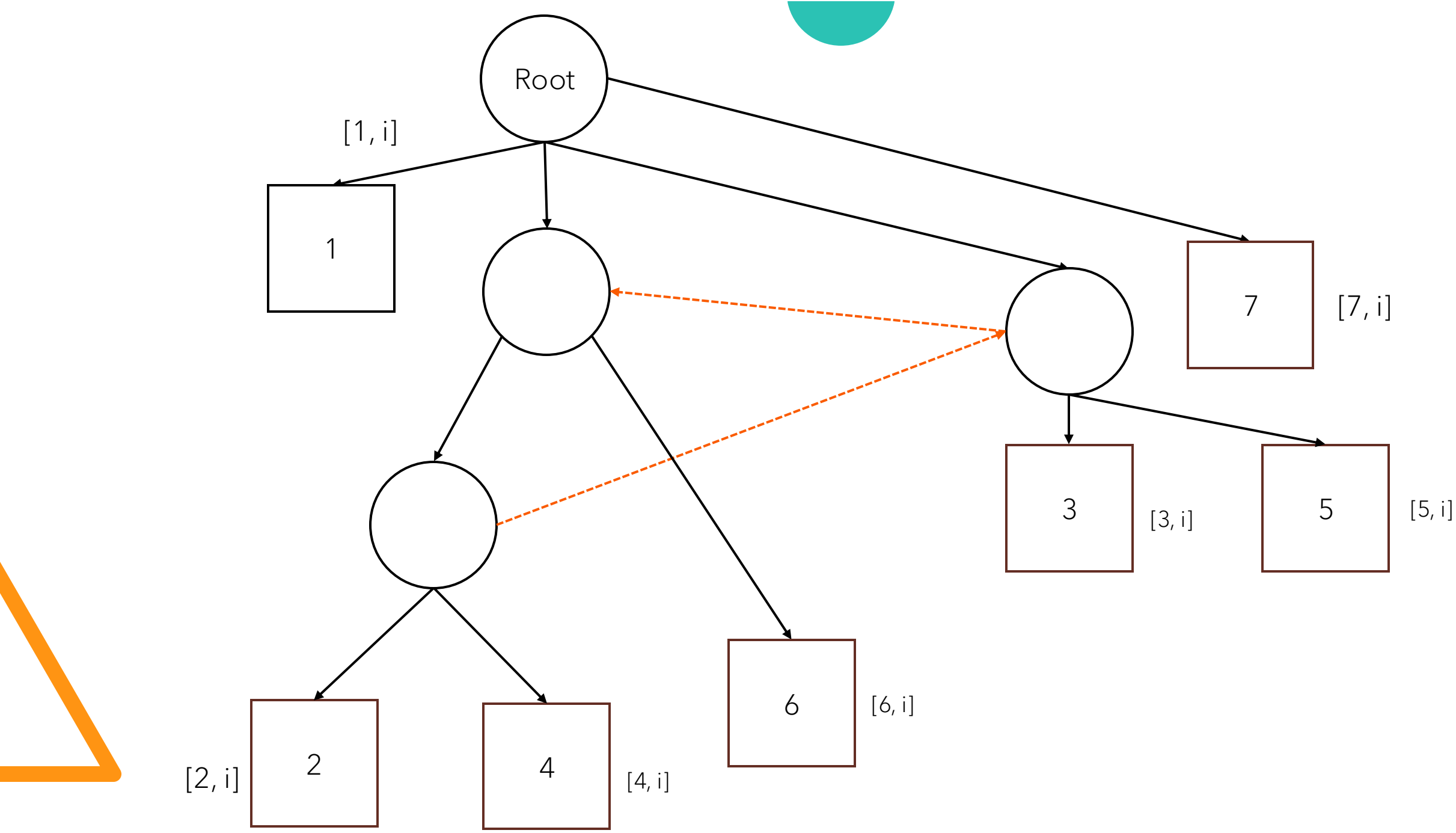




active_point = (root, ‘s’, 0)
Remainder = 1

active_length = 0. We are back to the basic kind of insertion!
We are searching for the edge ‘s’ at the root. Of course, there is no ‘s’ yet, so we can just create a leaf node that contains just the substring ‘s’.

When we reach remainder = 0, we reset the active_node to the aforementioned default values. Now, we can move on to step 8.


Root
[1, i]
1
7
[7, i]
3
[3, i]
5
[5, i]
6
[6, i]
[2, i]
2
4
[4, i]

Step 8:

Since ‘n’ is already an edge (determined by the map) at the active_node, we can do what we did before, incrementing remainder to 1, and change the active_point to (root, ‘a’, 1).

Step 9:

Since ‘a’ is already an edge (determined by the map) at the active_node, we can do what we did before, incrementing remainder to 2, and change the active_point to (root, ‘n’, 2). However, unlike before, we won’t be able to access any ‘next’ characters (since the sentinel is next). The active_point is moved to the node the edge, with active_edge to NULL and active_length to 0. active_point becomes (X, NULL, 0).

Step 10:

We encounter the sentinel. Remainder = 3, and we are yet to insert ‘na$’, ‘a$’, and ‘$’. The active_point is (X, NULL, 0).

Insertion of ‘$’ at the node corresponding to ‘na’ happens. Since $ cannot conflict, it is directly added. Now, we decrement remainder = 2.

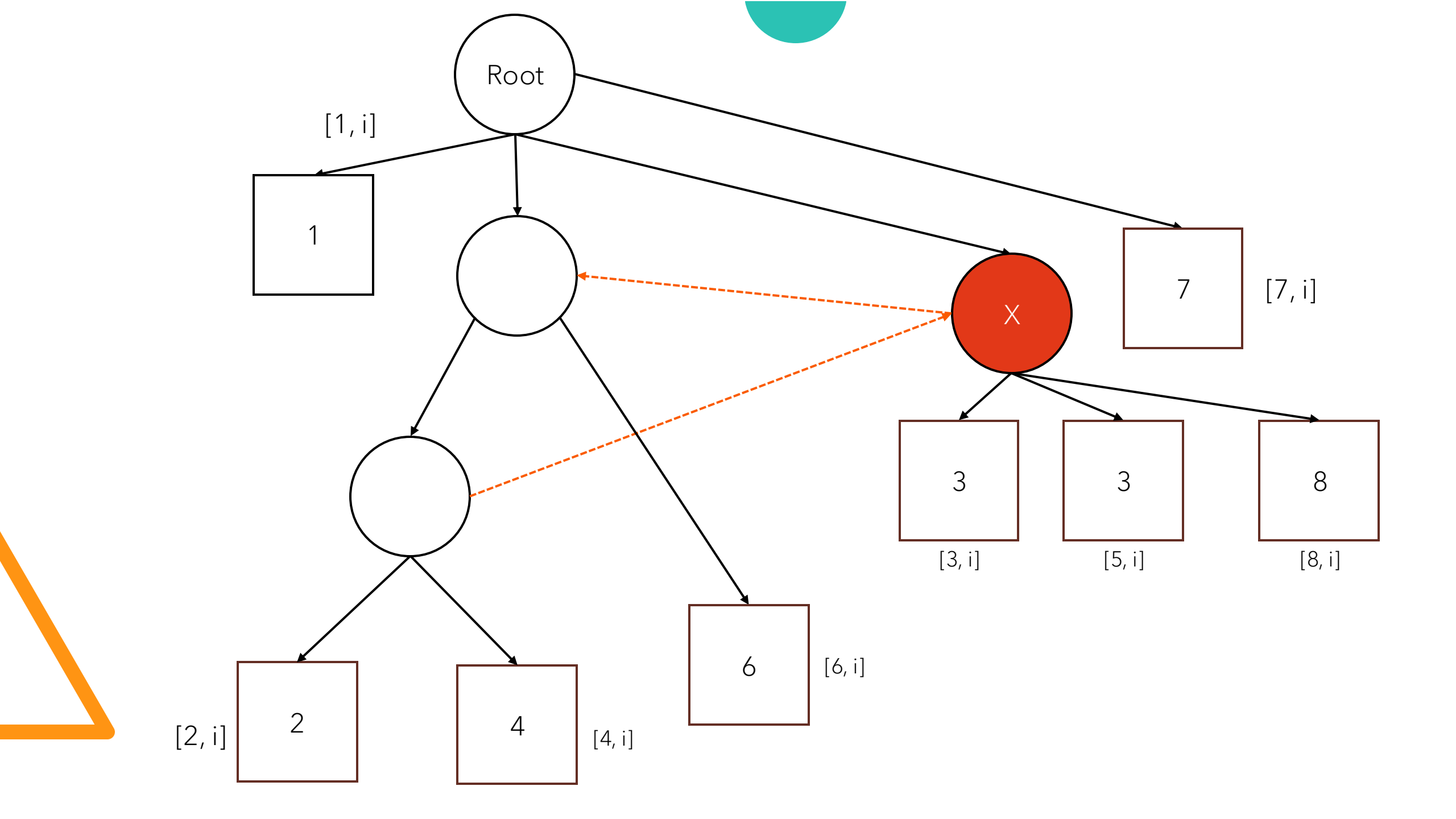
Here, the suffix link becomes importance.

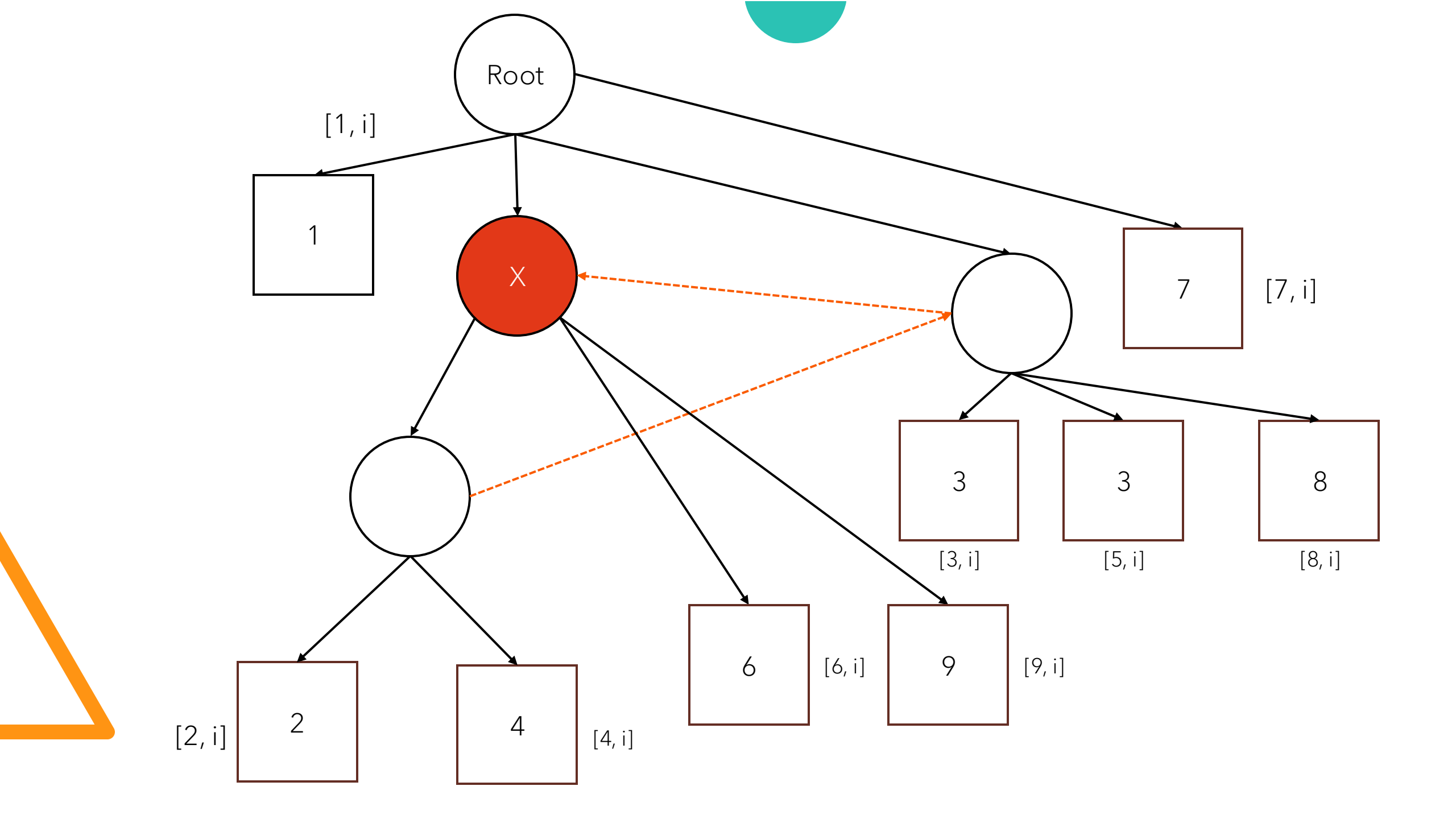
> When we split an edge / add an edge to a node that is not the root (like we just did), we follow the suffix link to make the encountered the new active_node. No changes are made to active_edge or active_length. Here, they become (Y, NULL, 0)

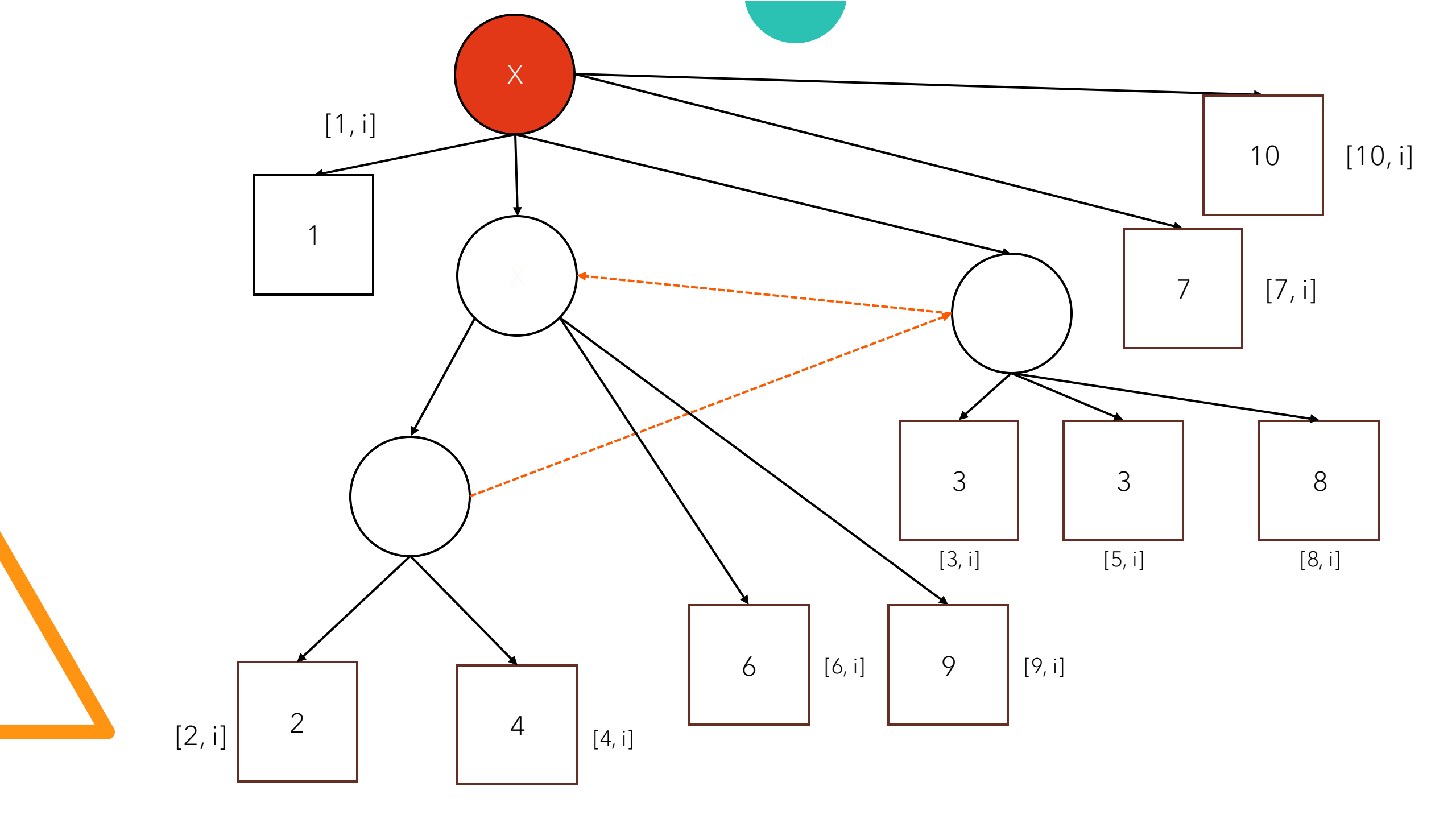
This lets us make the next insertion in O(1) time.

Insertion of ‘$’ at the node corresponding to ‘a’ happens. Since $ cannot conflict, it is directly added. Nowe we decrement remainder = 1. Here, when no suffix link found, we set active_node to ROOT.

Insertion of ‘$’ happens at the root without issue.


Root
[1, i]
1
X
7
[7, i]
3
3
8
[3, i]
[5, i]
[8, i]
6
[6, i]
[2, i]
2
4
[4, i]


Root
[1, i]
1
X
7
[7, i]
3
3
8
[3, i]
[5, i]
[8, i]
6
[6, i]
9
[9, i]
[2, i]
2
4
[4, i]


X
[1, i]
1
10
[10, i]
7
[7, i]
3
3
8
[3, i]
[5, i]
[8, i]
6
[6, i]
9
[9, i]
[2, i]
2
4
[4, i]

// Ukkonen's Algorithm

```
i = 0;
remainder = 0;
(active_node = root, active_edge = null, active_length = 0) // active point

create root node;
while S not empty {
  i++;
  remainder++;

  // S[i] is next character after active point
  if S[i] == next char after active point {
    if active_length == 0 {
      active_edge = edge beginning with S[i];
    }
    active_length++;

    if active point is at the end of an edge {
      active_node = node edge is pointing to;
      active_edge = null;
      active_length = 0;
    }
  }
  // S[i] is not next character after active point
  else {
    while remainder > 0 {

      if active_length == 0 {
        create edge for S[i];
        create leaf node with integer value of i;
        point new edge to new leaf node;
      }

      else {
        // split edge
        create internal node and point active_edge to it;
        active_edge stops at active point; // e.g., abc|def -> abc|
        create new edge with remains of active_edge; // e.g., def
        point new edge to node that active_edge previously pointed to;

        create new edge leaving new node with integer start and i for end;
        // e.g., if i = 4, [4, i]

        create new leaf node with integer value of i - active_length;
        point new edge to this leaf node;

        if not the first split edge {
          create suffix link from previous node to current;
        }
      }

      remainder--;

      if active_node == root {
        active_edge = first letter of next remaining suffix or null; // if remainder == 0
        active_length-- or 0;
      }
      else {
        if node has suffix link {
          active_node = node pointed to by suffix link;
        }
        else {
          active_node = root;
        }
      }
    }
  }
}
```

Searching for Patterns in A Suffix Tree

- Any single string P_i of length $|P_i|$ is found in the suffix tree (or) declared not to be there, in $O(|P_i|)$ time.

- As expected, this is accomplished by matching the string against a path in the tree starting from the root. If S is a substring of the text string, then the algorithm can find all occurrences of S as a substring. This takes $O(|P_i| + k)$ time, where k is the number of occurrences of the substring.

Searching for Patterns in A Suffix Tree

- As expected, this is achieved by traversing the subtree below the end of the matched path for P_i. Each leaf node rooted at the end of the matched path corresponds to one instance where the string P_i occurs as a substring (these are discovered by BFS/DFS)
- If the full string P_i cannot be matched against a path in *T*, then Pi is not in *T*.
- However, the matched path does specify the longest prefix of S that is contained as a substring in the database.

Results And Analysis

Complexity Analysis

- Nodes and edges are created in **constant time** → O(1)
- Each extension step creates one of the following in O(1) time:
  - a single edge + leaf node
  - a split edge + an internal node + a leaf node
  - or just a variable update
- Every leaf corresponds to one suffix → **n leaves** for a string of length *n*
- At most **m − 1 internal nodes**, so total nodes ≤ **2m**
- Total creation cost → **O(1) × 2m = O(m)**

Running The Code

Comparison with alternative Algorithm

Suffix Tree v/s Naïve Algo(m = 10000 and r = 10)

Test Cases	Suffix Tree Time (ms)	Naive Algo Time (ms)	Performance Gain
T1	9989	28918	2.89x Faster
T2	2360	14417	6.10x Faster
T3	1540	24618	15.98x Faster
T4	0*	16064	Instant
T5	6657	22181	3.33x Faster

Bibliography:

- 'Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology' by Dan Gusfield.
- https://en.wikipedia.org/wiki/Ukkonen%27s_algorithm
- https://www.geeksforgeeks.org/dsa/ukkonens-suffix-tree-construction-part-1/
- https://cp-algorithms.com/string/suffix-tree-ukkonen.html
- https://www.baeldung.com/cs/ukkonens-suffix-tree-algorithm
- https://programmerspatch.blogspot.com/2013/02/ukkonens-suffix-tree-algorithm.html
- https://brenden.github.io/ukkonen-animation/
- https://www.youtube.com/watch?v=ALEV0Hc5dDk

Thank you

Krish Agarwal – MA24BTECH11014

Sagar Kumar - MA24BTECH11020