STRING PROCESSING

ISIS 2801

String processing algorithms

- Longest Common Subsequence (LCS)
- **Suffix tries, trees & arrays, wavelet trees**
- Palindromes

Suffix strings

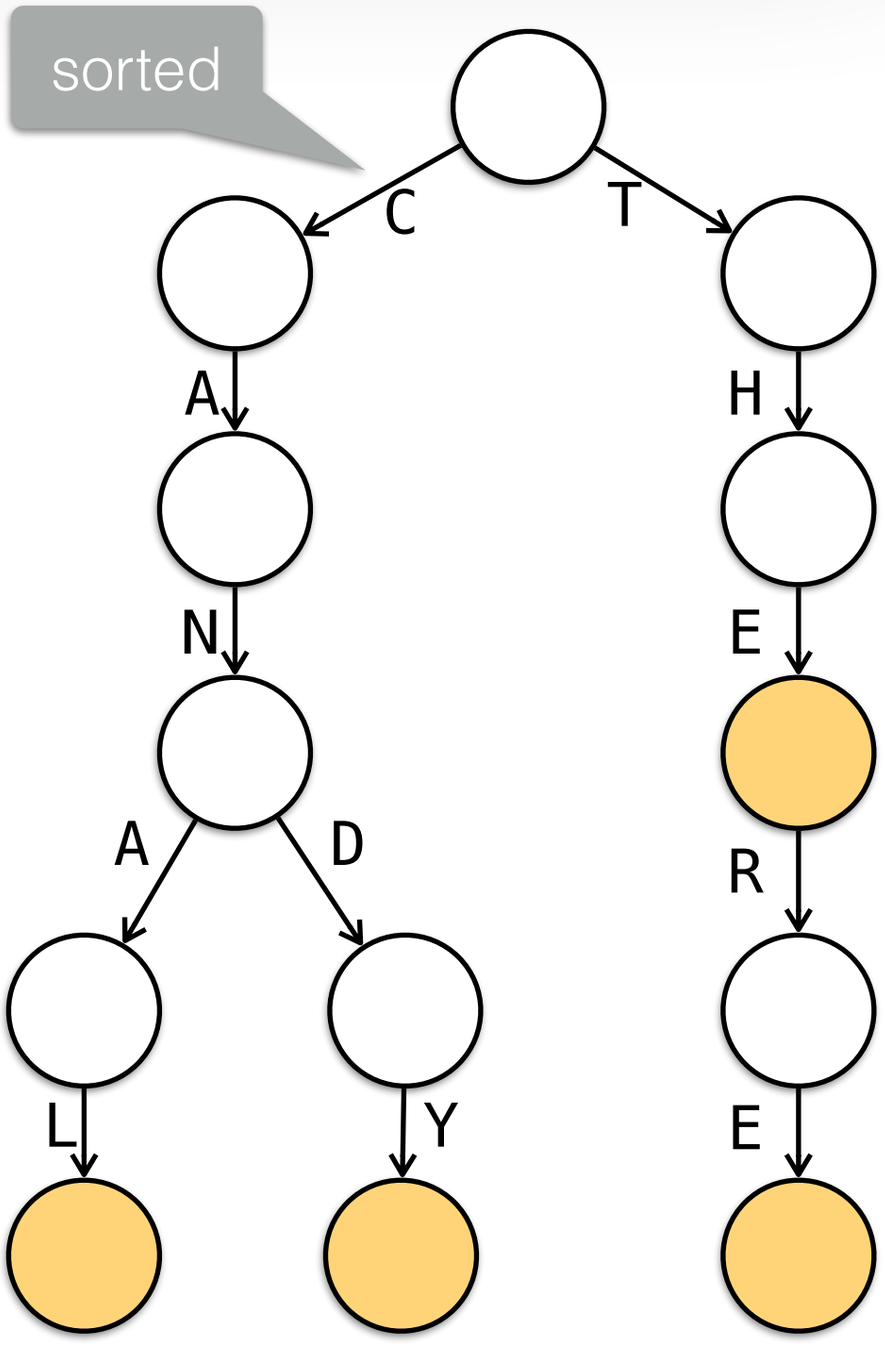
Definition A suffix is the last part of a string. Given a string we can get all its substrings based on their position

```
                0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
      s = h  e  l  l  o     s  t  r  i  n  g     s  u  f  f  i  x  e  s

                0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
suffix(s,6) =                    t  r  i  n  g     s  u  f  f  i  x  e  s

                0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
suffix(s,14) =                                       f  f  i  x  e  s
```

SUFFIX TRIES

Tries

Definition A suffix trie is used to keep track of all suffixes for the strings in a set S.



Edges have string characters

Vertices have suffixes (aka paths to the vertex)

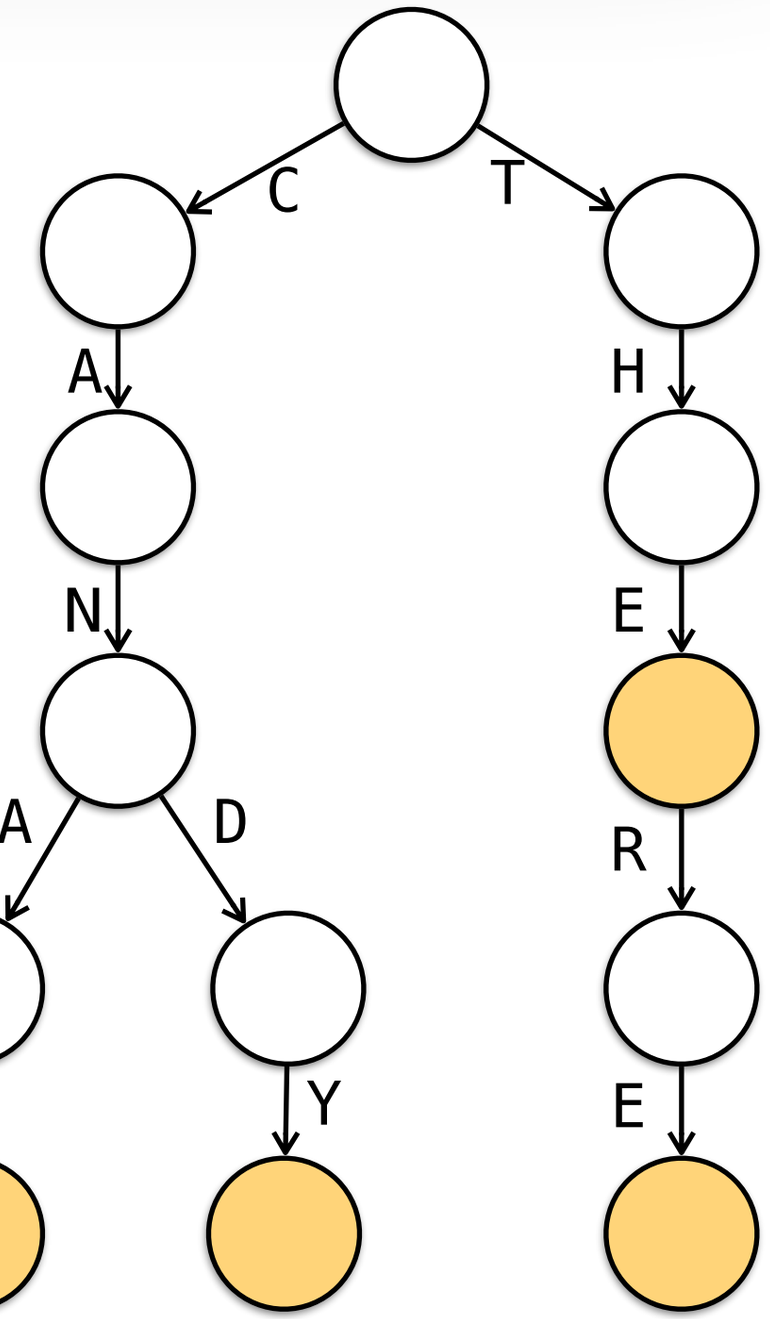
Vertices know whether there is a suffix terminating the vertex

S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

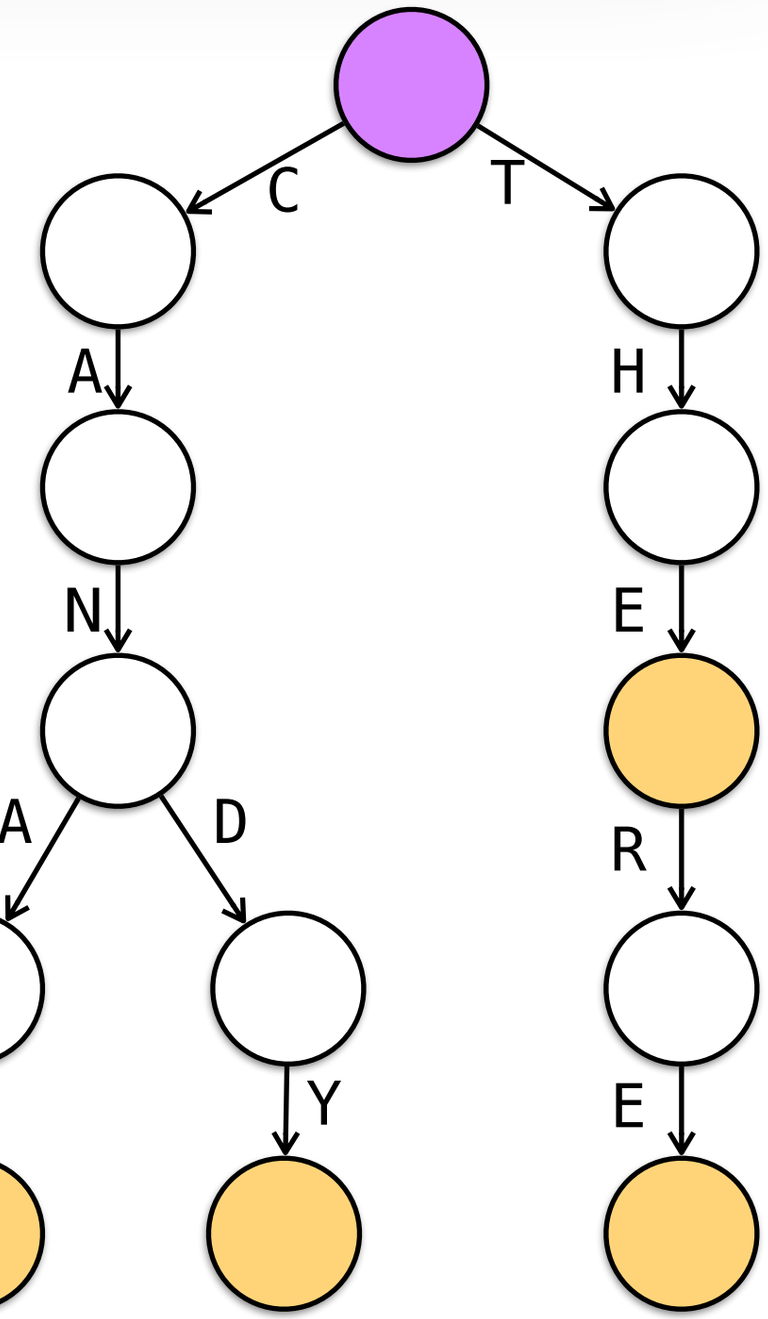


S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

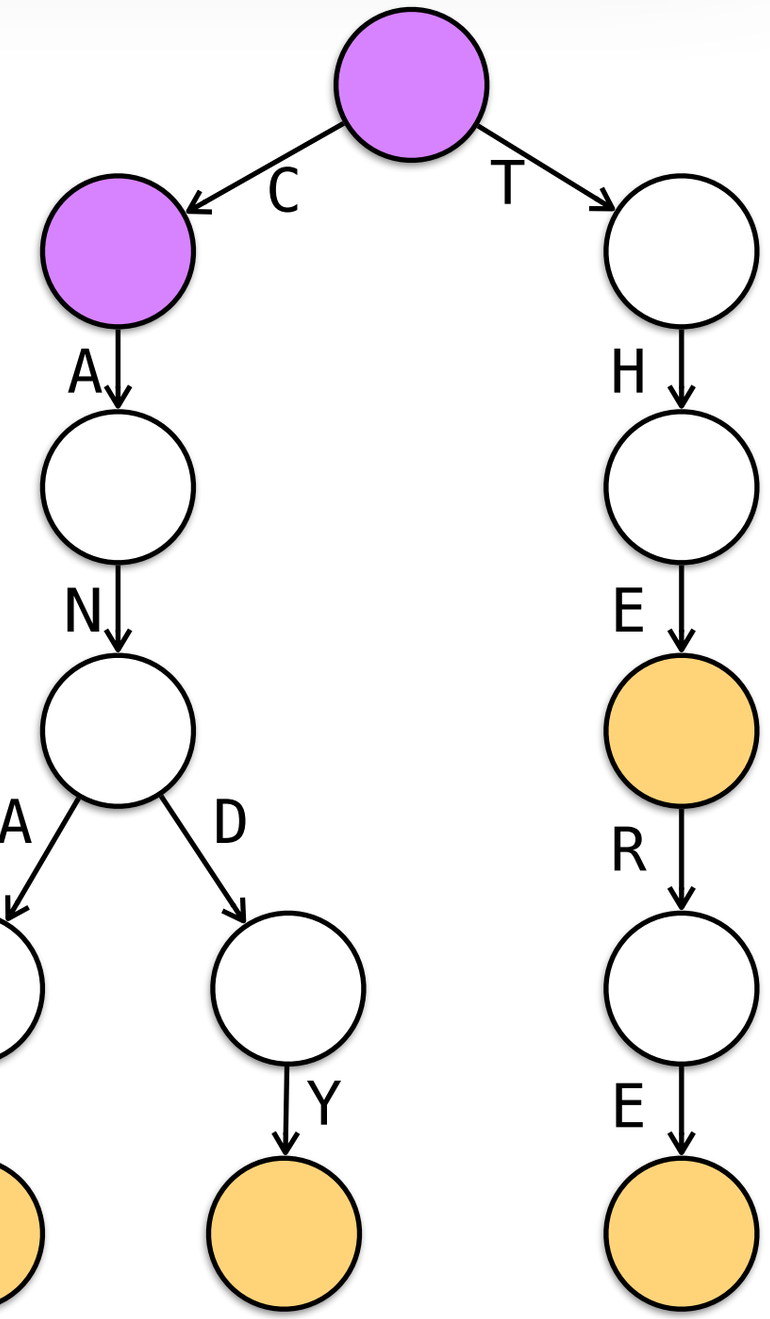


S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

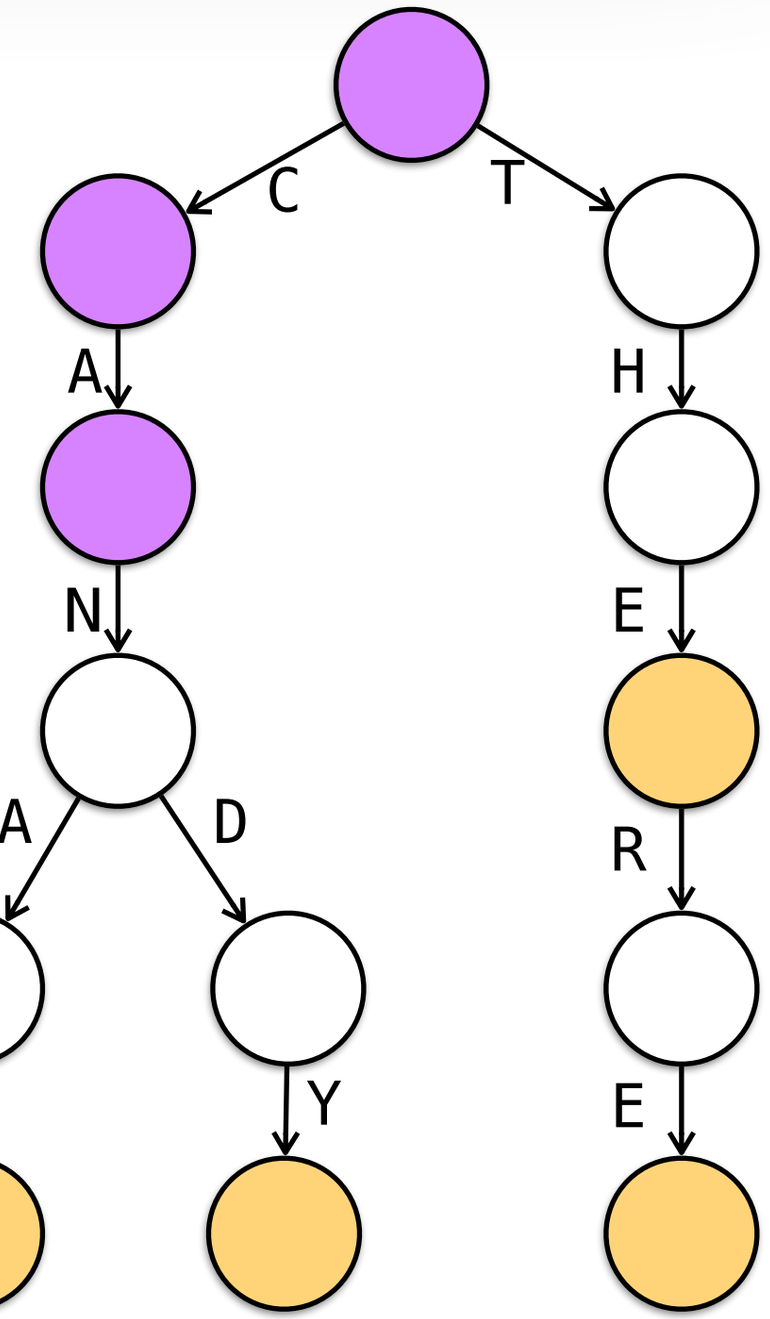


S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

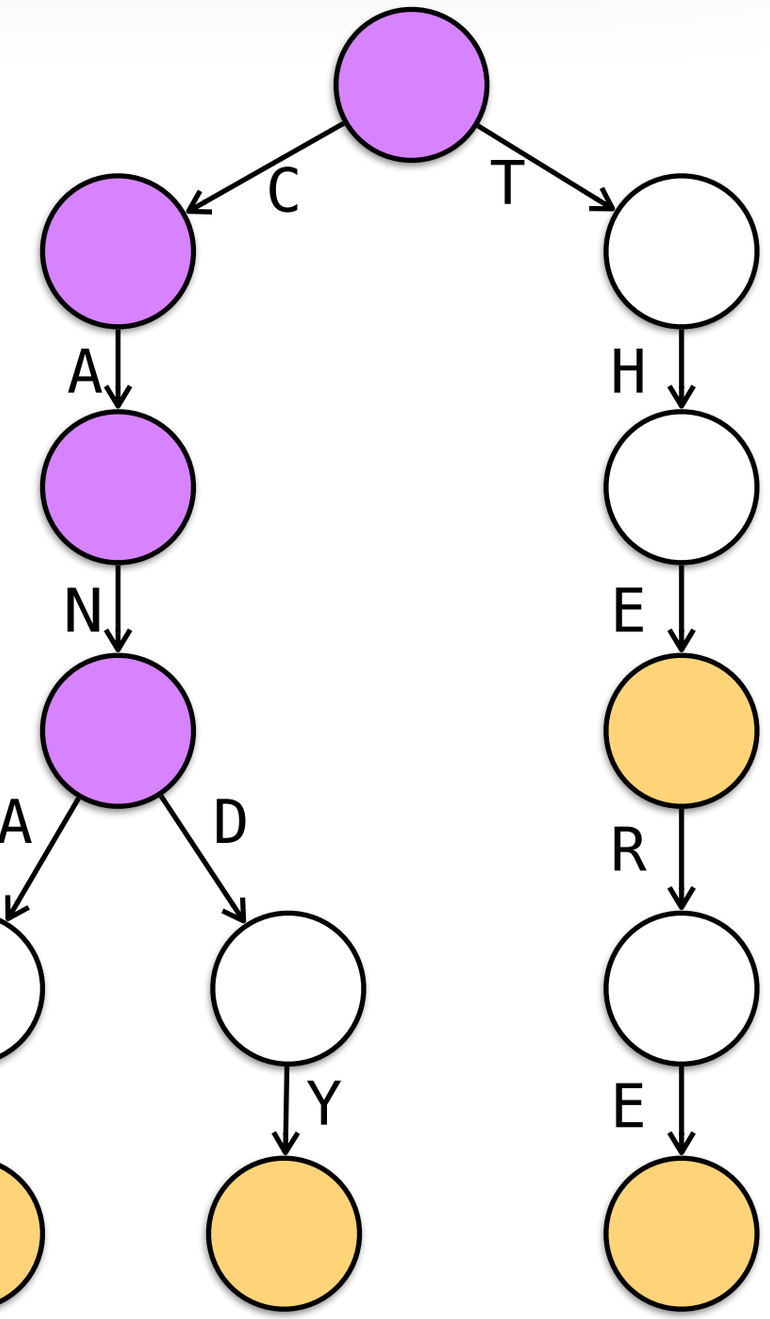


S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

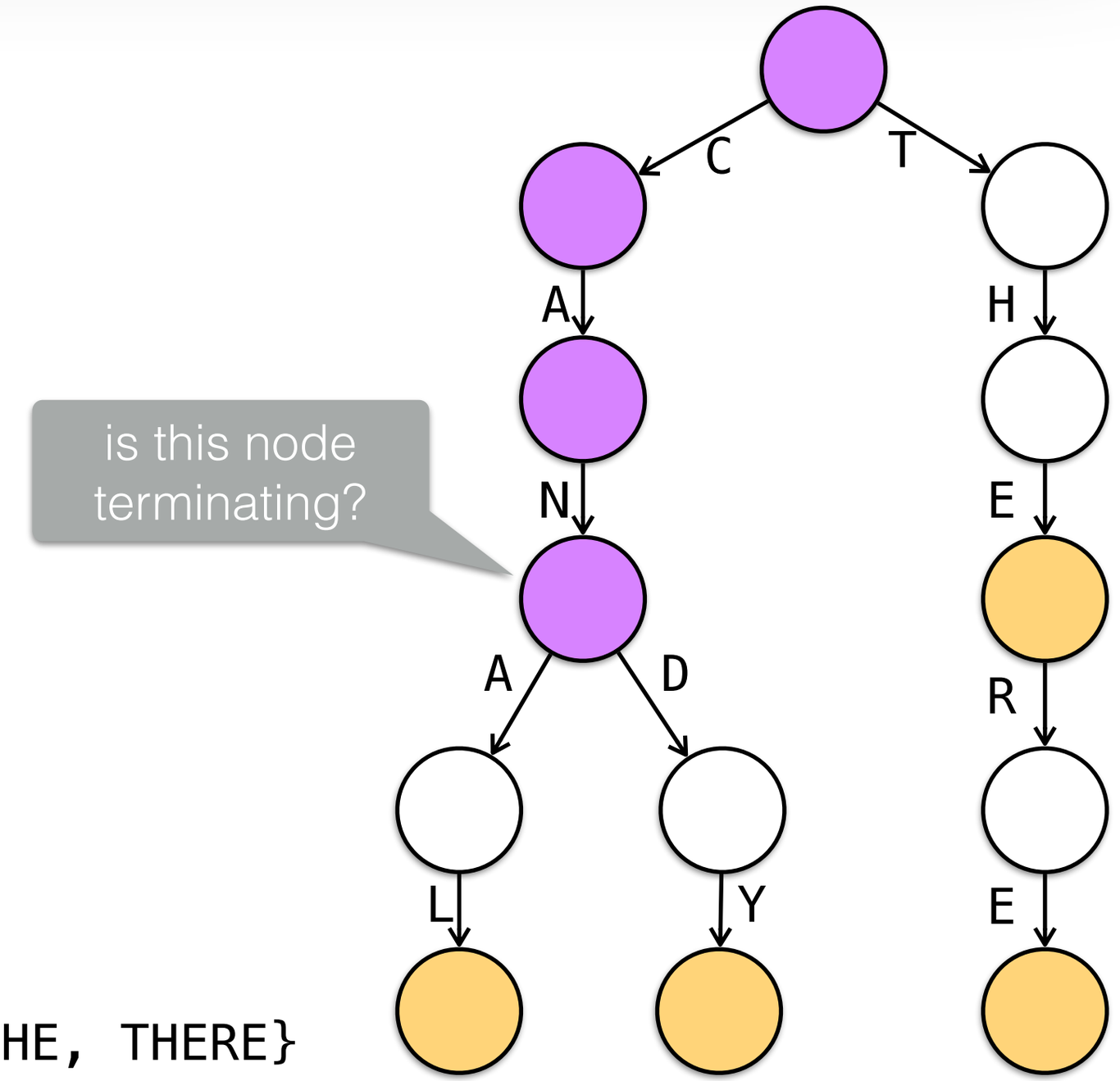


S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

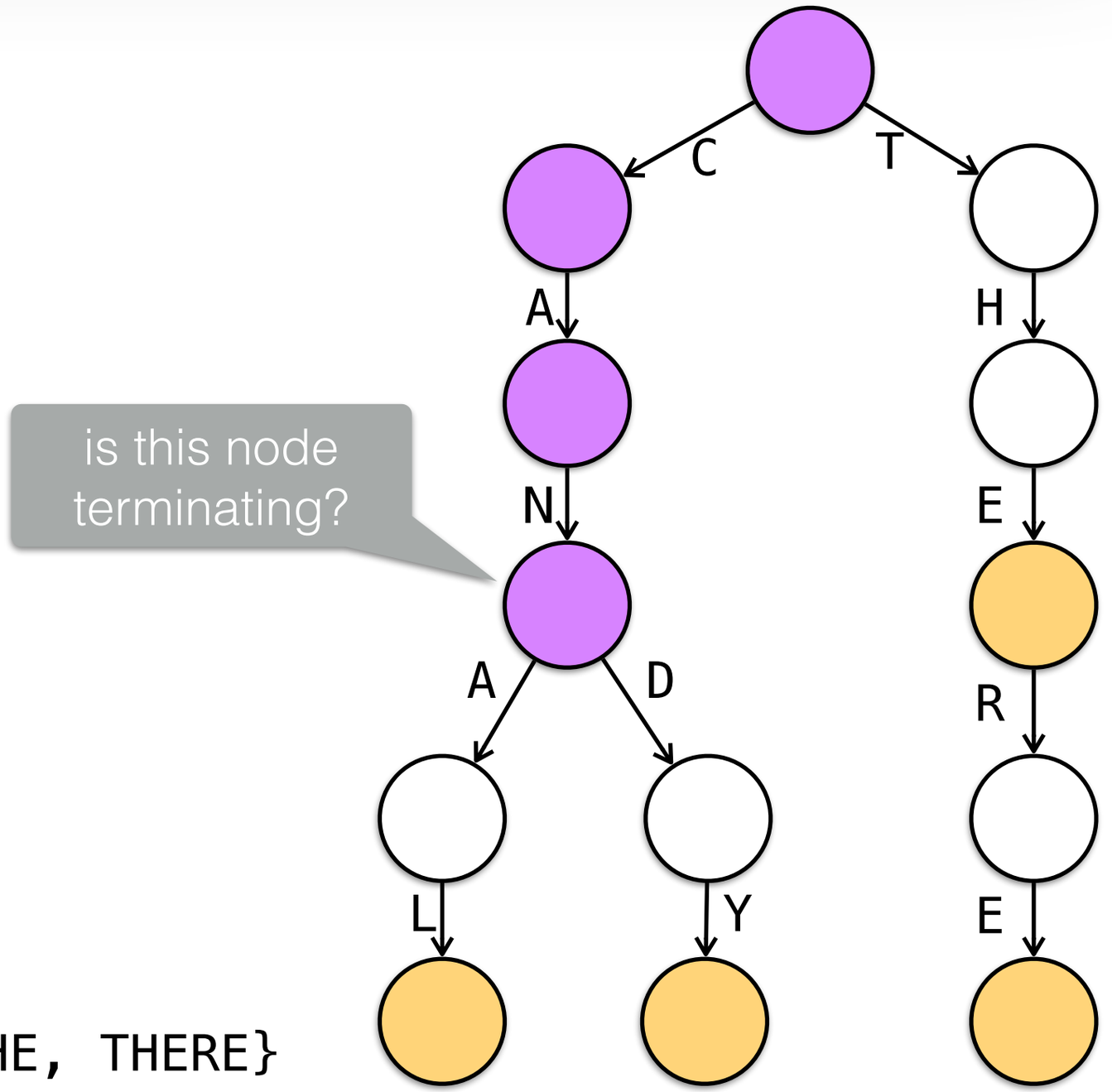


S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?



Can check **all** words in the dictionary, or **generating** words

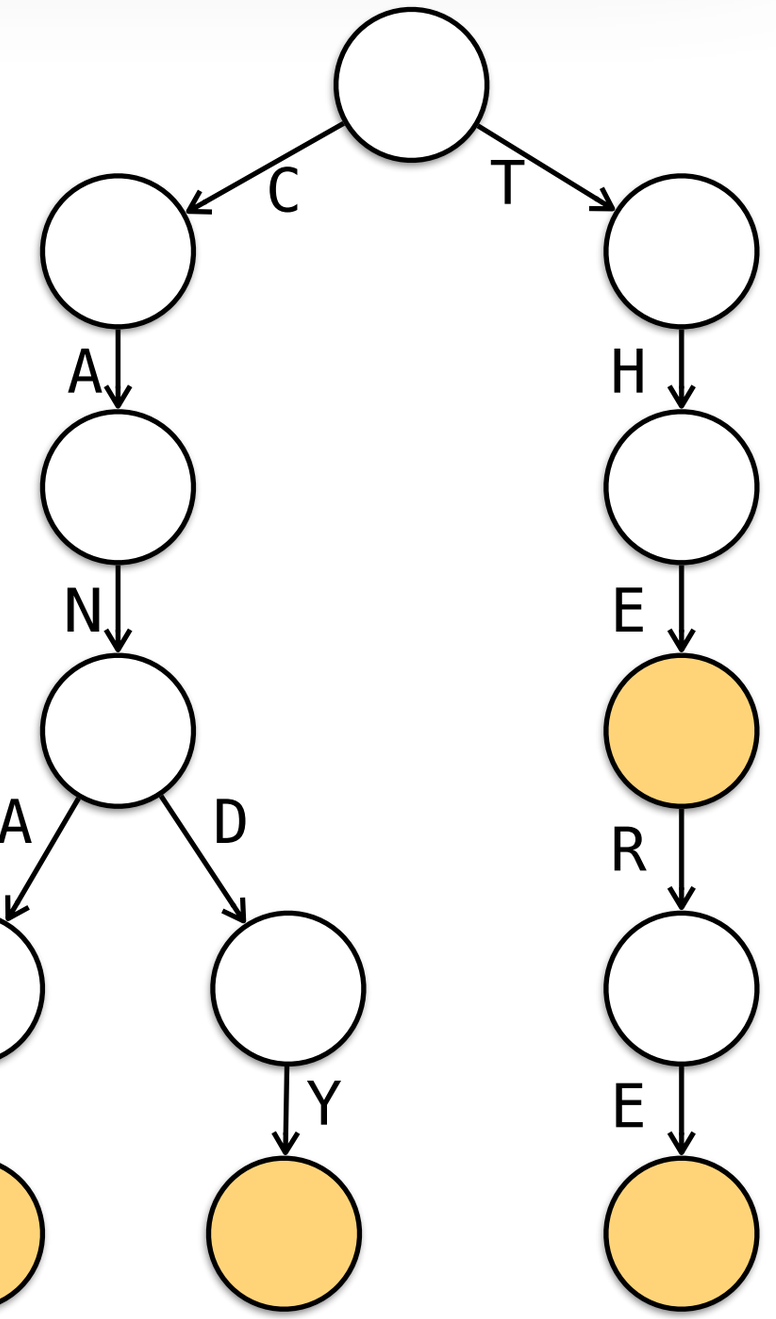
```
S = {CANAL, CANDY, THE, THERE}
```

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



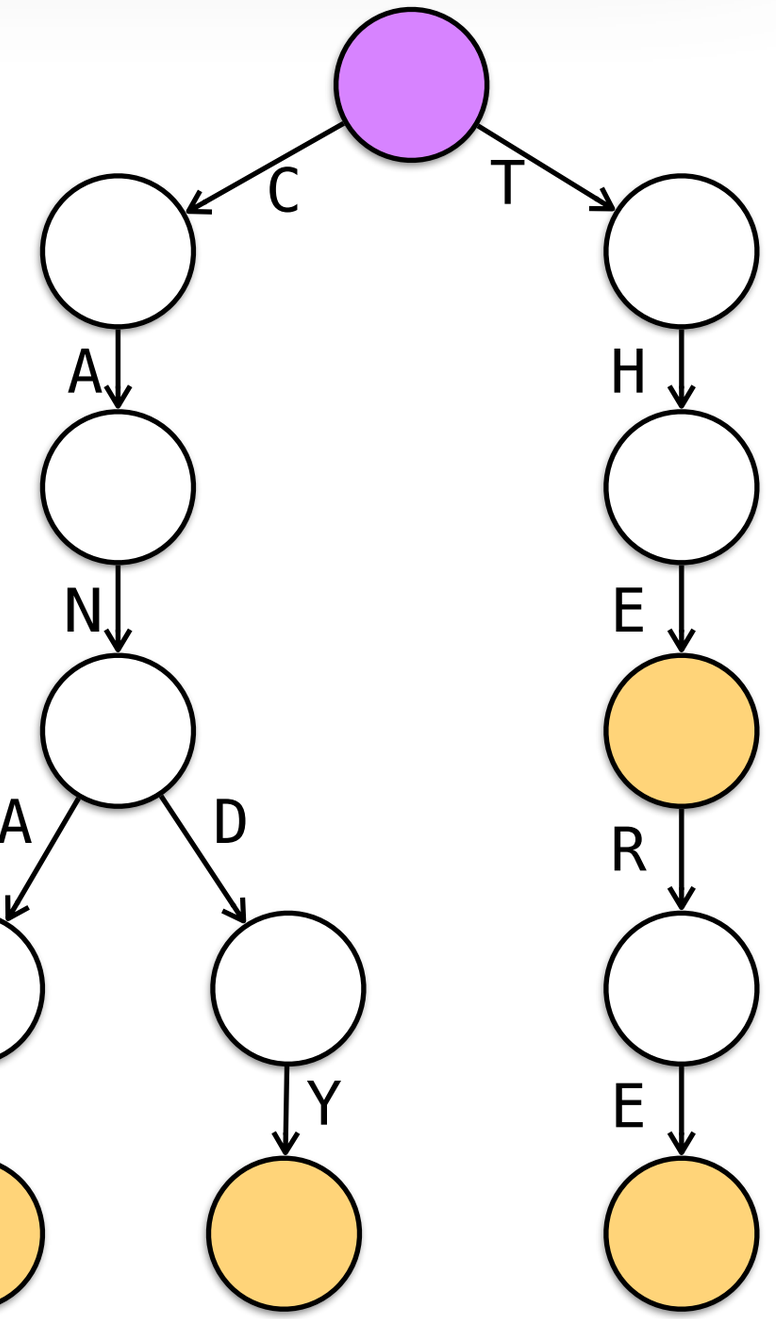
S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



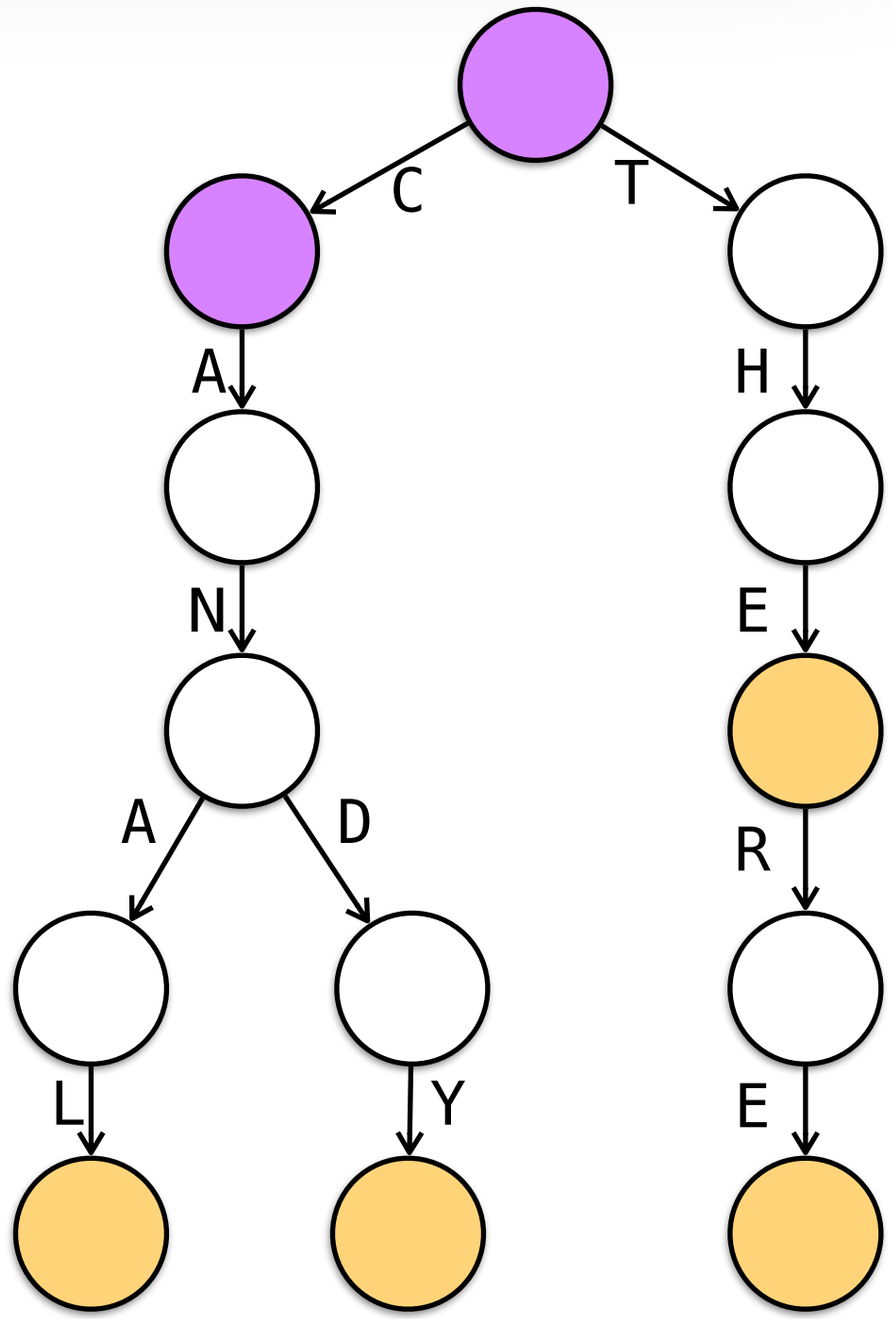
S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



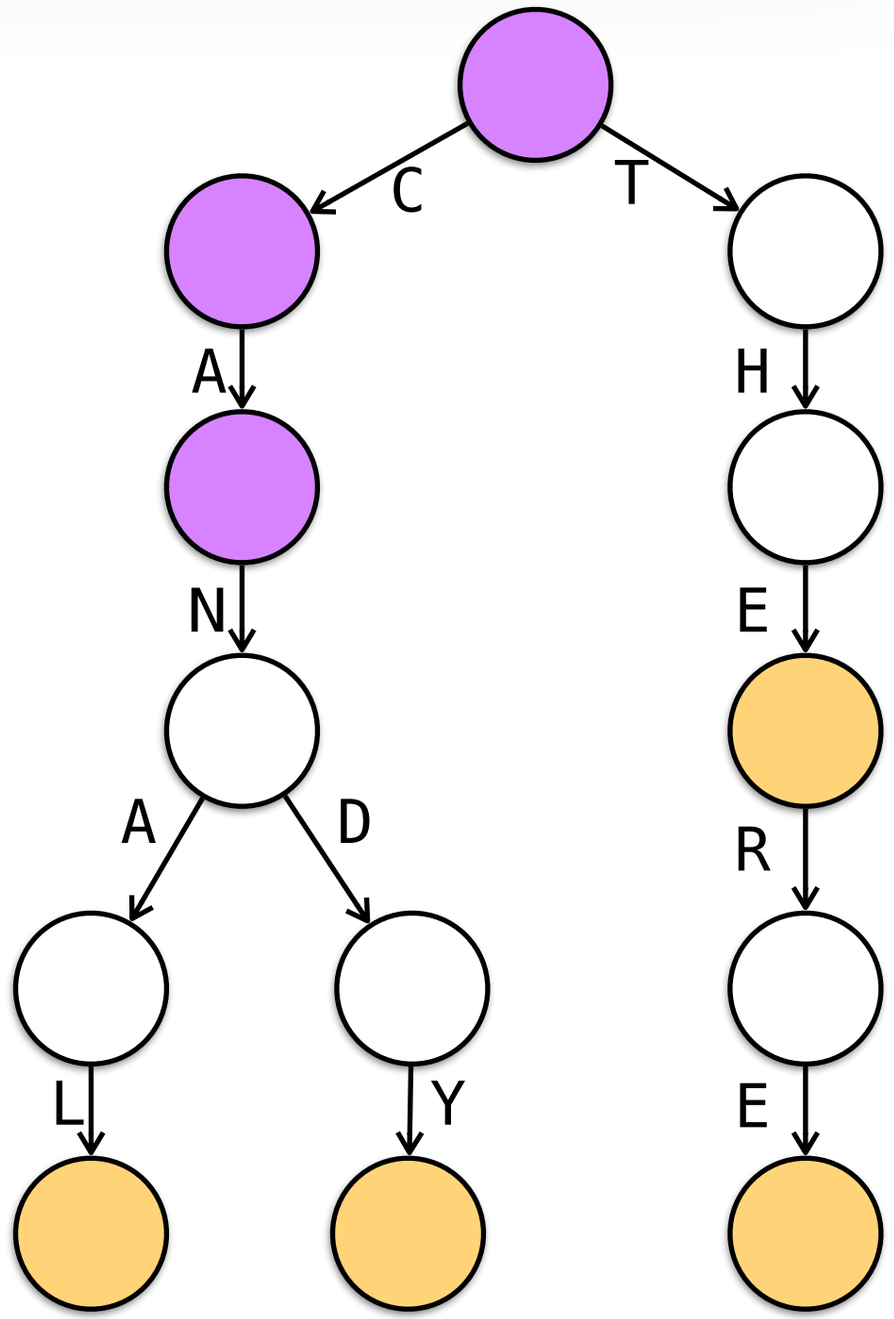
S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



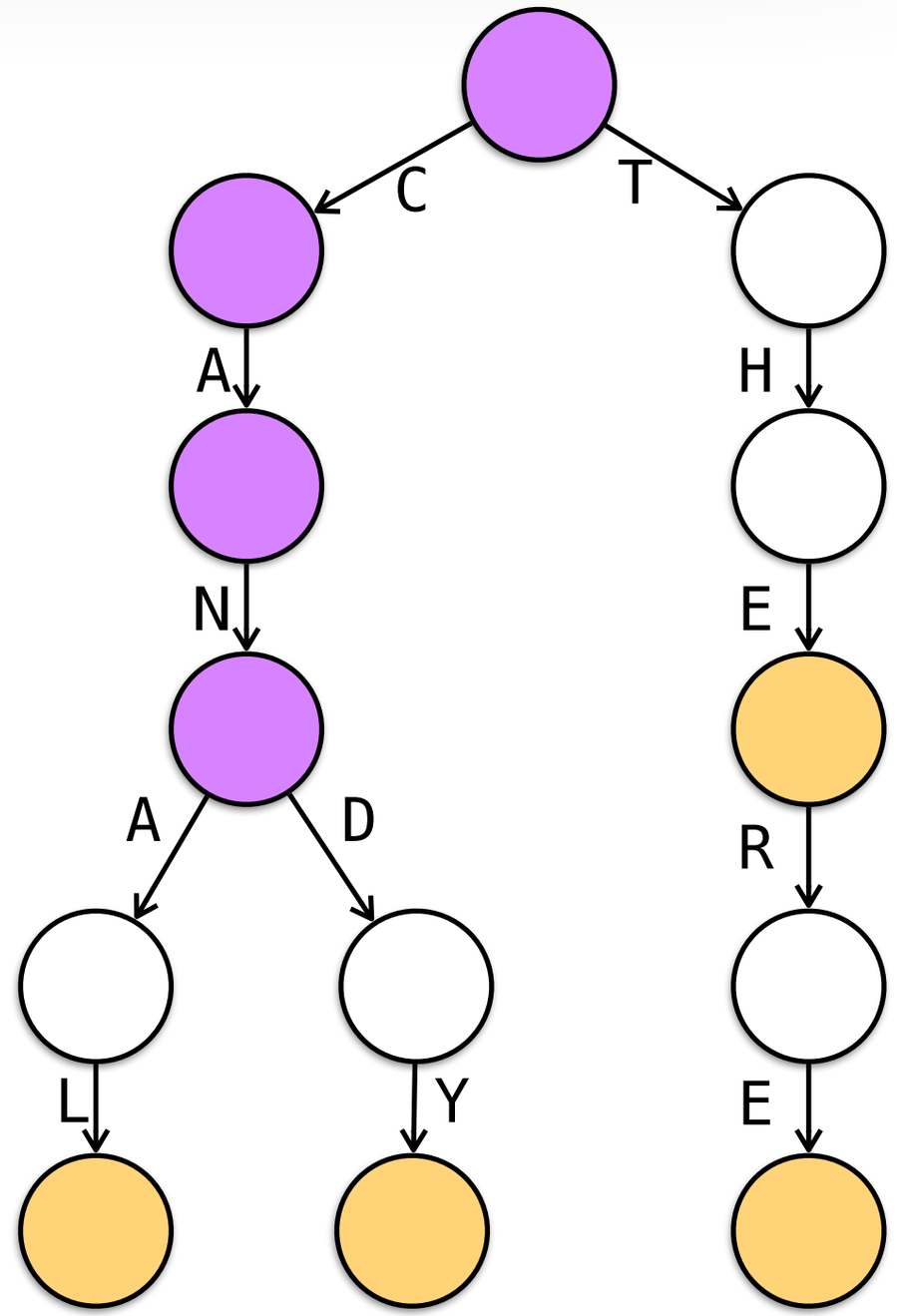
S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



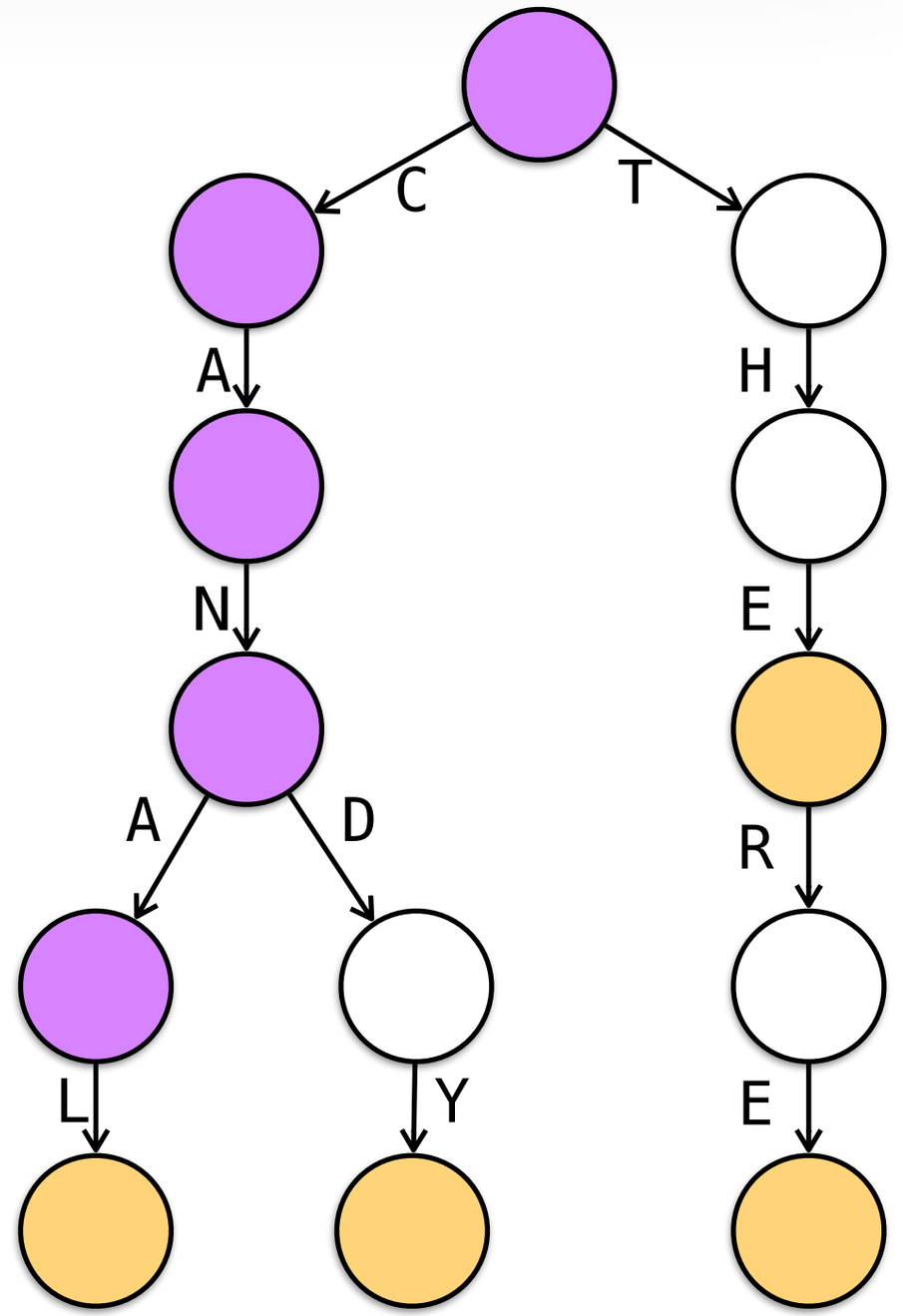

```
S = {CANAL, CANDY, THE, THERE}
```

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



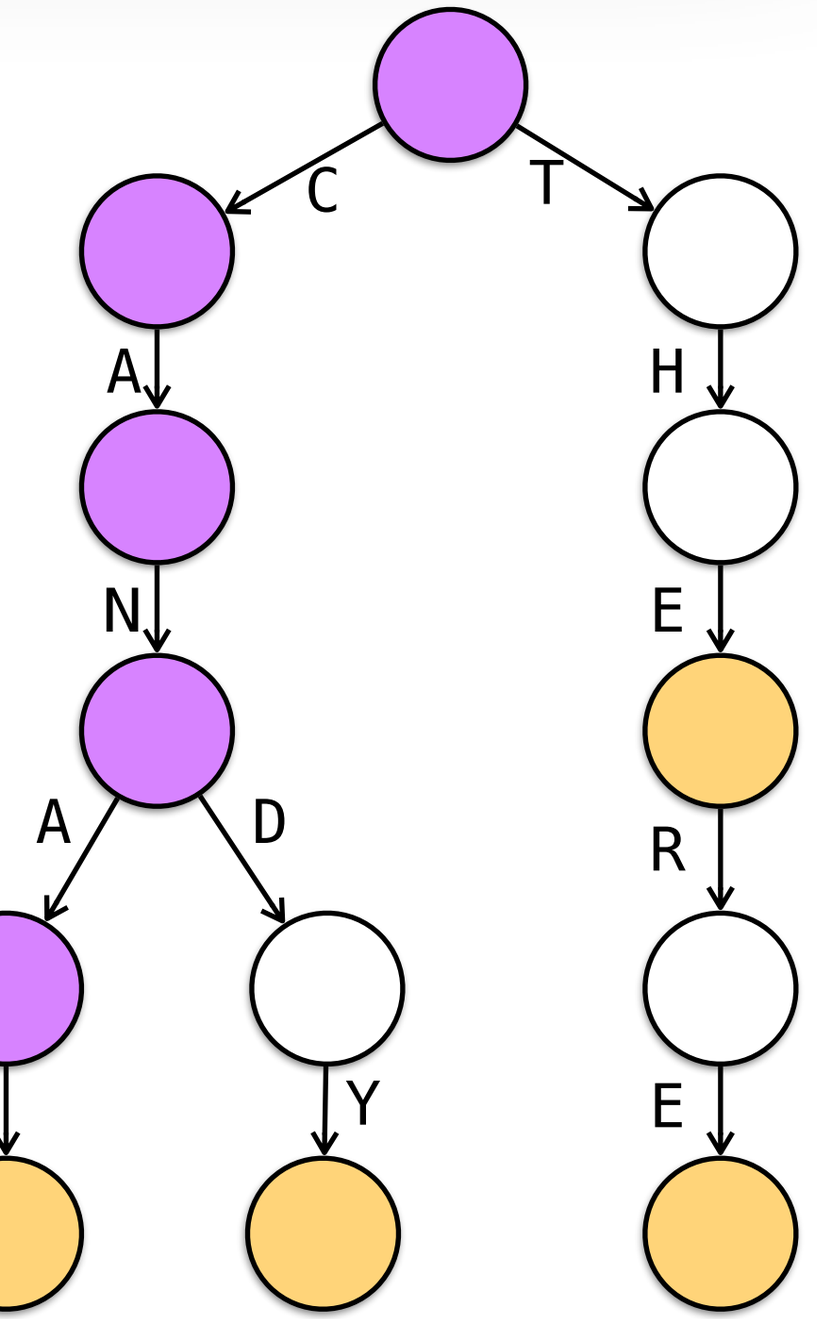
S = {CANAL, CANDY, THE, THERE}

Tries

Suffix tries are used as dictionaries (to find words)

CAN?

CANARY?



S = {CANAL, CANDY, THE, THERE}

Tries

```
int trie[INT_MAX][A];
```

Tries

```
int trie[INT_MAX][A];
void buildTrie(string s) {
    int i=0, v=0;
    while(i<s.size()) {
        if(trie[v][s[i]-65] == -1) {
            v = trie[v][s[i++]-65] = following++;
        } else {
            v = trie[v][s[i++]-65];
        }
    }
}
```

Tries

```
int trie[INT_MAX][A];
void buildTrie(string s) {
    int i=0, v=0;
    while(i<s.size()) {
        if(trie[v][s[i]-65] == -1) {
            v = trie[v][s[i++]-65] = following++;
        } else {
            v = trie[v][s[i++]-65];
        }
    }
}
```

try S = {CAR, CAT}

Tries

```
def insert(self,key):
        # If not present, inserts key into trie
        # If the key is prefix of trie node,
        # just marks leaf node
        pCrawl = self.root
        length = len(key)
    for level in range(length):
        index = self._charToIndex(key[level])
        # if current character is not present
        if not pCrawl.children[index]:
            pCrawl.children[index] = self.getNode()
        pCrawl = pCrawl.children[index]
    # mark last node as leaf
    pCrawl.isEndOfWord = True
```

Application of tries

- Word search and generation
- Tries are very useful to find **longest common prefixes** (as the least common ancestor of the words in the dictionary)
- Data analytics

UVa 11362 - Phone list

Given a list of phone numbers, determine if it is consistent in the sense that no number is the prefix of another. Let's say the phone catalogue listed these numbers:

- Emergency 911
- Alice 97 625 999
- Bob 91 12 54 26

In this case, it's not possible to call Bob, because the central would direct your call to the emergency line as soon as you had dialled the first three digits of Bob's phone number. So this list would not be consistent.

UVa 11362 - Phone list

Input

The first line of input gives a single integer, $1 \leq t \leq 40$, the number of test cases. Each test case starts with n, the number of phone numbers, on a separate line, $1 \leq n \leq 10000$. Then follows n lines with one unique phone number on each line. A phone number is a sequence of at most ten digits.

Output

For each test case, output ‘YES’ if the list is consistent, or ‘NO’ otherwise.

UVa 11362 - Phone list

```
fun main() {
    val In = BufferedReader(InputStreamReader(System.`in`))
    val t = In.readLine()!!.toInt()
    for(i in 1..t) {
        val n = In.readLine()!!.toInt()
        val trie = Node(0)
        var ans = true
        for(j in 1..n) {
            val s = In.readLine()!!
            ans = buildTrie(trie!!, s.trim())
            if(!ans) {
                println("NO")
                break
            }
        }
        if(ans)
            println("YES")
    }
}
```

UVa 11362 - Phone list

```
data class Node(val id : Int) {
    var end = false
    val links = arrayOfNulls<Node>(10)
}
```

UVa 11362 - Phone list

```
tailrec fun buildTrie(trie : Node, s : String) : Boolean {
    if(s.isEmpty()) {
        trie.end = true
        val l = trie.links.filter { x -> x != null }
        return l.isEmpty()
    }else {
        if(trie.end) {
            return !trie.end
        } else {
            when(trie.links[s.first().toInt() -48]) {
                null -> {
                    val n = Node(s.first().toInt()-48)
                    trie.links[s.first().toInt()-48] = (n)
                    n
                }
                else -> trie.links[s.first().toInt()-48]
            }
            return buildTrie(trie.links[s.first().toInt()-48]!!, s.drop(1))
        }

    }
}
```

SUFFIX TREES

Suffix trees

Definition A suffix tree for a string S is a tree representation of the suffixes of a **larger** string.

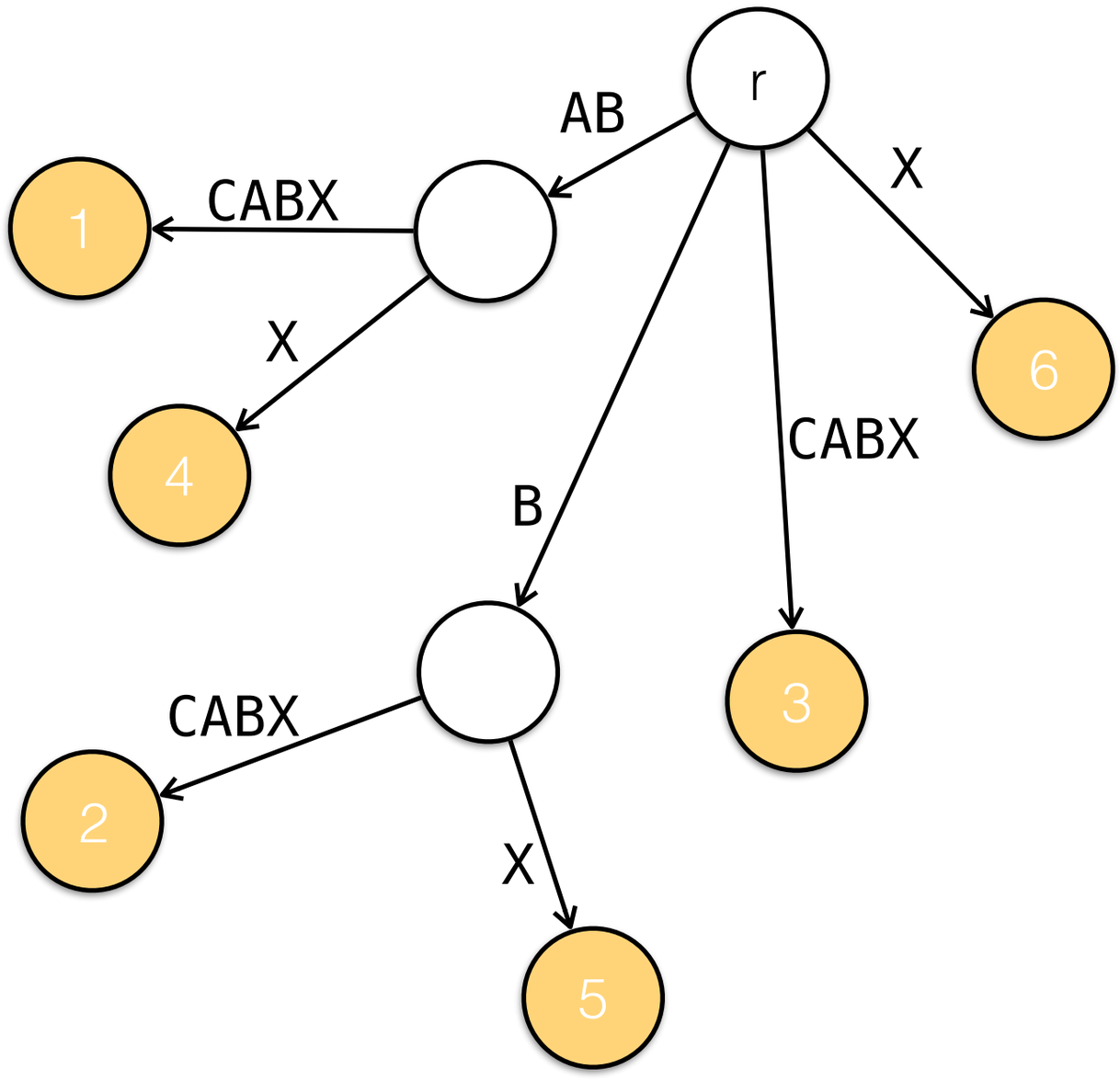
Edges have substrings of S

Leaves have the position of the suffix (exactly `S.length`)

Vertices have at least two children

Edges from a source have different starting characters

`suffix(i, m)` contains the path to node `i`



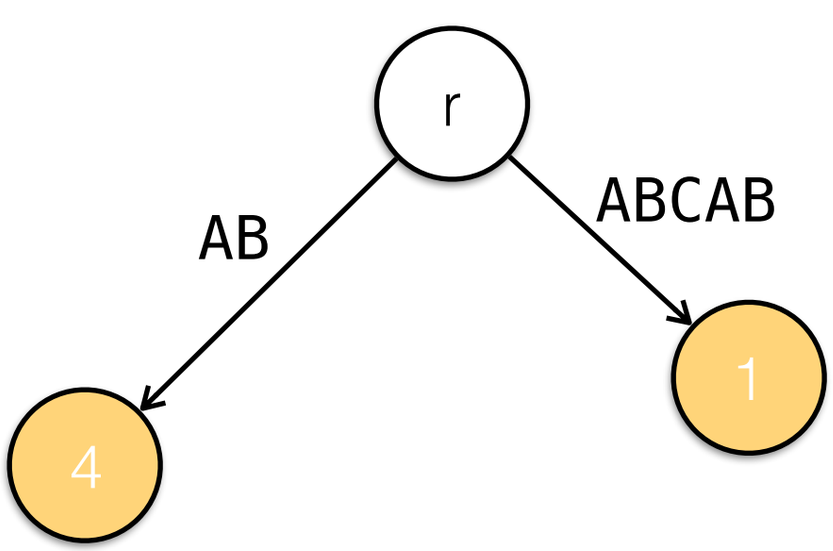
S = ABCABX

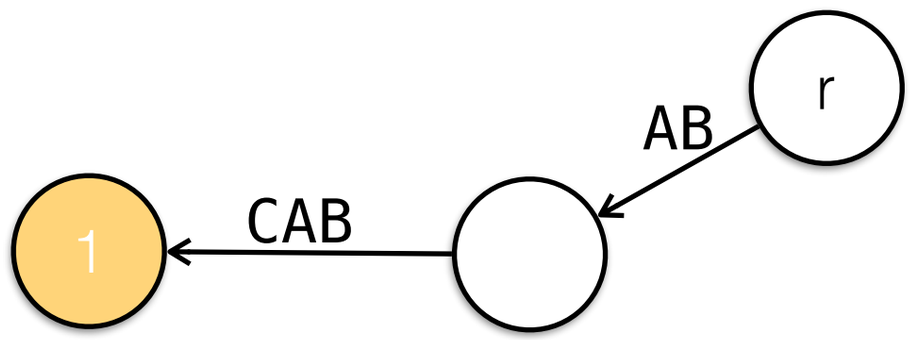
Suffix trees

The **predecesor** problem

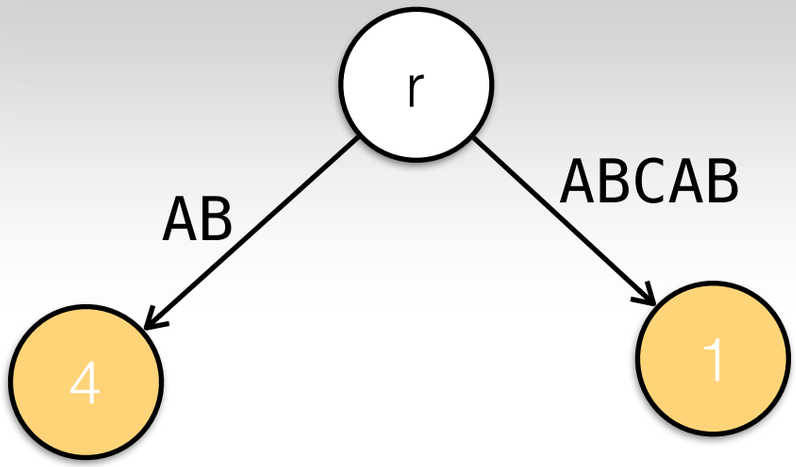
Consider S = {abcab}

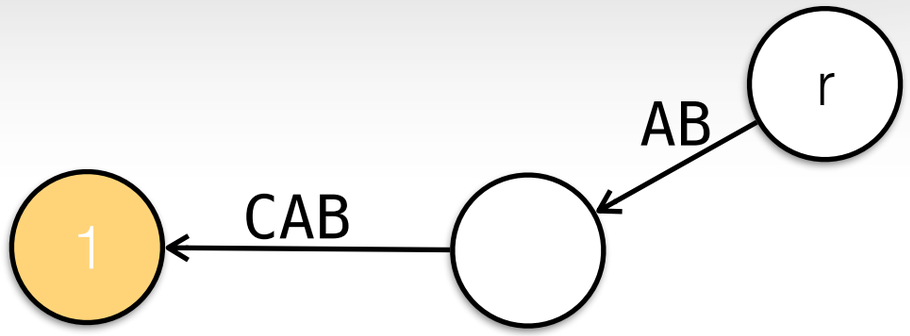
the suffix ab matches the prefix of suffix abcac





Tree



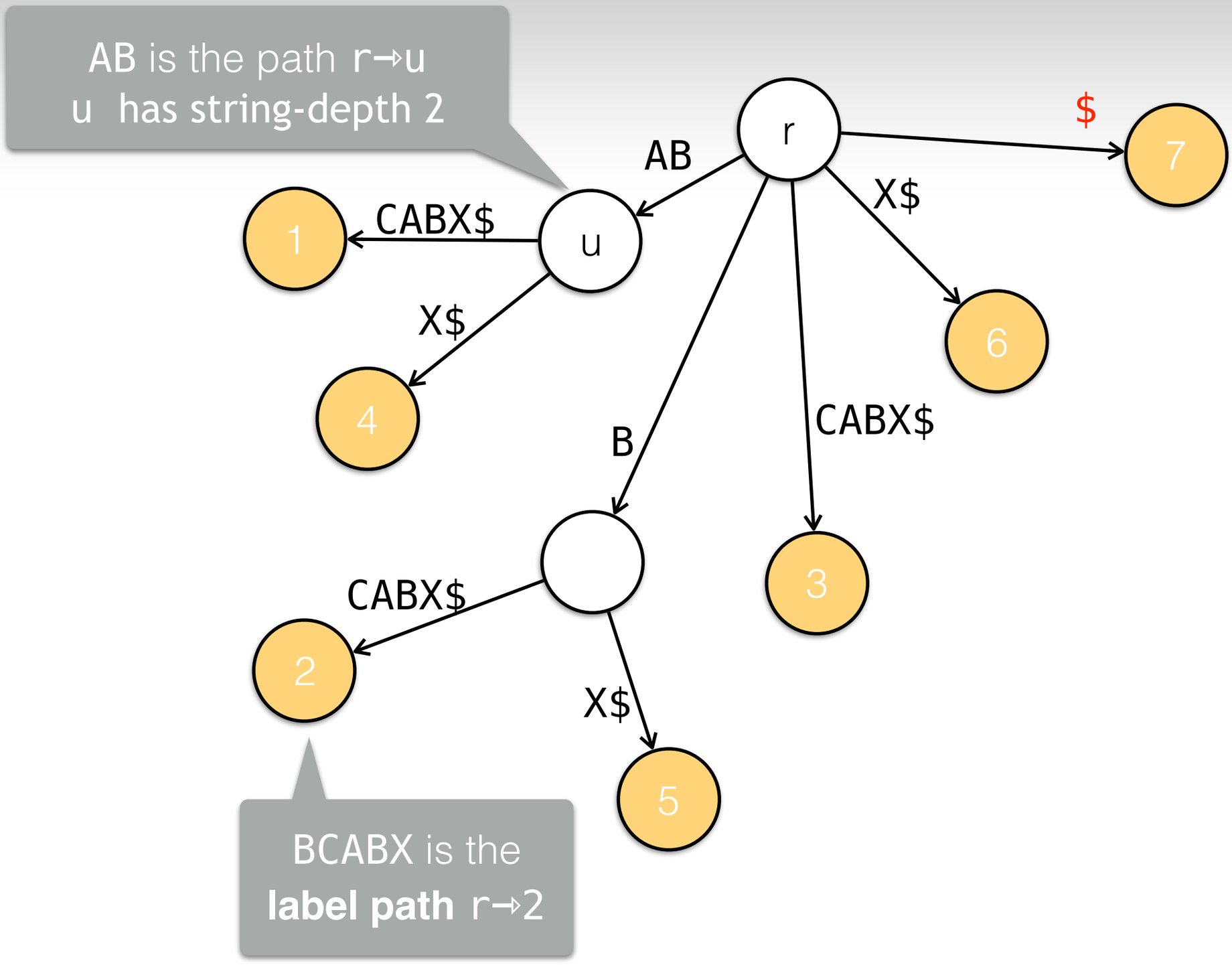


To avoid such problems we add a **termination $** character at the end of every word

S = abcab**$**

$
b$
ab$
cab$
bcab$
abcab$

Tree


AB is the path r→u
u has string-depth 2
r
AB
$
7
X$
u
1
CABX$
X$
6
4
B
CABX$
3
CABX$
2
X$
5
BCABX is the
label path r→2

Tree building

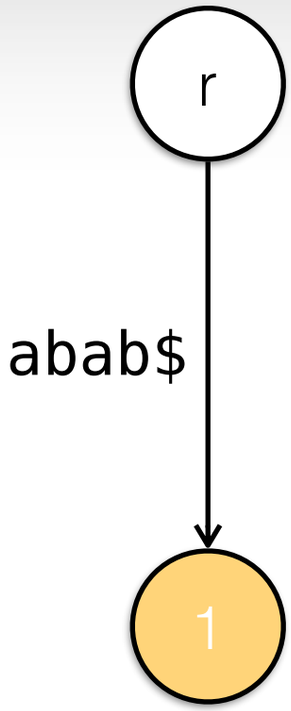
1. Start by inserting S to the tree, `T(S[1..n]$)` which has a single node

2. If `Ti` exists, Traverse the tree from the root to find the **longest prefix** that matches a path in `Ti`.
    1. If no path exists, create a new leaf i+1 with edge `S[i+1..n]$`.
    2. Otherwise, find k such that `S[i+1..k]` is a prefix of `S[i+1..n]`. Let `(u,v)` be the string containing k. Create a new node w and a leaf `i+1` such that `(u,v)` is split into `(u,w)` and `(w,v)`, and add a new edge `(w, i+1)`

3. The resulting tree is T_{i+1}

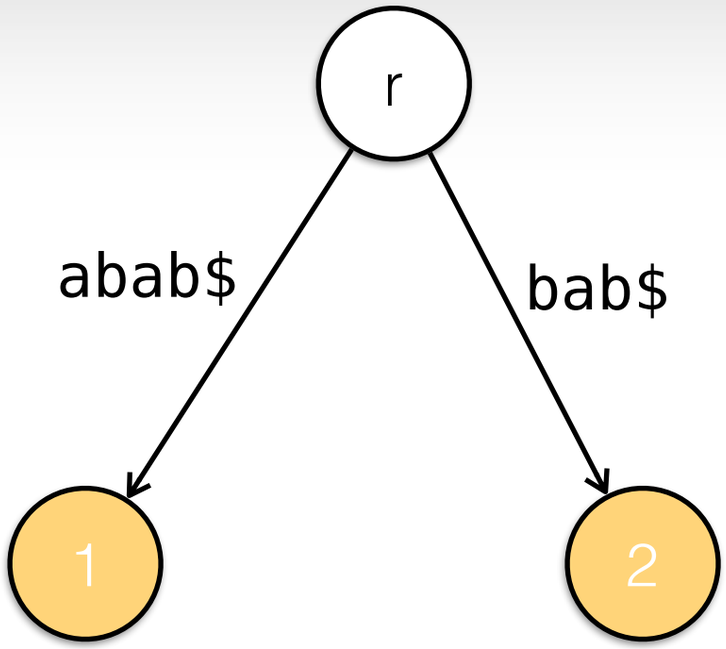
Tree building

S = {abab$}



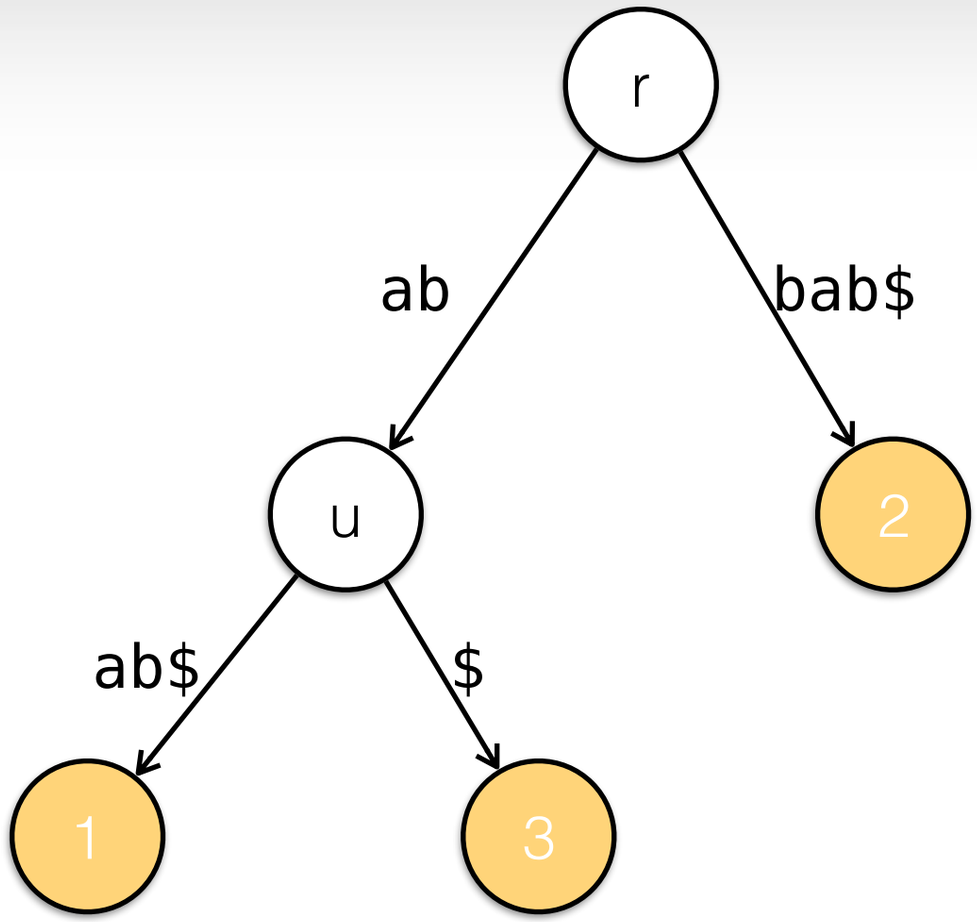
Tree building

S = {abab$}



Tree building

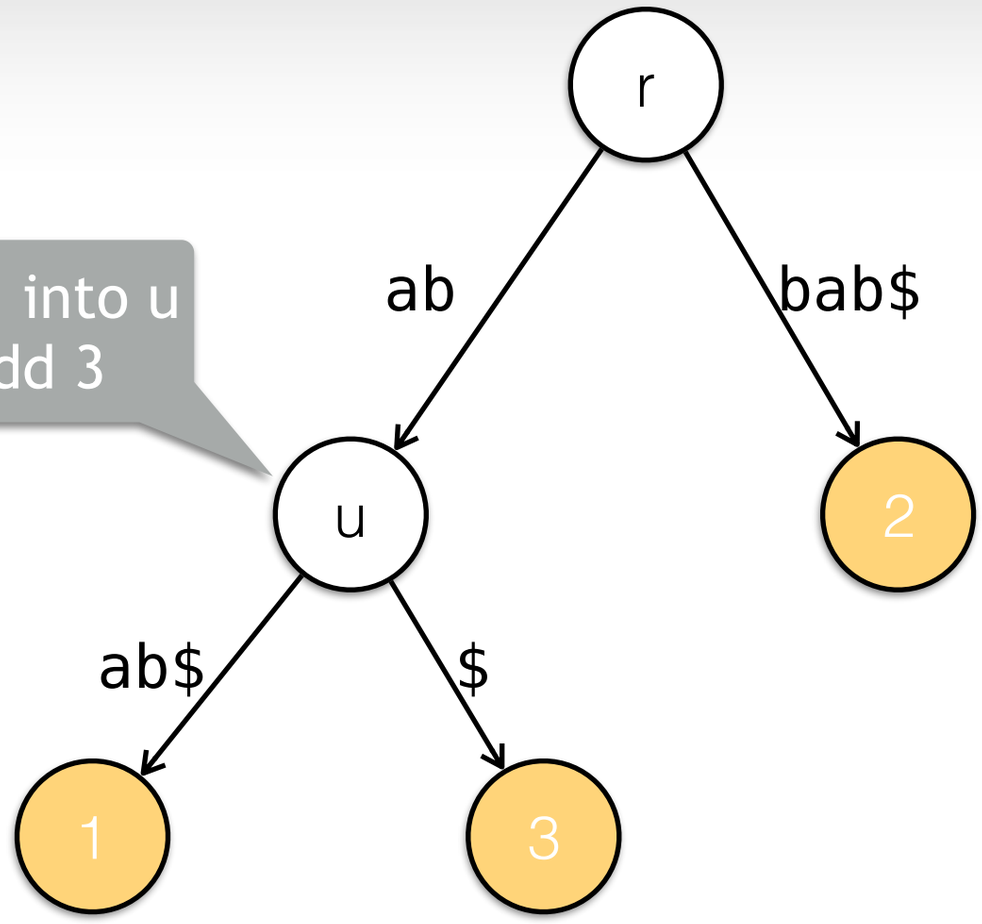
S = {abab$}



Tree building

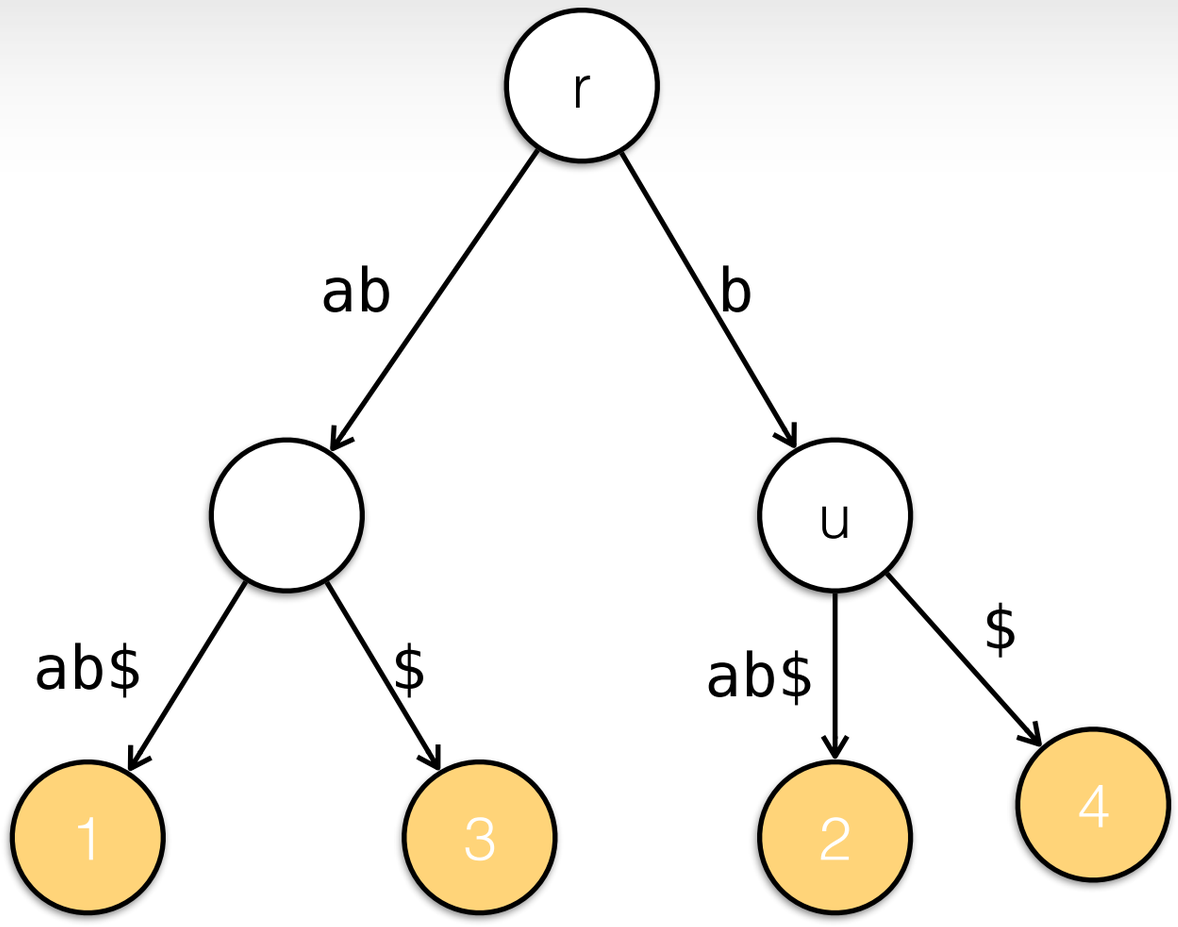
S = {abab$}

split the node 1 into u and 1, and add 3



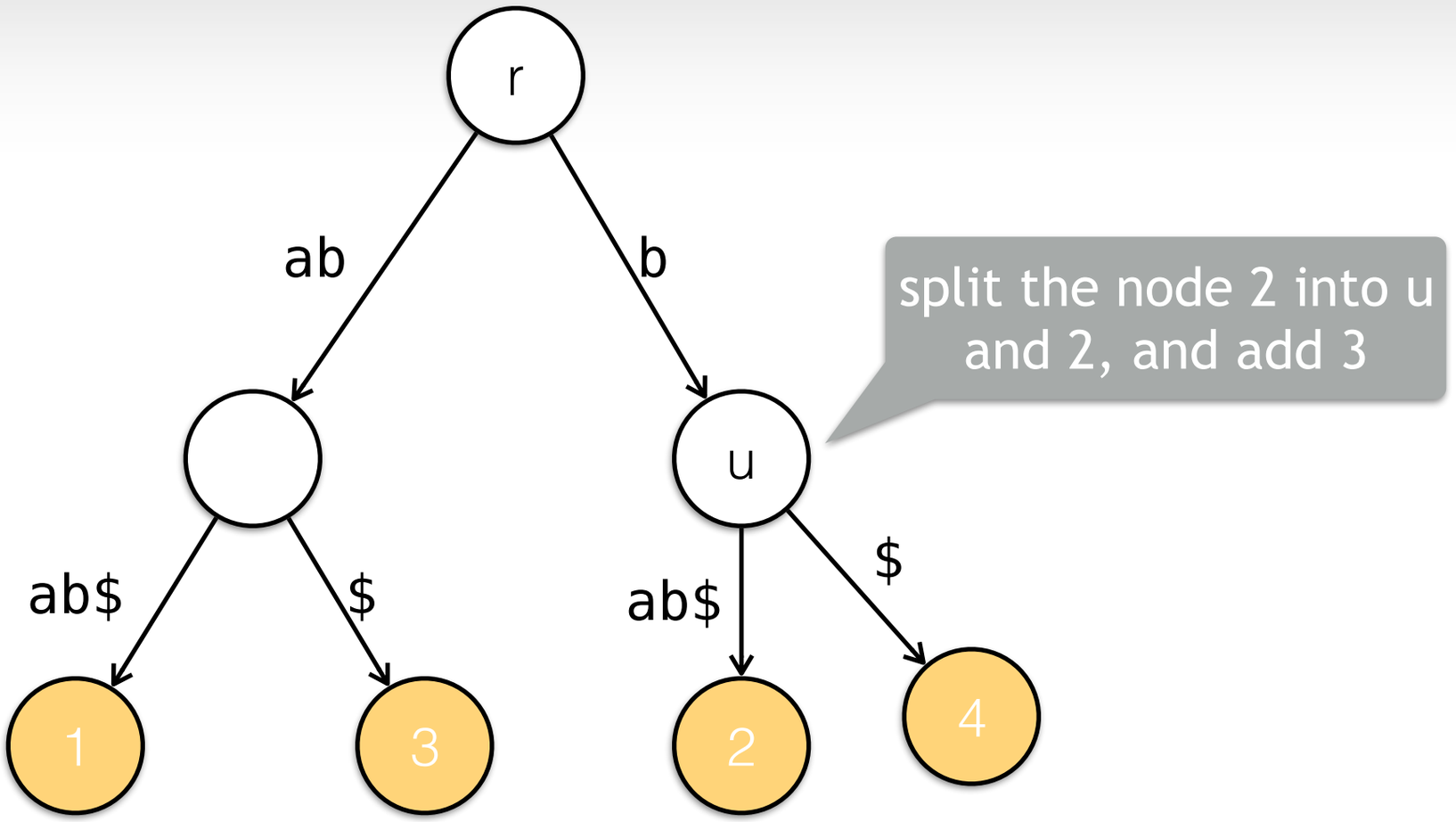
Tree building

S = {abab$}



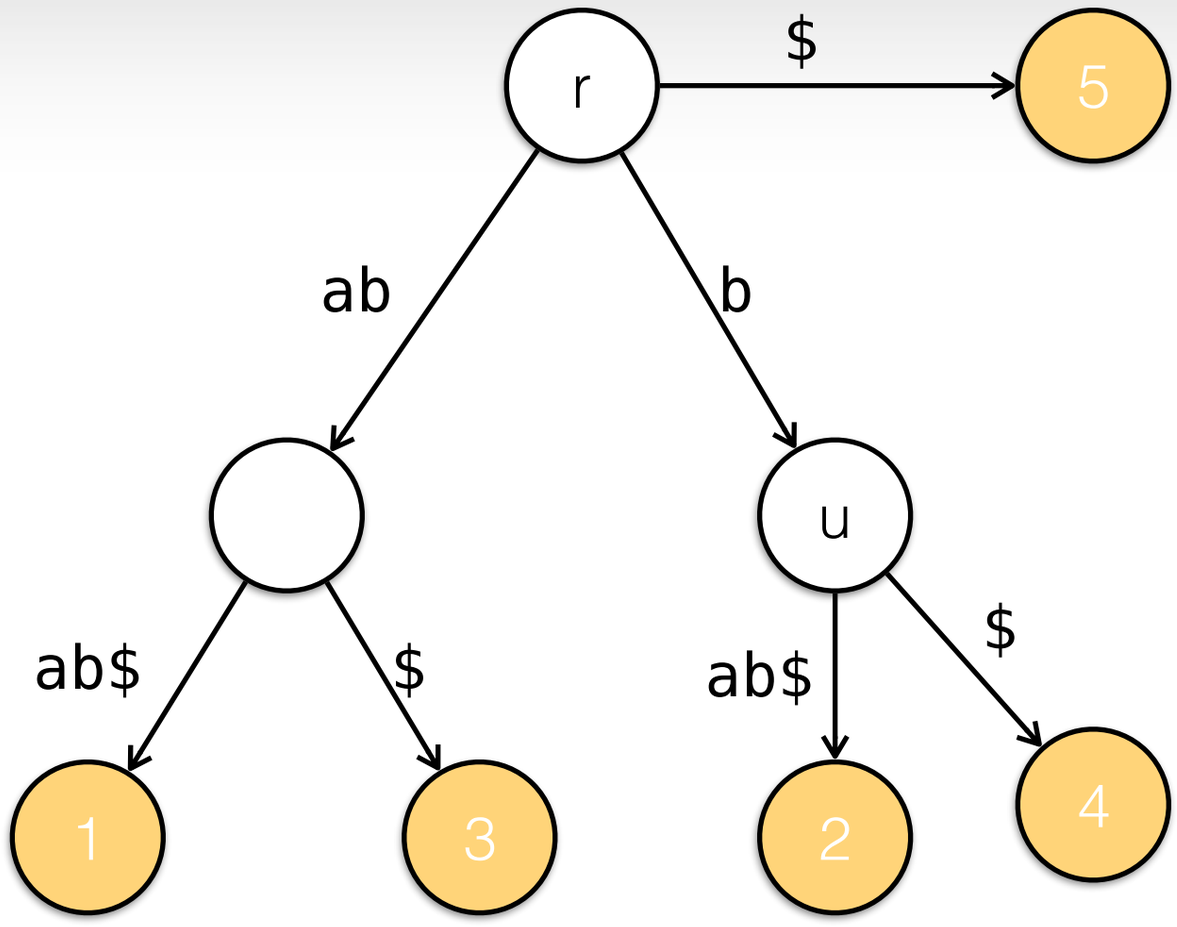
Tree building

S = {abab$}



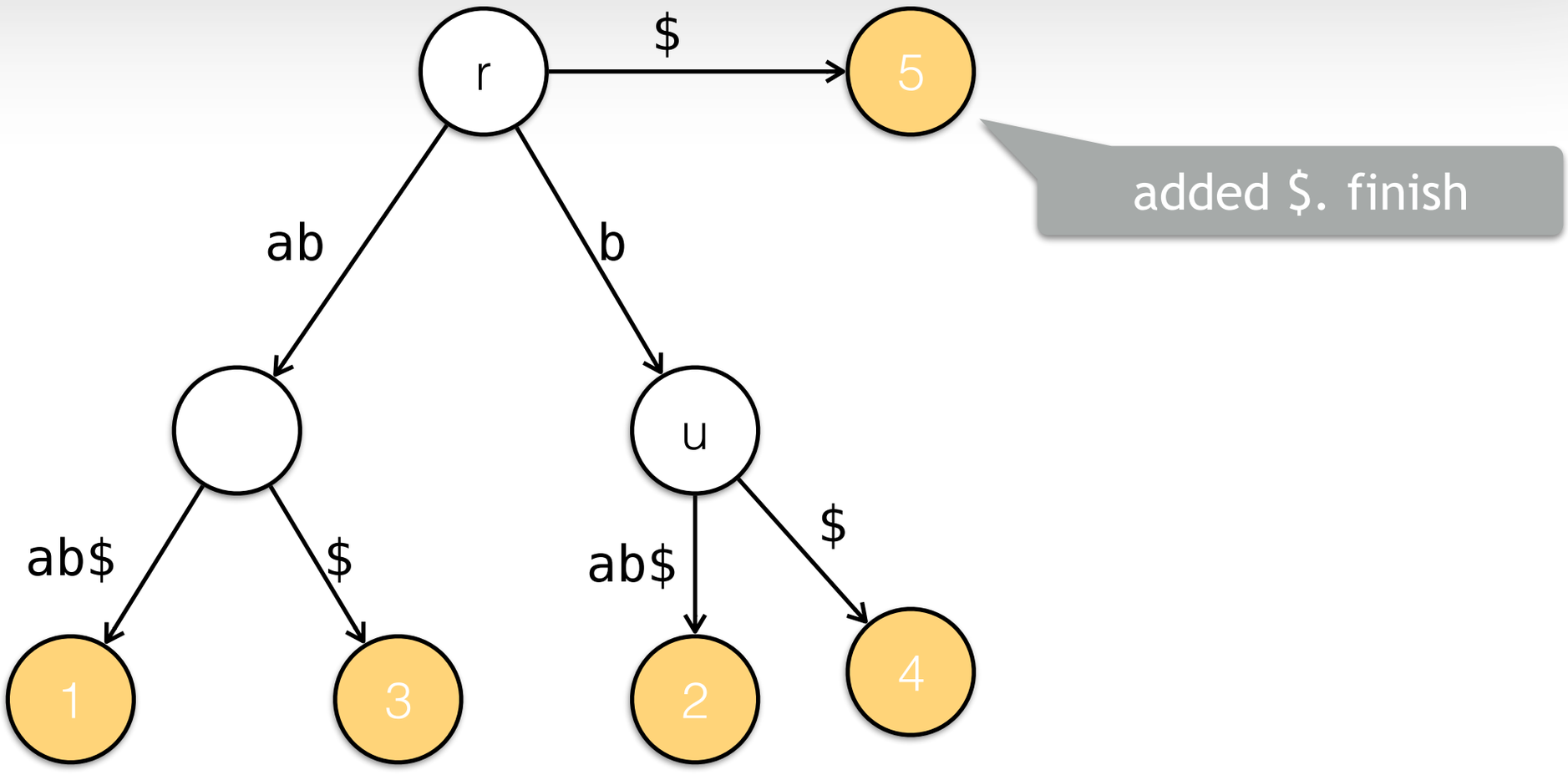
Tree building

S = {abab$}



Tree building

S = {abab$}



Tree building

```
private void addSuffix(String suf) {
    int n = 0;
    int i = 0;
    while (i < suf.length()) {
        char b = suf.charAt(i);
        List<Integer> children = nodes.get(n).ch;
        int x2 = 0;
        int n2;
        while (true) {
            if (x2 == children.size()) {
                n2 = nodes.size();
                Node temp = new Node();
                temp.sub = suf.substring(i);
                nodes.add(temp);
                children.add(n2);
                return;
            }
            n2 = children.get(x2);
            if (nodes.get(n2).sub.charAt(0) == b) break;
            x2++;
        }
        String sub2 = nodes.get(n2).sub;
        int j = 0;
        while (j < sub2.length()) {
           if (suf.charAt(i + j) != sub2.charAt(j)) {
              int n3 = n2; // split n2
              n2 = nodes.size(); // new node for the part in common
              Node temp = new Node();
              temp.sub = sub2.substring(0, j);
              temp.ch.add(n3);
              nodes.add(temp);
              nodes.get(n3).sub = sub2.substring(j); // old node loses the part in common
              nodes.get(n).ch.set(x2, n2);
              break;  // continue down the tree
           }
           j++;
        }
        i += j;  // advance past part in common
        n = n2;  // continue down the tree
    }
}
```

Tree building

```
private void addSuffix(String suf) {
    int n = 0;
    int i = 0;
    while (i < suf.length()) {
        char b = suf.charAt(i);
        List<Integer> children = nodes.get(n).ch;
        int x2 = 0;
        int n2;
        while (true) {
            if (x2 == children.size()) {
                n2 = nodes.size();
                Node temp = new Node();
                temp.sub = suf.substring(i);
                nodes.add(temp);
                children.add(n2);
                return;
            }
            n2 = children.get(x2);
            if (nodes.get(n2).sub.charAt(0) == b) break;
            x2++;
        }
        String sub2 = nodes.get(n2).sub;
        int j = 0;
        while (j < sub2.length()) {
           if (suf.charAt(i + j) != sub2.charAt(j)) {
              int n3 = n2; // split n2
              n2 = nodes.size(); // new node for the part in common
              Node temp = new Node();
              temp.sub = sub2.substring(0, j);
              temp.ch.add(n3);
              nodes.add(temp);
              nodes.get(n3).sub = sub2.substring(j); // old node loses the part in common
              nodes.get(n).ch.set(x2, n2);
              break;  // continue down the tree
           }
           j++;
        }
        i += j;  // advance past part in common
        n = n2;  // continue down the tree
    }
}
```

O(☹)

Tree building

```
private void addSuffix(String suf) {
    int n = 0;
    int i = 0;
    while (i < suf.length()) {
        char b = suf.charAt(i);
        List<Integer> children = nodes.get(n).ch;
        int x2 = 0;
        int n2;
        while (true) {
            if (x2 == children.size()) {
                n2 = nodes.size();
                Node temp = new Node();
                temp.sub = suf.substring(i);
                nodes.add(temp);
                children.add(n2);
                return;
            }
            n2 = children.get(x2);
            if (nodes.get(n2).sub.charAt(0) == b) break;
            x2++;
        }
        String sub2 = nodes.get(n2).sub;
        int j = 0;
        while (j < sub2.length()) {
           if (suf.charAt(i + j) != sub2.charAt(j)) {
              int n3 = n2; // split n2
              n2 = nodes.size(); // new node for the part in common
              Node temp = new Node();
              temp.sub = sub2.substring(0, j);
              temp.ch.add(n3);
              nodes.add(temp);
              nodes.get(n3).sub = sub2.substring(j); // old node loses the part in common
              nodes.get(n).ch.set(x2, n2);
              break;  // continue down the tree
           }
           j++;
        }
        i += j;  // advance past part in common
        n = n2;  // continue down the tree
    }
}
```

O(1)	😎
O(log n)	😁
O(n)	😐
O(nlog n)	🙁
O(n²)	☹️
O(2ⁿ)	😱
O(n!)	🖕

O(☹️)

Ukkonen's algorithm

Building Suffix trees in the linear time is cumbersome and requires to go over several structures and optimizations of the quadratic code

1. Build an **implicit tree**
2. Use **suffix links**

Ukkonen's algorithm

Ukkonen's algorithm builds trees by taking prefixes S[1..i] (building **implicit trees** T_i) and introducing the **suffix extensions** S[j..i]

1. Find the path of suffix S[j..i]
2. Find the end of the path
3. Extend S[j..i] with S[i+1]

Ukkonen's algorithm

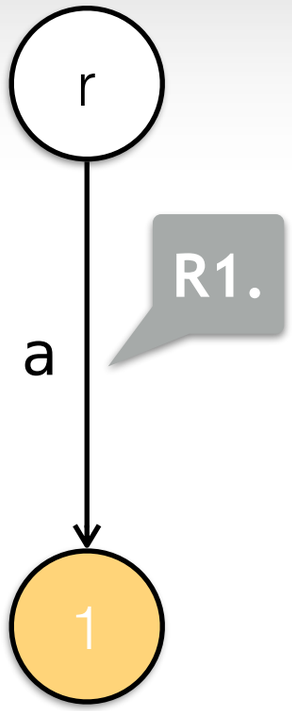
R1. Path containing S[j..i] ends at a leaf.
—> Add S[i+1]

R2. Path containing S[j..i] doesn't end at a leaf.
Next character in the label path is not S[i+1]
—> Split the edge and create node u and leaf j. Assign S[i+1] to (u, i+1).

R3. Path containing S[j..i] doesn't end at a leaf.
Next character in the label path **is** S[i+1]
—> nothing

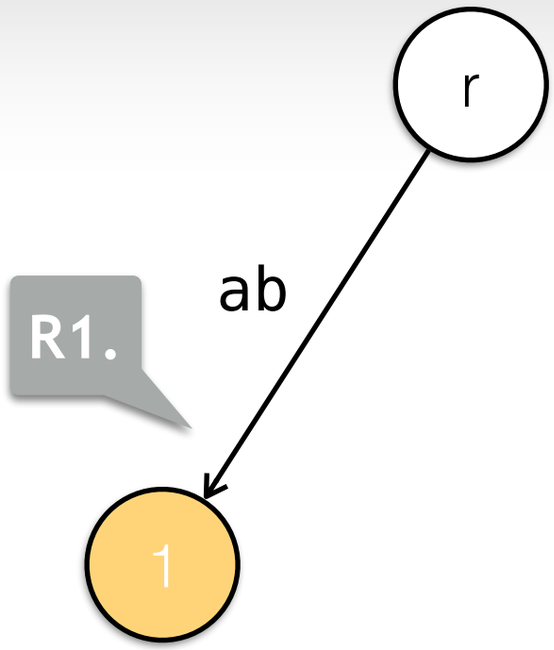
Implicit tree

S = {abba}



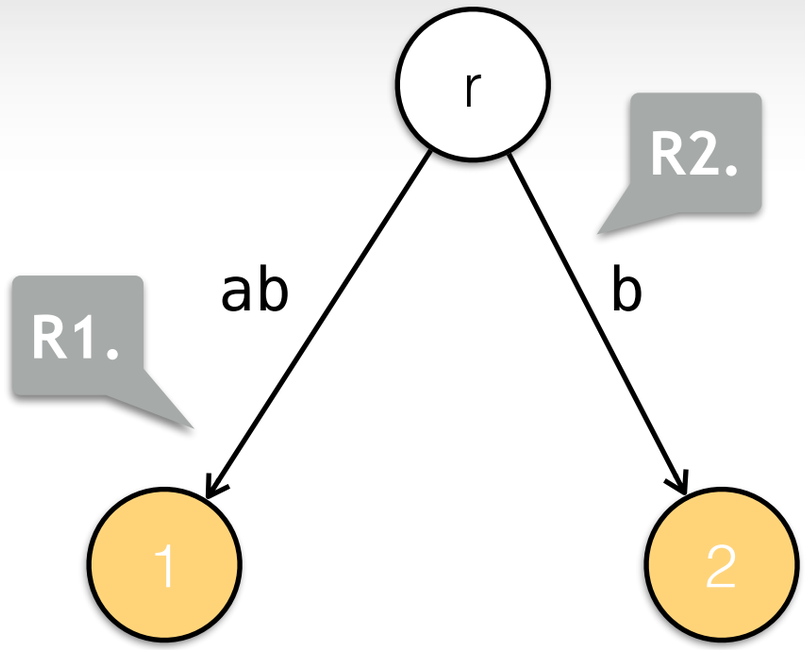
Implicit tree

S = {abba}



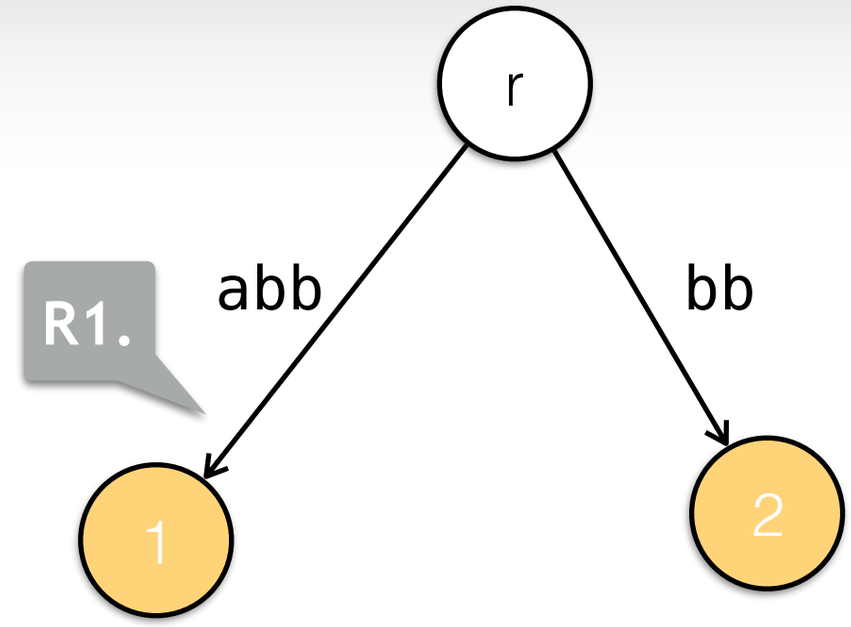
Implicit tree

S = {abba}



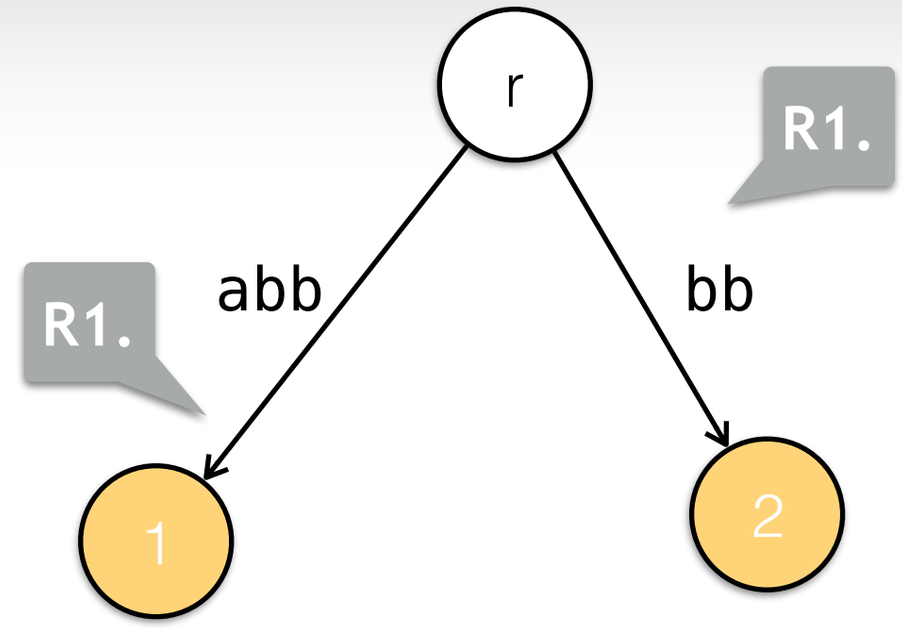
Implicit tree

S = {abba}



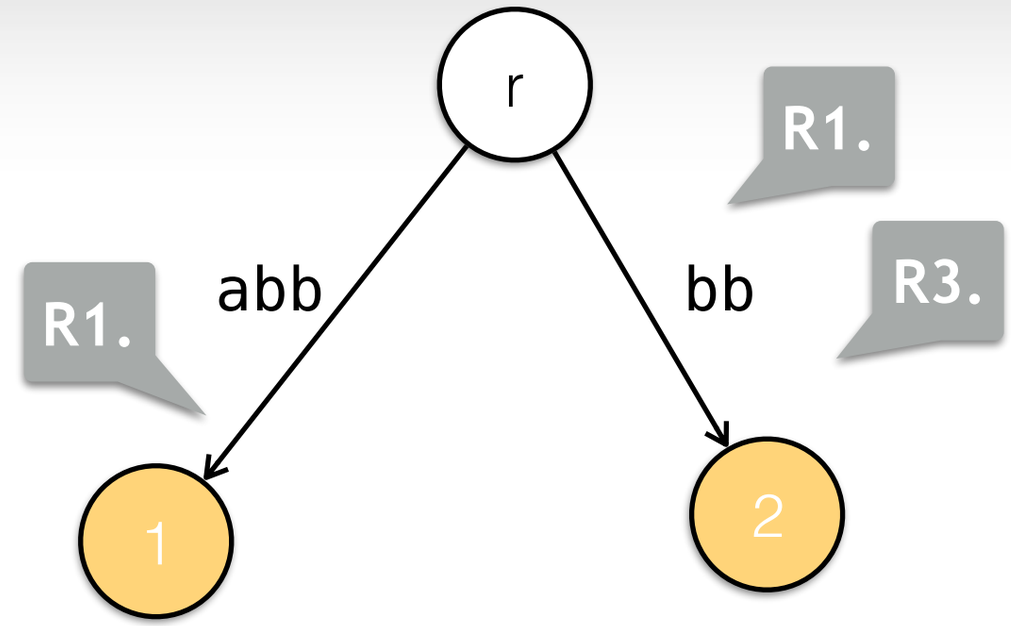
Implicit tree

S = {abba}



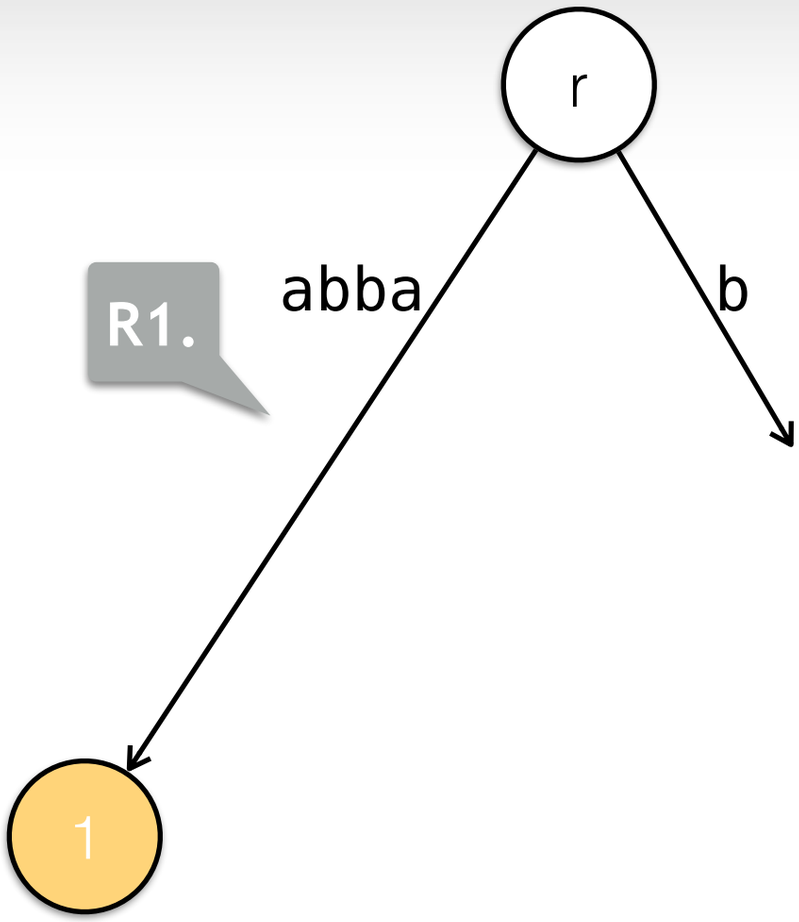
Implicit tree

S = {abba}



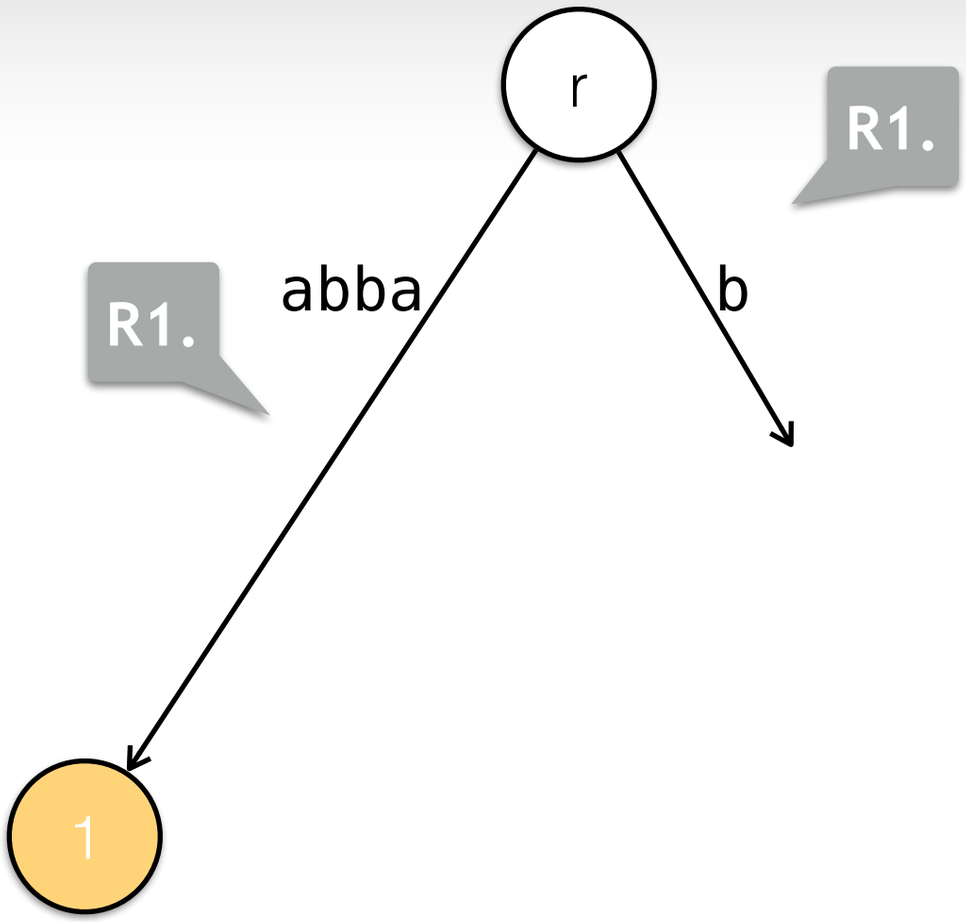
Implicit tree

S = {abba}



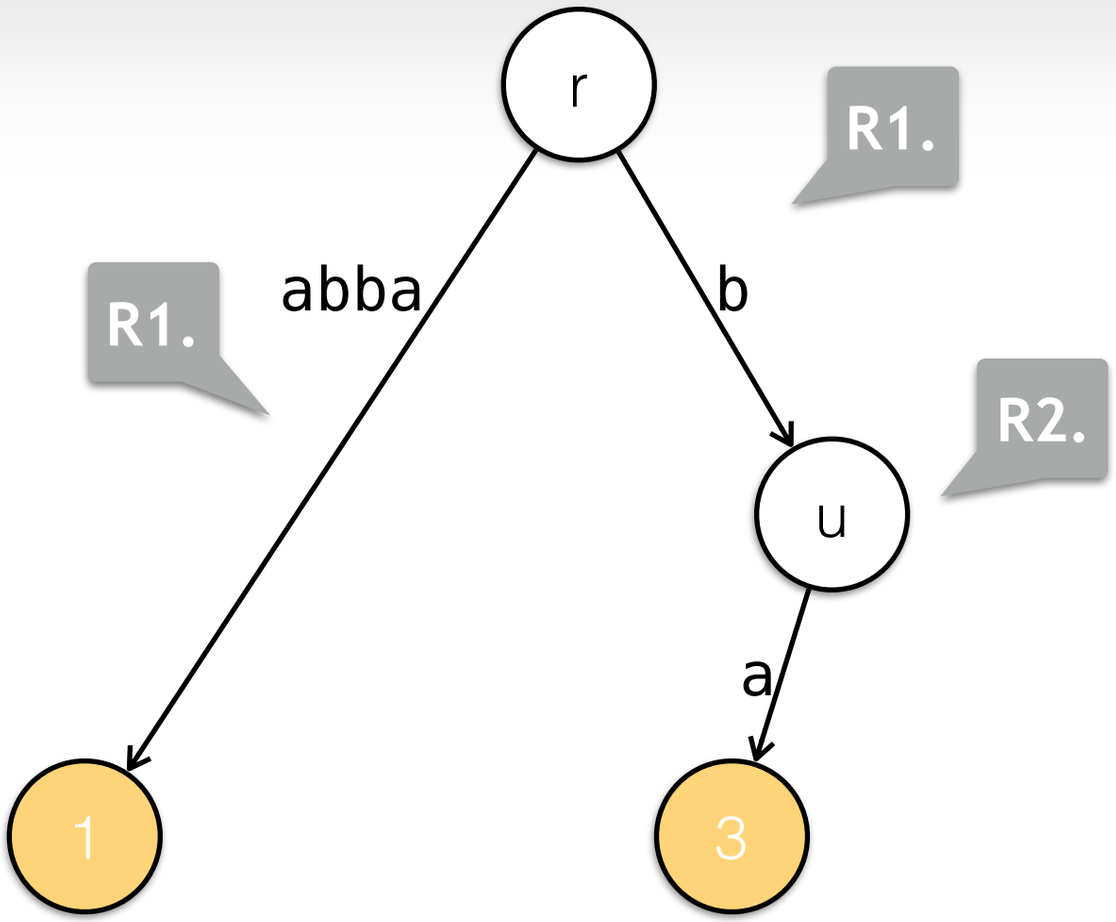
Implicit tree

S = {abba}



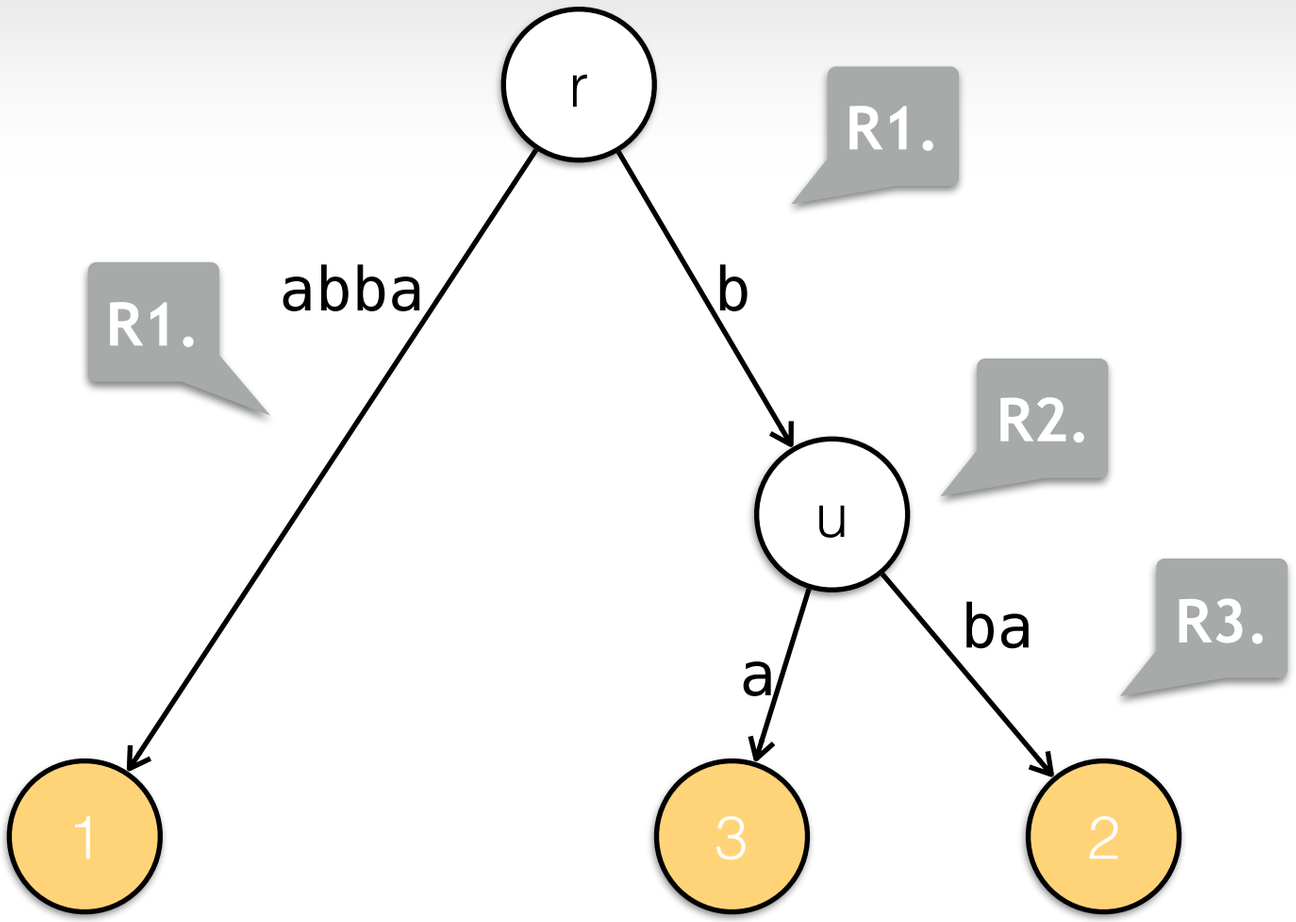
Implicit tree

S = {abba}



Implicit tree

S = {abba}



Implicit tree

S = {abba}

Char	Phase	Extension	Rule	Node	Label
{a}	1	1	2	(r,1)	a
{ab}	2	1	1	(r,1)	ab
{b}	2	2	2	(r,2)	b
{abb}	3	1	1	(r,1)	abb
{bb}	3	2	1	(r,2)	bb
{b}	3	3	3	(r,2)	b
{abba}	4	1	1	(r,1)	abba
{bba}	4	2	1	(r,2)	bba
ba	4	3	2	(u,3)	a
a	4	4	3	(u,1)	a

Implicit tree

S = {abba}

Char	Phase	Extension	Rule	Node	Label
{a}	1	1	2	(r,1)	a
{ab}	2	1	1	(r,1)	ab
{b}	2	2	2	(r,2)	b
{abb}	3	1	1	(r,1)	abb
{bb}	3	2	1	(r,2)	bb
{b}	3	3	3	(r,2)	b
{abba}	4	1	1	(r,1)	abba
{bba}	4	2	1	(r,2)	bba
ba	4	3	2	(u,3)	a
a	4	4	3	(u,1)	a

O(😢)

Ukkonen’s algorithm

don’t!

Application of suffix trees

pretty much the same as tries

- Word search and generation
- String matching
- Longest repeated sequence
- **longest common substring**
- Data analytics

UVa 10679

Hmmmmmm..... strings again :) Then it must be an easy task for you. Well . . . you are given with a string S of length not more than 100,000 (only ‘a’-‘z’ and ‘A’-‘Z’). Then follows q (q < 1000) queries where each query contains a string T of maximum length 1,000 (also contains only ‘a’-‘z’ and ‘A’-‘Z’). You have to determine whether or not T is a sub-string of S.

UVa 10679

Input

First line contains an integer k (k < 10) telling the number of test cases to follow. Each test case begins with S. It is followed by q. After this line there are q lines each of which has a string T as defined before.

Output

For each query print ‘y’ if it is a sub-string of S or ‘n’ otherwise followed by a new line. See the sample output below.

SUFFIX ARRAYS

Suffix arrays

Definition A suffix array for a string S is a lexicographical order of its suffixes

Suffix arrays are an easier way to represent suffix trees

Suffix arrays are integer permutations of the leafs in a suffix tree (the permutation corresponds to the tree walk)

S = ABAACBAB

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

Prefix doubling

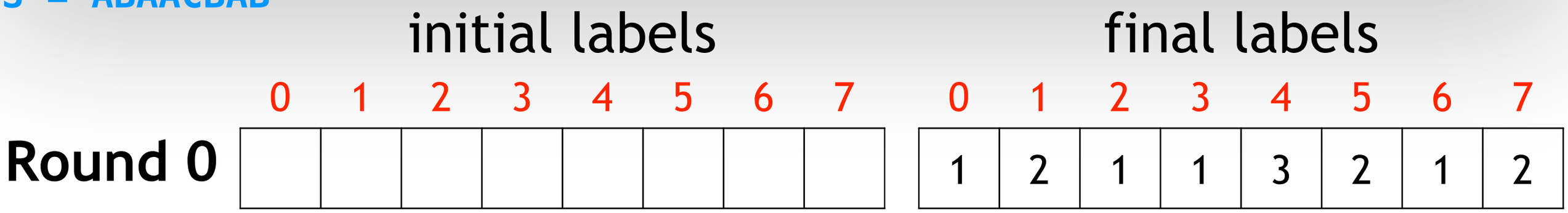
The algorithm consists of a series of *rounds* iterating over substrings of size 2^i

Round 0. Give a label to each character in increasing order

Round i. divide x (of length 2^i) into two halves a and b of length 2^{i-1}. Give x the label `(l(a), l(b))`, and sort them. Then, give the strings a **new** label in ascending order 1, 2, …

Prefix doubling

S = ABAACBAB



Prefix doubling

S = ABAACBAB

initial labels

0	1	2	3	4	5	6	7
Round 0								

final labels

0	1	2	3	4	5	6	7
1	2	1	1	3	2	1	2

Round 1 {AB, BA, AA, AC, CB, BA, AB}

0	1	2	3	4	5	6	7
1,2	2,1	1,1	1,3	3,2	2,1	1,2	2,0

Prefix doubling

S = ABAACBAB

initial labels

0	1	2	3	4	5	6	7

Round 0

final labels

0	1	2	3	4	5	6	7
1	2	1	1	3	2	1	2

Round 1 {AB, BA, AA, AC, CB, BA, AB}

0	1	2	3	4	5	6	7
1,2	2,1	1,1	1,3	3,2	2,1	1,2	2,0

0	1	2	3	4	5	6	7
1,1	1,2	1,2	1,3	2,0	2,1	2,1	3,2

Prefix doubling

S = ABAACBAB

initial labels

0	1	2	3	4	5	6	7

Round 0

final labels

0	1	2	3	4	5	6	7
1	2	1	1	3	2	1	2

Round 1 {AB, BA, AA, AC, CB, BA, AB}

0	1	2	3	4	5	6	7
1,2	2,1	1,1	1,3	3,2	2,1	1,2	2,0

0	1	2	3	4	5	6	7
2	5	1	3	6	5	2	4

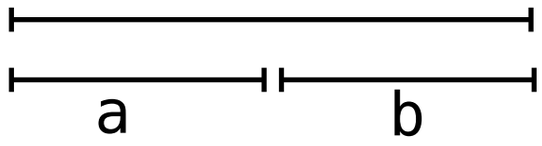
0	1	2	3	4	5	6	7
1,1	1,2	1,2	1,3	2,0	2,1	2,1	3,2

Prefix doubling

S = ABAACBAB

initial labels

final labels



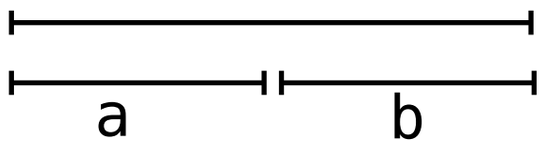
Prefix doubling

S = ABAACBAB

initial labels

final labels

Round 2



0	1	2	3	4	5	6	7
2,1	5,3	1,6	3,5	6,2	5,4	2,0	4,0

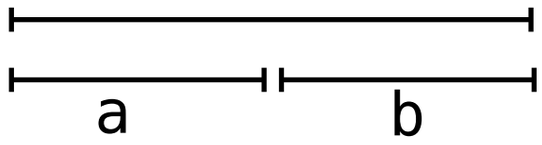
Prefix doubling

S = ABAACBAB

initial labels

final labels

Round 2



0	1	2	3	4	5	6	7
2,1	5,3	1,6	3,5	6,2	5,4	2,0	4,0

0	1	2	3	4	5	6	7
3	6	1	4	8	7	2	5

Prefix doubling

S = ABAACBAB

initial labels

final labels

Round 2

0	1	2	3	4	5	6	7
2	5	1	3	6	5	2	4

a b

0	1	2	3	4	5	6	7
2,1	5,3	1,6	3,5	6,2	5,4	2,0	4,0

0	1	2	3	4	5	6	7
3	6	1	4	8	7	2	5

Round 3

0	1	2	3	4	5	6	7
3,8	6,7	1,2	4,5	8,0	7,0	2,0	5,0

Prefix doubling

S = ABAACBAB

initial labels

final labels

Round 2

0	1	2	3	4	5	6	7
2	5	1	3	6	5	2	4

a b

0	1	2	3	4	5	6	7
2,1	5,3	1,6	3,5	6,2	5,4	2,0	4,0

0	1	2	3	4	5	6	7
3	6	1	4	8	7	2	5

Round 3

0	1	2	3	4	5	6	7
3,8	6,7	1,2	4,5	8,0	7,0	2,0	5,0

0	1	2	3	4	5	6	7
3	6	1	4	8	7	2	5

Prefix doubling

S = ABAACBAB

initial labels

final labels

Round 2

0	1	2	3	4	5	6	7
2	5	1	3	6	5	2	4

a b

0	1	2	3	4	5	6	7
2,1	5,3	1,6	3,5	6,2	5,4	2,0	4,0

0	1	2	3	4	5	6	7
3	6	1	4	8	7	2	5

Round 3

0	1	2	3	4	5	6	7
3,8	6,7	1,2	4,5	8,0	7,0	2,0	5,0

0	1	2	3	4	5	6	7
3	6	1	4	8	7	2	5

0	1	2	3	4	5	6	7
2	**5**	**0**	**3**	**7**	**6**	**1**	**4**

Prefix doubling

S = ABAACBAB

find the occurrences of BA

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

Prefix doubling

S = ABAACBAB

find the occurrences of BA

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	**B**
5	1	**B**AACBAB
6	5	**B**AB
7	4	CBAB

Prefix doubling

S = ABAACBAB

find the occurrences of BA

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	**B**
5	1	**B**AACBAB
6	5	**B**AB
7	4	CBAB

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	**BA**ACBAB
6	5	**BA**B
7	4	CBAB

Prefix doubling

```
void buildSA() {
    N = strlen(S);
    REP(i, N) sa[i] = i, pos[i] = S[i];
    for (gap = 1;; gap *= 2) {
        sort(sa, sa + N, sufCmp);
        REP(i, N - 1) tmp[i + 1] = tmp[i] + sufCmp(sa[i], sa[i + 1]);
        REP(i, N) pos[sa[i]] = tmp[i];
        if (tmp[N - 1] == N - 1) break;
    }
}

void buildLCP() {
    for (int i = 0, k = 0; i < N; ++i) if (pos[i] != N - 1) {
        for (int j = sa[pos[i] + 1]; S[i + k] == S[j + k];)
           ++k;
        lcp[pos[i]] = k;
        if (k)--k;
    }
}
```

Prefix doubling

```
const int MAXN = 1 << 21;
char * S;
int N, gap;
int sa[MAXN], pos[MAXN], tmp[MAXN], lcp[MAXN];

bool sufCmp(int i, int j) {
    if (pos[i] != pos[j])
        return pos[i] < pos[j];
    i += gap;
    j += gap;
    return (i < N && j < N) ? pos[i] < pos[j] : i > j;
}
```

Application of suffix arrays

yes, just suffix trees

- Word search and generation
- String matching
- Longest repeated sequence
- **longest common substring**
- Data analytics

https://www.spoj.com/problems/SUBLEX/

Little Daniel loves to play with strings! He always finds different ways to have fun with strings! Knowing that, his friend Kinan decided to test his skills so he gave him a string S and asked him Q questions of the form:

If all distinct substrings of string S were sorted lexicographically, which one will be the K-th smallest?

After knowing the huge number of questions Kinan will ask, Daniel figured out that he can't do this alone. Daniel, of course, knows your exceptional programming skills, so he asked you to write him a program which given S will answer Kinan's questions.

https://www.spoj.com/problems/SUBLEX/

Input
In the first line there is Kinan's string S (with length no more than 90000 characters). It contains only small letters of English alphabet. The second line contains a single integer Q (Q <= 500) , the number of questions Daniel will be asked. In the next Q lines a single integer K is given (0 < K < 2^31).

Output
Output consists of Q lines, the i-th contains a string which is the answer to the i-th asked question.

https://www.spoj.com/problems/SUBLEX/

Example
Input:
aaa
2
2
3

Output:
aa
aaa

S = "aaa" (without quotes)
substrings of S are "a" , "a" , "a" , "aa" , "aa" , "aaa". The sorted list of substrings will be:
"a", "aa", "aaa".

https://www.spoj.com/problems/SUBLEX/

Code in git