Imperial College
London

CO202 – Software Engineering – Algorithms

Radix Search

Ben Glocker
Huxley Building, Room 377
b.glocker@imperial.ac.uk

The material is partly based on previous lectures by Prof Alex Wolf

Searching

- Recall how we improved on the time complexity of the brute-force algorithm `NAIVE-STRING-MATCHER`

  we added a **preprocessing** step that is applied to the **search pattern**, while the text (i.e., the database) is left as is

  different types of preprocessing have led to different algorithms (`KMP-MATCHER`, `BM-MATCHER`)

- Recall binary search trees

  a BST is the result of a **preprocessing** step applied to a **list of keys** (i.e., the database), while the search pattern (i.e., the search key) is left as is

  different types of preprocessing have led to different trees (e.g. Randomised BST)

Recall Binary Search

```
BST-SEARCH(t,k)
 1: if t == NIL
 2:     return NIL
 3: if k == t.key
 4:     return t
 5: if k < t.key
 6:     return BST-SEARCH(t.left,k)
 7: else
 8:     return BST-SEARCH(t.right,k)
```

"Ultimate" D&C Algorithm

Running time: $O(\log n)$

Can we ignore the cost of key comparison?

What if keys are long strings of characters? Song titles, URLs, …

String Comparison in Binary Search

Assuming a string is an array of characters, the comparison `k == t.key` can be implemented as

```
STRING-EQUALS(S1,S2)
 1: m = S1.length
 2: if S2.length ≠ m
 3:     return false
 4: for i = 1 to m
 5:     if S1[i] ≠ S2[i]
 6:         return false
 7: return true
```

The running time of `STRING-EQUALS` is $O(m)$, so, the running time of `BST-SEARCH` on strings is actually $O(m \log n)$.

Some Observations

- When we start `BST-SEARCH(t,k)`
  - `t.key` is probably quite different from `k`
  - so, `STRING-EQUALS` is likely to return quickly

- But as we progress down the tree in `BST-SEARCH(t,k)`
  - `t.key` becomes more and more similar to `k`
  - so, `STRING-EQUALS` is likely to return later and later
  - in fact, `STRING-EQUALS` is likely to compare the same **prefix** of `k` many times

Radix Search

Radix search proceeds by examining keys **one small piece at a time**, rather than doing full comparison at each step

Motivation

- reasonable worst-case performance without complications of balanced trees
- good way to handle long keys, where key comparison is a significant overhead
- fast access to database, in fact competitive to hashing

Google uses radix search to support “suggestion” completion.

Radix Keys

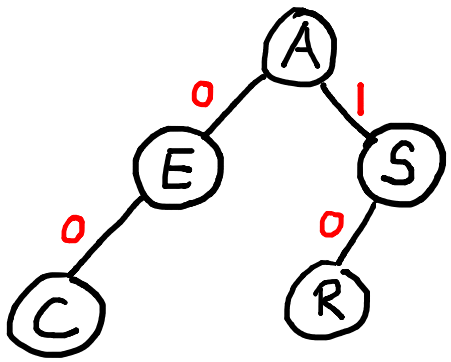
- A key is typically an **integer**, a **character**, a **word** (a fixed number of characters), or a **string** (a variable number of characters)
- Radix search algorithms treat keys as numbers represented in a **base-R** number system, for various values of R (the radix)

  Different values of R are appropriate for different applications
- We access individual **digits** of those numbers using the abstract operation `DIGIT(k,d)` which retrieves the d-th digit of key k
- We use the name **bit** (binary digit) when $R = 2$

Starting point: Digital Search Trees

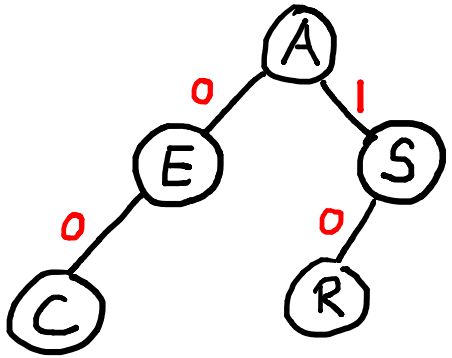
- The simplest radix search method is based on DSTs
- Search and insert algorithms are identical to binary search trees except for one difference
  - Left/right branching is not based on comparison between full keys, but rather according to selected bits of the keys
  - First level uses the leading bit, second level the second leading bit, and so on, until a leaf (or `NIL`) is encountered

A	00001
S	10011
E	00101
R	10010
C	00011



Digital Search Trees

```
DST-SEARCH(t,k,d)
 1: if t == NIL
 2:     return NIL
 3: if k == t.key
 4:     return t
 5: if DIGIT(k,d) == 0
 6:     return DST-SEARCH(t.left,k,d+1)
 7: else
 8:     return DST-SEARCH(t.right,k,d+1)
```



Still, whole keys are compared

Digital Search Trees

Analysis

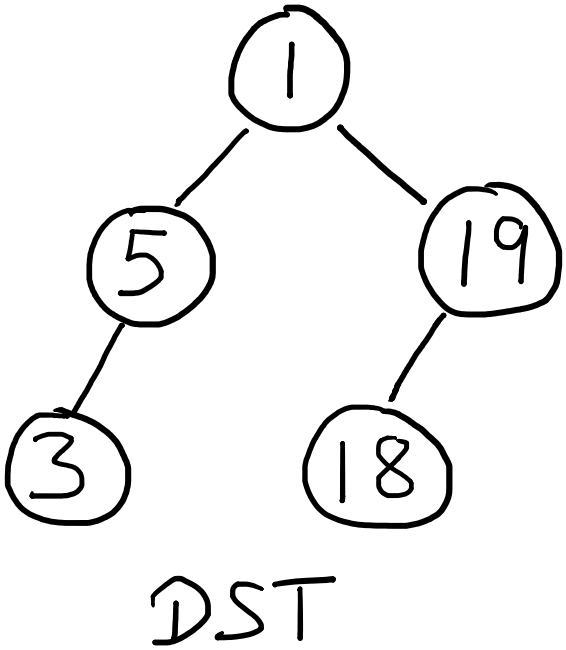
- Maximum depth of a DST is the maximum key length b, independent of key values
- Search/insert requires $\lg n$ key comparisons on average, and b key comparisons in the worst-case for a tree built from n random b-bit keys

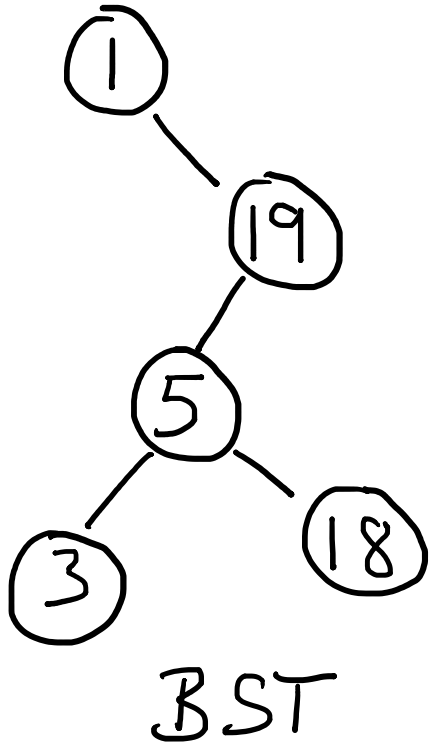
  results in nearly balanced trees; intuitively, the “next” bit of a random key is equally likely to be 0 or 1
- Keys with **similar prefixes degrade performance**

  e.g., suppose all keys started with 0, then right branch of root would be empty

DST vs BST

DSTs do not have the ordering property that characterizes BSTs





A	1	00001
S	19	10011
E	5	00101
R	18	10010
C	3	00011

Binary Search Tries

- A **trie** (from re*trie*val, but pronounced “try”) is a tree where keys are stored only in the **leaf nodes**
- As with digital search trees, branch on digit *j* at level *j*
- Unlike DSTs, **don’t compare the full key** until a leaf is reached

  solves the expensive `STRING-EQUALS` problem
- The shape of the trie is independent of the order of insert of keys

  there is a unique trie for any given set of distinct keys

Binary Search Tries

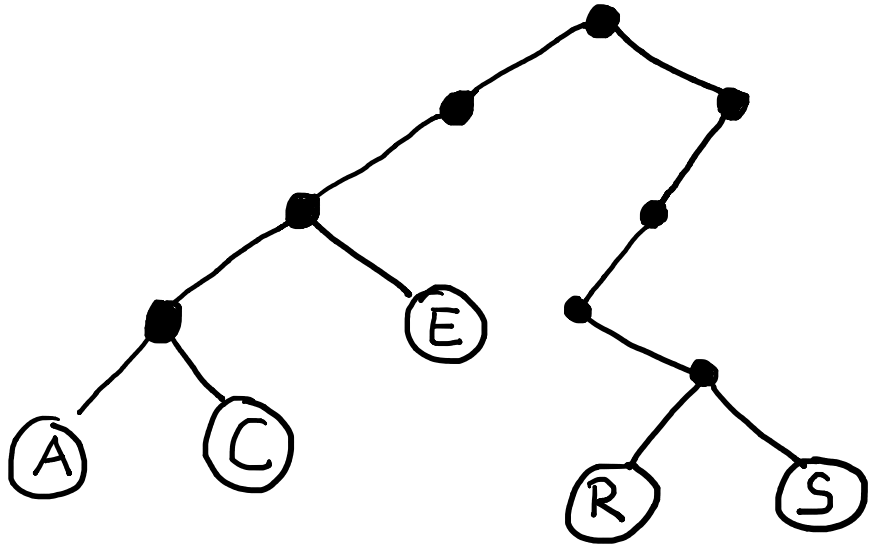
Definition

A **binary search trie** is a binary tree that has keys associated with each of its leaves, defined recursively as follows:

- the trie for an empty set of keys is a `NIL` link;
- the trie for a single key is a leaf containing that key;
- and the trie for a set of keys of cardinality greater than one is an internal node with left link referring to the trie for the keys whose initial bit is 0 and right link referring to the trie for the keys whose initial bit is 1, with the leading bit considered to be removed for the purpose of constructing the subtrees.

Binary Search Tries

- Radix of $R = 2$, so we consider 1 bit at a time
- Two kinds of nodes
  - leaf nodes: contain full keys
  - internal nodes: acting as “sign posts”



A	00001
S	10011
E	00101
R	10010
C	00011

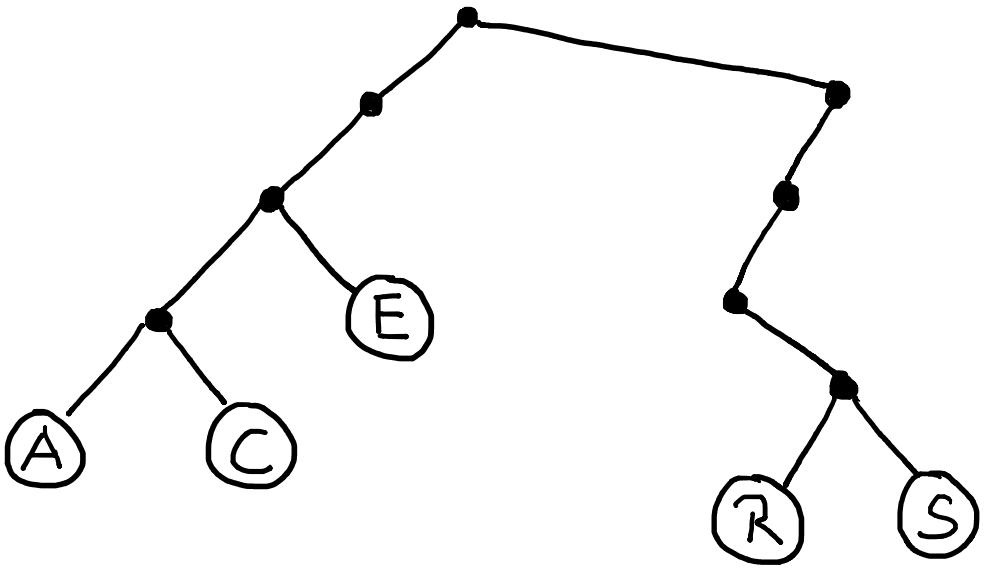
Binary Search Tries

```
BSTRIE-SEARCH(t,k,d)
 1: if t == NIL
 2:     return NIL
 3: if t.left == NIL and t.right == NIL
 4:     if k == t.key
 5:         return t
 6:     else
 7:         return NIL
 8: if DIGIT(k,d) == 0
 9:     return BSTRIE-SEARCH(t.left,k,d+1)
10: else
11:     return BSTRIE-SEARCH(t.right,k,d+1)
```

Whole key comparison only at leaf nodes

Binary Search Tries

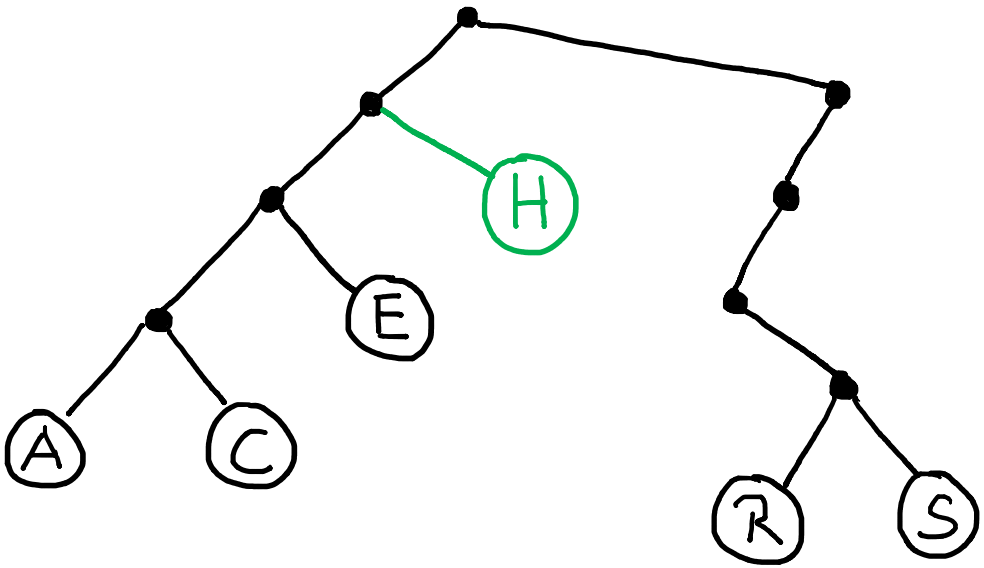
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011

Binary Search Tries

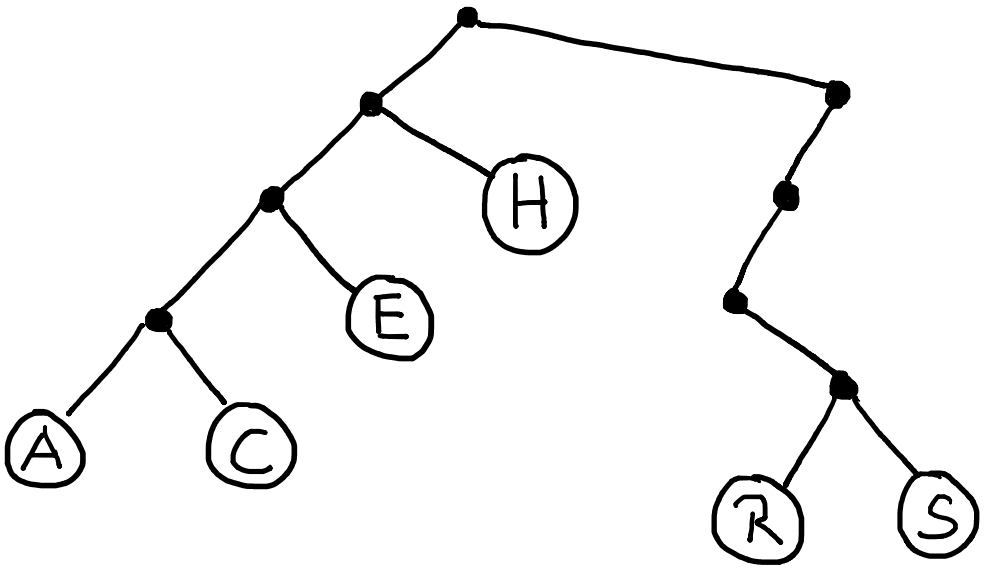
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000

Binary Search Tries

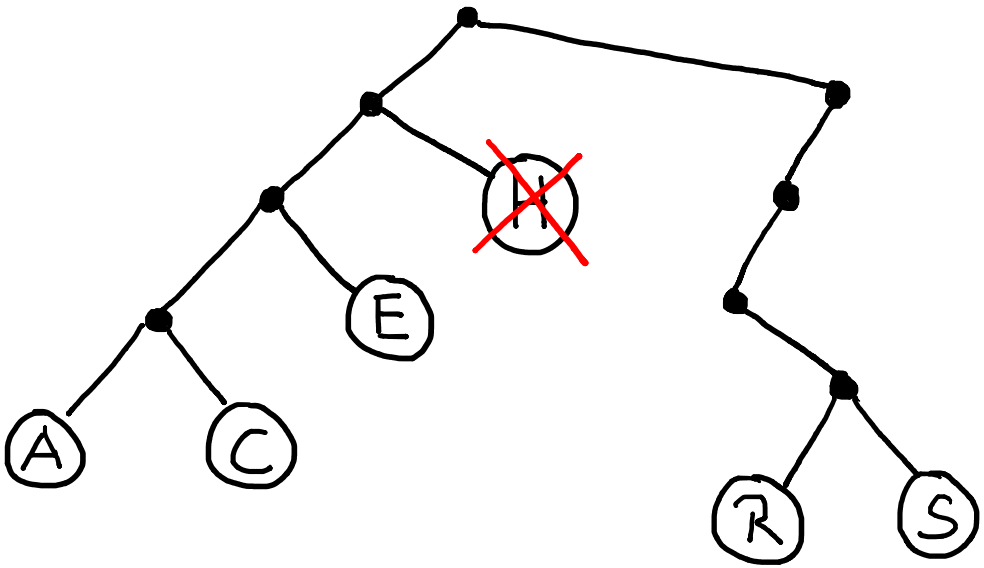
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000

Binary Search Tries

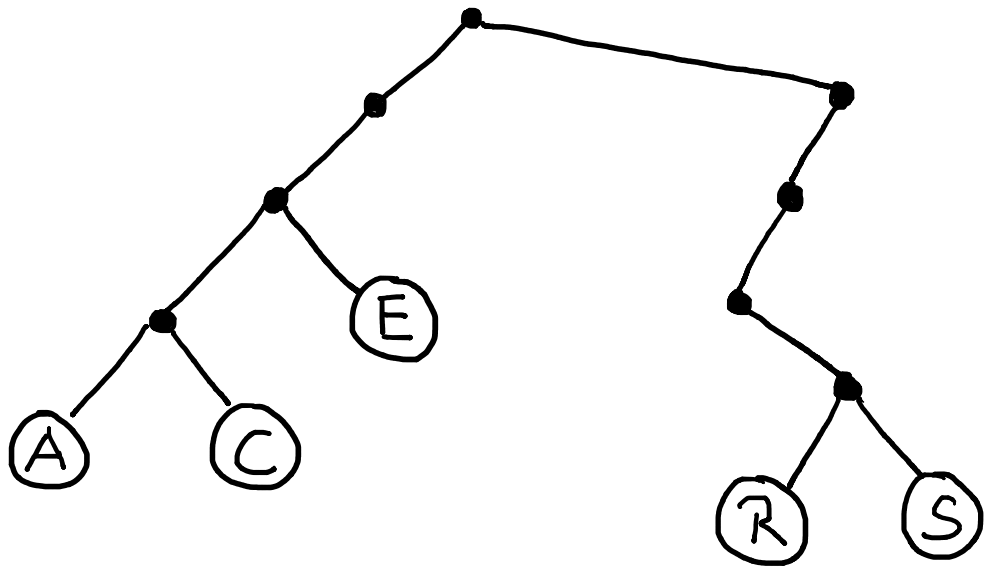
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

Binary Search Tries

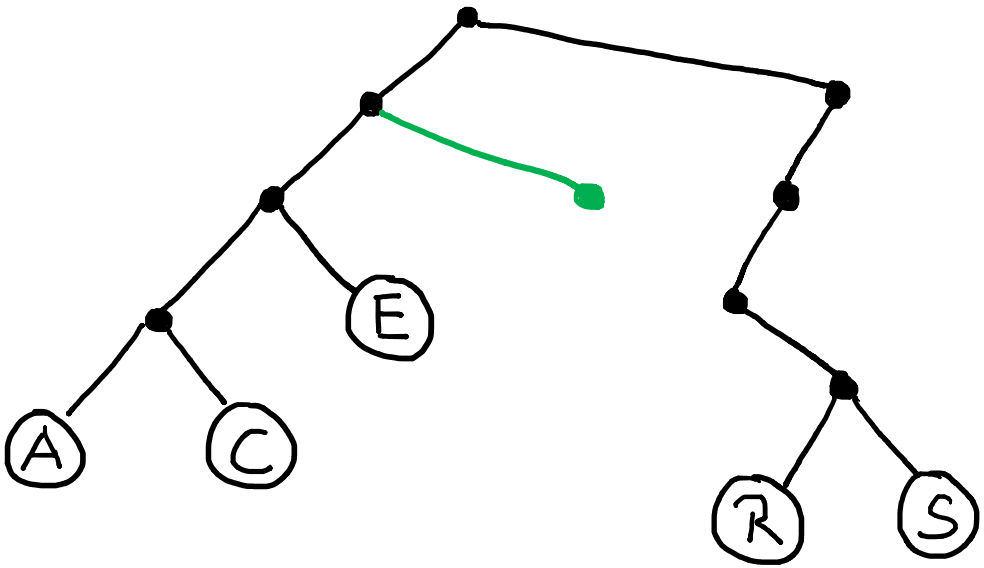
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

Binary Search Tries

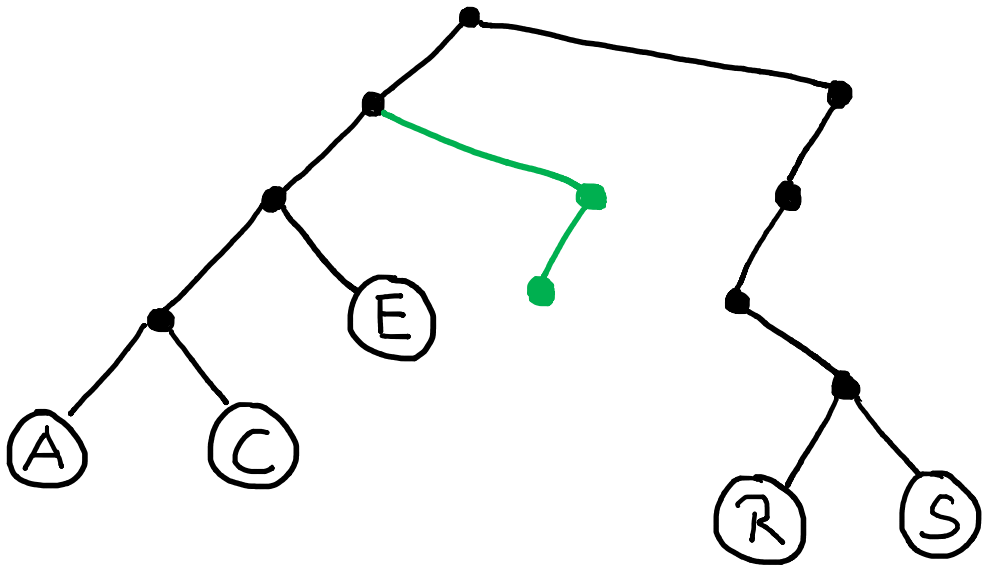
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

Binary Search Tries

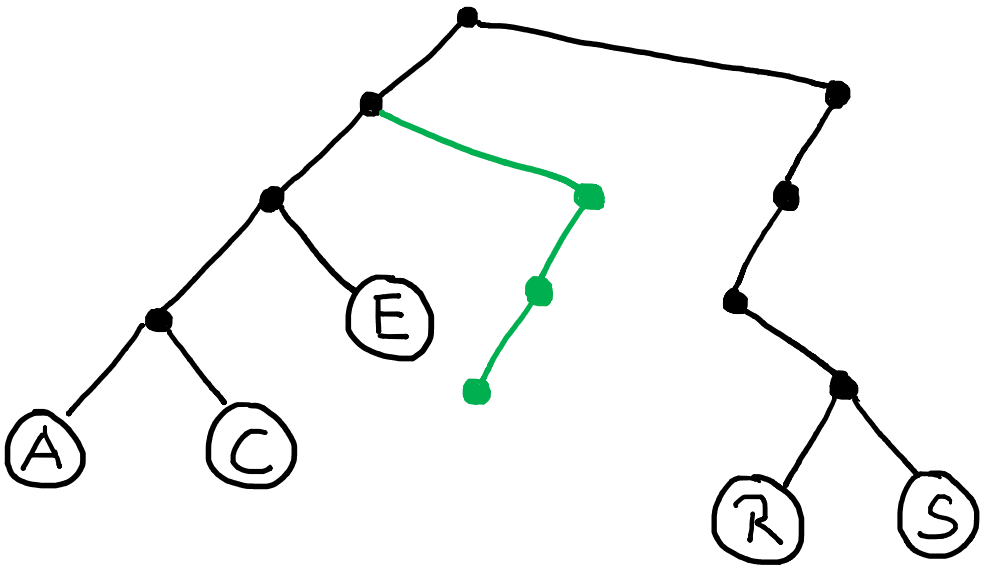
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

Binary Search Tries

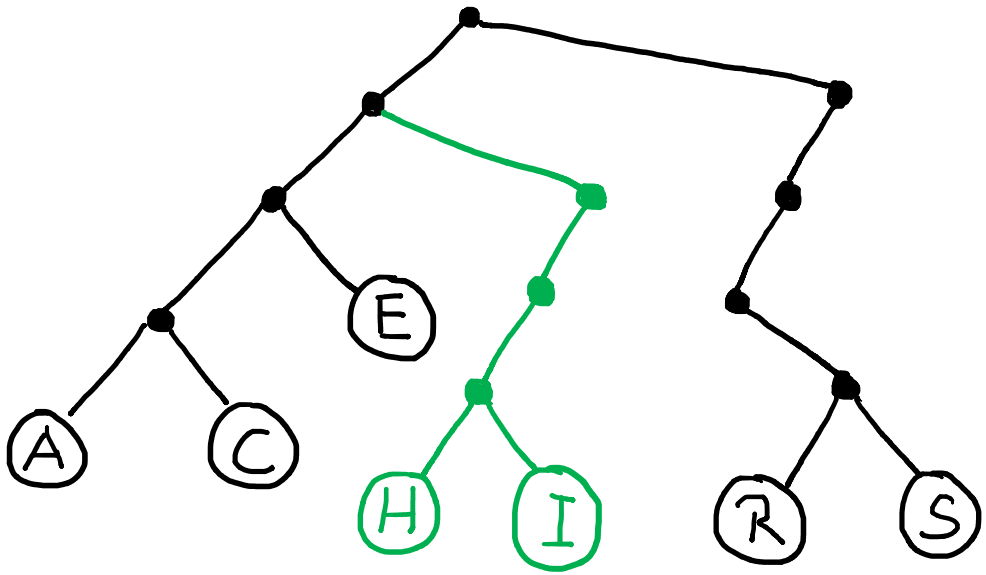
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

Binary Search Tries

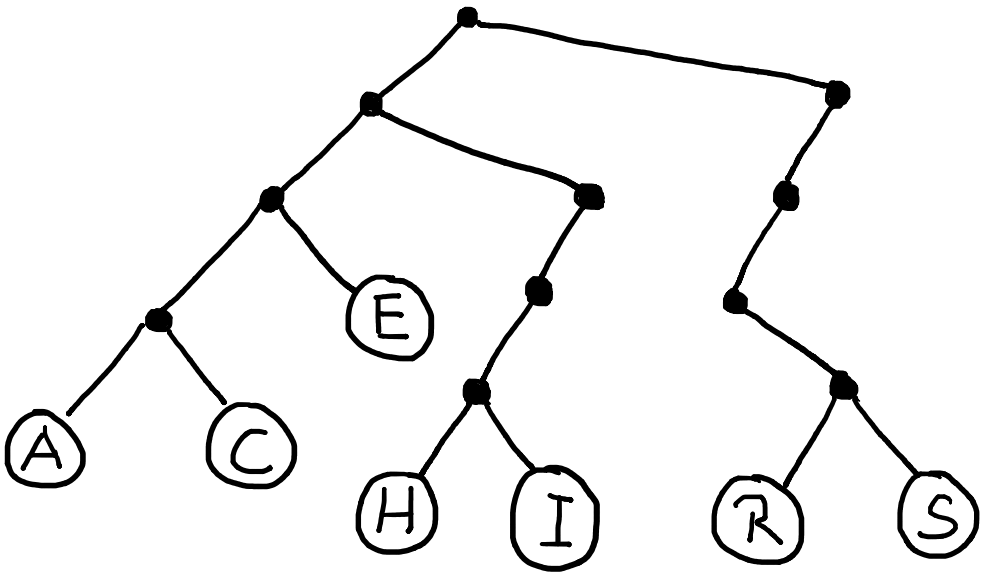
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

Binary Search Tries

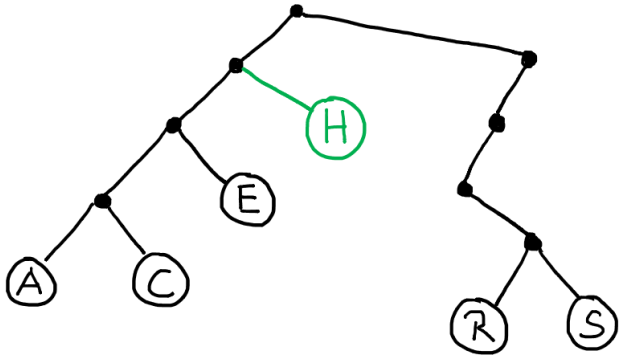
Insert algorithm, however, is more complex



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001

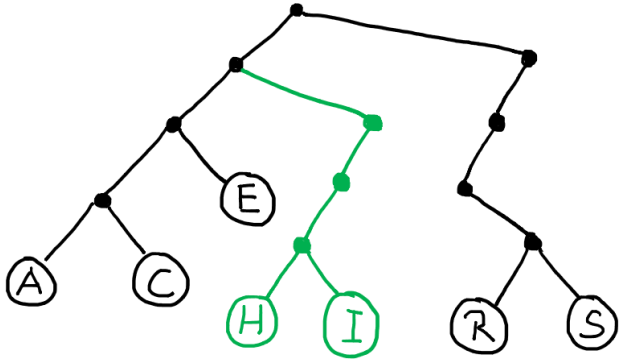
Binary Search Tries

```
BSTRIE-INSERT(t,z,d)
1: if t == NIL
2:     return z
3: if t.left == NIL and t.right == NIL
4:     return BSTRIE-SPLIT(z,t,d)
5: if DIGIT(z.key,d) == 0
6:     t.left = BSTRIE-INSERT(t.left,z,d+1)
7: else
8:     t.right = BSTRIE-INSERT(t.right,z,d+1)
9: return t
```



Binary Search Tries

```
BSTRIE-INSERT(t,z,d)
 1: if t == NIL
 2:     return z
 3: if t.left == NIL and t.right == NIL
 4:     return BSTRIE-SPLIT(z,t,d)
 5: if DIGIT(z.key,d) == 0
 6:     t.left = BSTRIE-INSERT(t.left,z,d+1)
 7: else
 8:     t.right = BSTRIE-INSERT(t.right,z,d+1)
 9: return t
```



Binary Search Tries

```
BSTRIE-SPLIT(p,q,d)
 1: t = NEWNODE()
 2: switch DIGIT(p.key,d)*2 + DIGIT(q.key,d)
 3:     case 0:
 4:         t.left = BSTRIE-SPLIT(p,q,d+1)
 5:     case 1:
 6:         t.left = p
 7:         t.right = q
 8:     case 2:
 9:         t.right = p
10:         t.left = q
11:     case 3:
12:         t.right = BSTRIE-SPLIT(p,q,d+1)
13: return t
```

Binary Search Tries

Analysis

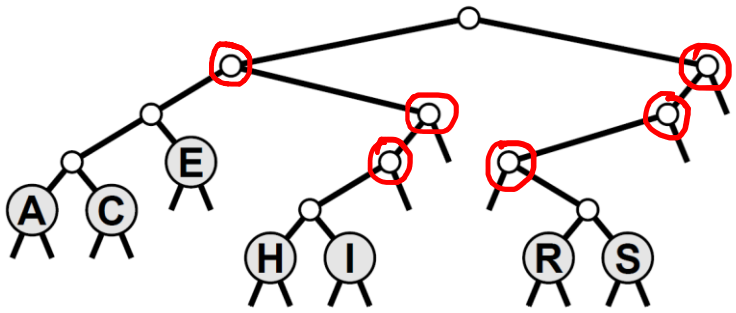
- Search/insert requires about $\lg n$ bit comparisons on average and b bit comparisons in the worst-case in a trie built from n random b-bit keys
- A trie built from n random b-bit keys has about $n / \ln 2 \approx 1.44n$ nodes on average
- Keys that differ only in the last bit have a path equal to the key length no matter how many nodes are in the trie

Patricia Tries

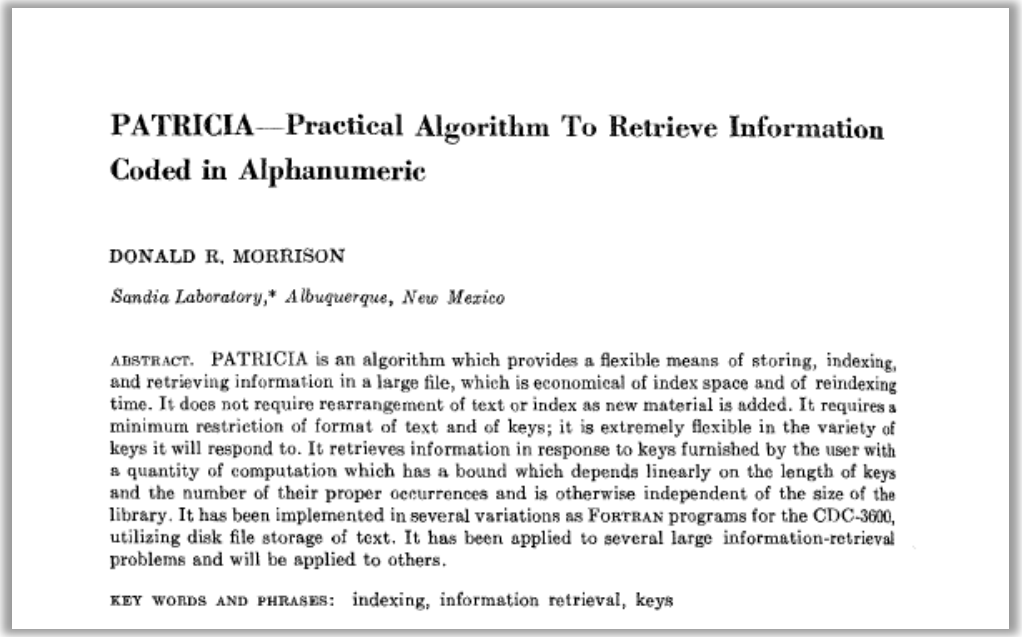
Trie-based search has two inconvenient flaws

First, the one-way branching leads to the **creation of extra nodes** in the trie, which seem unnecessary

Second, there are **two different types of nodes** in the trie, which leads to complications in implementations



[Sedgewick] p.642



PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric

DONALD R. MORRISON

Sandia Laboratory, Albuquerque, New Mexico*

ABSTRACT. PATRICIA is an algorithm which provides a flexible means of storing, indexing, and retrieving information in a large file, which is economical of index space and of reindexing time. It does not require rearrangement of text or index as new material is added. It requires a minimum restriction of format of text and of keys; it is extremely flexible in the variety of keys it will respond to. It retrieves information in response to keys furnished by the user with a quantity of computation which has a bound which depends linearly on the length of keys and the number of their proper occurrences and is otherwise independent of the size of the library. It has been implemented in several variations as FORTRAN programs for the CDC-3600, utilizing disk file storage of text. It has been applied to several large information-retrieval problems and will be applied to others.

KEY WORDS AND PHRASES: indexing, information retrieval, keys

Journal of the ACM (JACM), Volume 15, Issue 4, Oct. 1968

Patricia Tries

Like DSTs, Patricia tries allow search for n keys in a tree with just n nodes

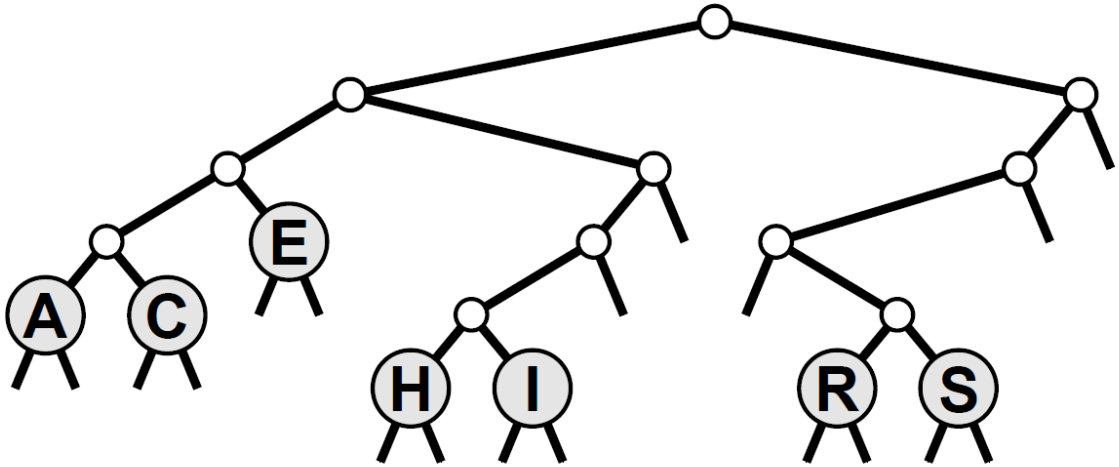
Like BSTries, they require only about $\lg n$ bit comparisons and one full key comparison per search

Building Patricia tries:

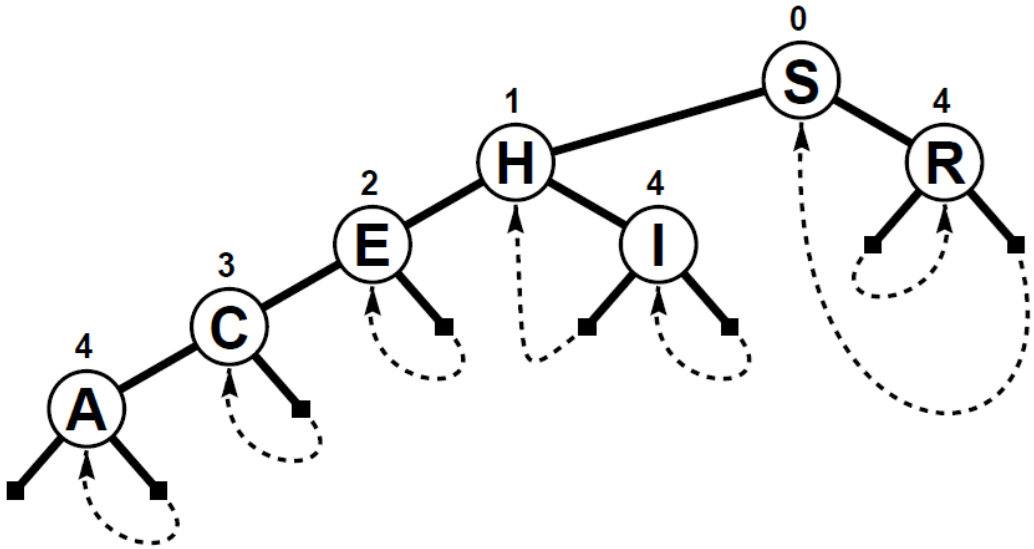
1) We put into each node the index of the bit to be tested, accessible via `t.bit`
2) We store data in every node, and replace `NIL` links with links that point back upwards into the trie

Patricia Tries vs BSTrie

BSTrie



Patricia Trie



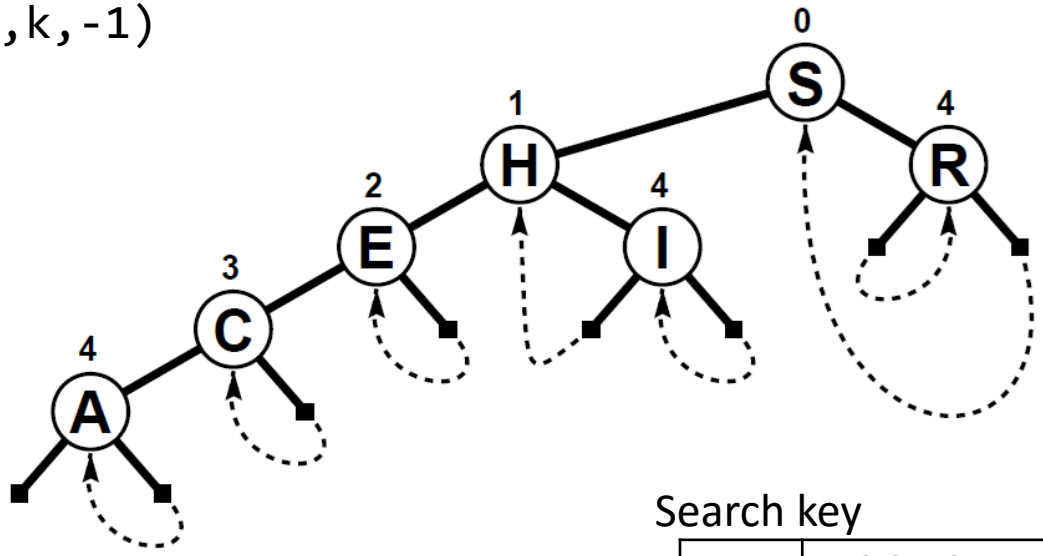
[Sedgewick] p.653

Patricia Tries Search

```
PATRICIA-SEARCH(t,k)
 1: t = PATRICIA-FIND-NODE(t,k,-1)
 2: if t ≠ NIL
 3:     if k == t.key
 4:         return t
 5: return NIL

PATRICIA-FIND-NODE(t,k,i)
 1: if t == NIL
 2:     return NIL
 3: if t.bit ≤ i
 4:     return t
 5: if DIGIT(k,t.bit) == 0
 6:     return PATRICIA-FIND-NODE(t.left,k,t.bit)
 7: else
 8:     return PATRICIA-FIND-NODE(t.right,k,t.bit)
```

Patricia Tries Search

```
PATRICIA-SEARCH(t,k)
 1: t = PATRICIA-FIND-NODE(t,k,-1)
 2: if t ≠ NIL
 3:     if k == t.key
 4:         return t
 5: return NIL

PATRICIA-FIND-NODE(t,k,i)
 1: if t == NIL
 2:     return NIL
 3: if t.bit ≤ i
 4:     return t
 5: if DIGIT(k,t.bit) == 0
 6:     return PATRICIA-FIND-NODE(t.left,k,t.bit)
 7: else
 8:     return PATRICIA-FIND-NODE(t.right,k,t.bit)
```

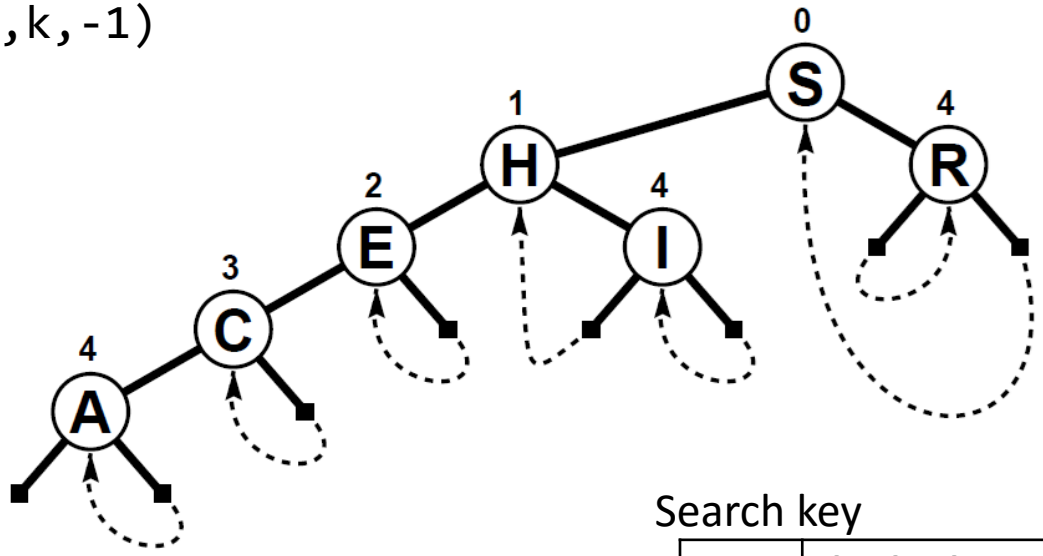


Search key

R	10010

Patricia Tries Search

```
PATRICIA-SEARCH(t,k)
 1: t = PATRICIA-FIND-NODE(t,k,-1)
 2: if t ≠ NIL
 3:     if k == t.key
 4:         return t
 5: return NIL

PATRICIA-FIND-NODE(t,k,i)
 1: if t == NIL
 2:     return NIL
 3: if t.bit ≤ i
 4:     return t
 5: if DIGIT(k,t.bit) == 0
 6:     return PATRICIA-FIND-NODE(t.left,k,t.bit)
 7: else
 8:     return PATRICIA-FIND-NODE(t.right,k,t.bit)
```



Search key

J	01010

Patricia Tries Insertion

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t
```

For `NIL` keys, we reserve the representation of all digits to be zeros.

Patricia Tries Insertion

```
PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```

Patricia Tries

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

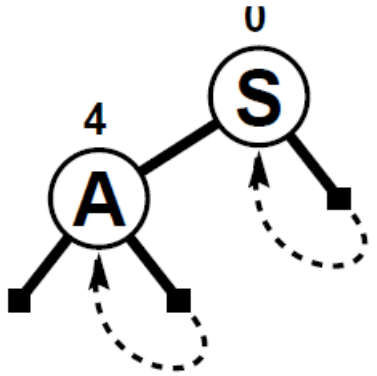
Patricia Tries

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries

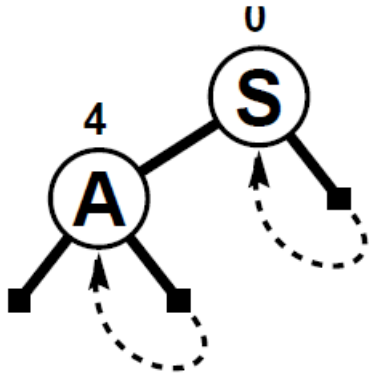
```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```

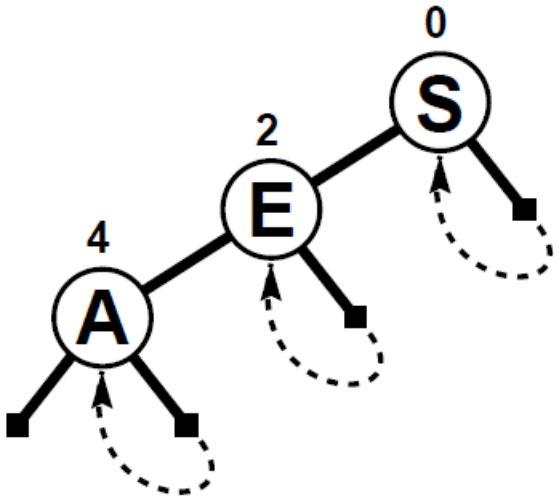


A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t
```

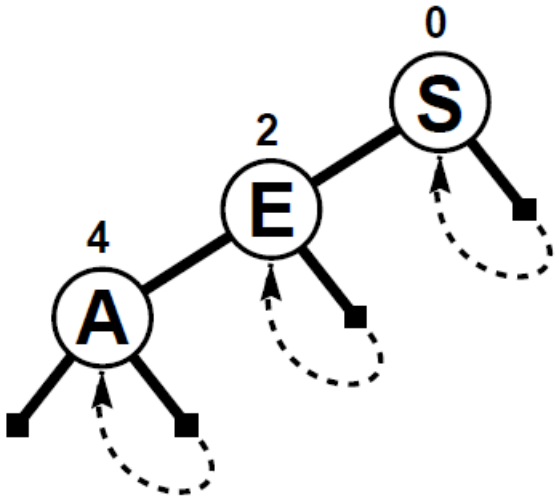
```
PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries

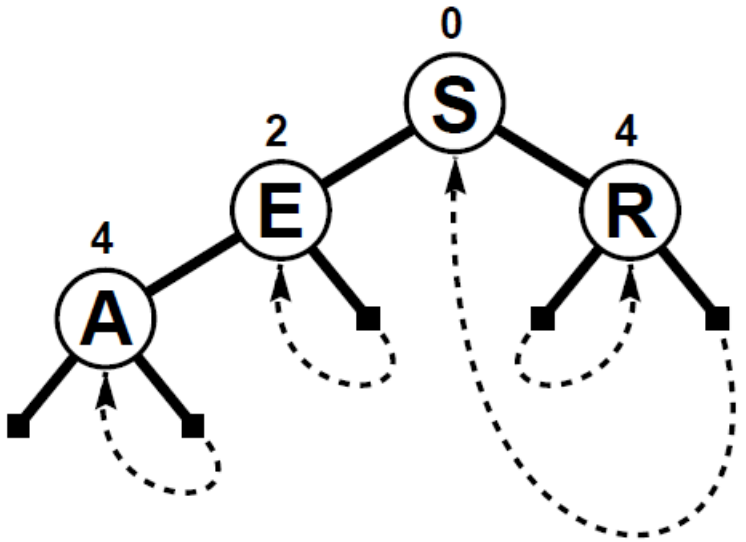
```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

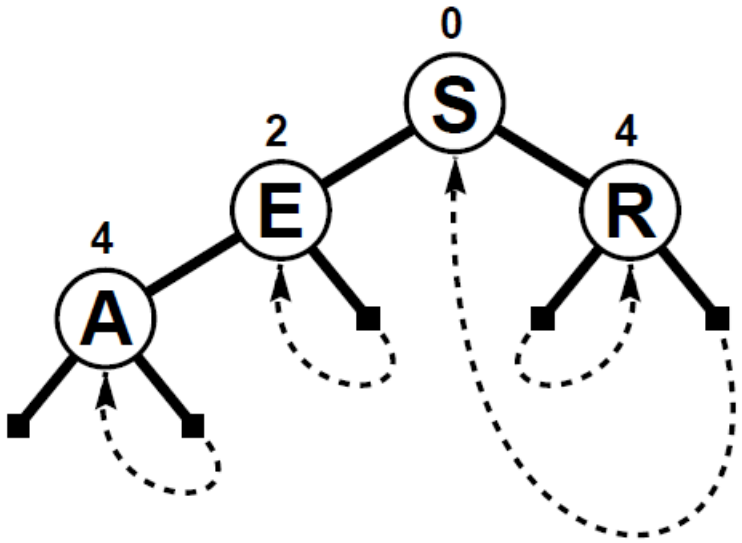
Patricia Tries

```
PATRICIA-INSERT(t,z,d)
 1: h = PATRICIA-FIND-NODE(t,z.key,-1)
 2: w = (h == NIL) ? NIL : h.key
 3: d = 0
 4: while DIGIT(z.key,d) == DIGIT(w,d)
 5:     d = d+1
 6: t = PATRICIA-INSERT-NODE(t,z,d,-1)
 7: return t

PATRICIA-INSERT-NODE(t,z,d,q)
 1: if t == NIL or t.bit ≥ d or t.bit ≤ q
 2:     z.bit = d
 3:     if DIGIT(z.key,d) == 0
 4:         z.left = z
 5:         z.right = t
 6:     else
 7:         z.left = t
 8:         z.right = z
 9:     return z
10: if DIGIT(z.key,t.bit) == 0
11:     t.left = PATRICIA-INSERT-NODE(t.left,z,d,t.bit)
12: else
13:     t.right = PATRICIA-INSERT-NODE(t.right,z,d,t.bit)
14: return t
```



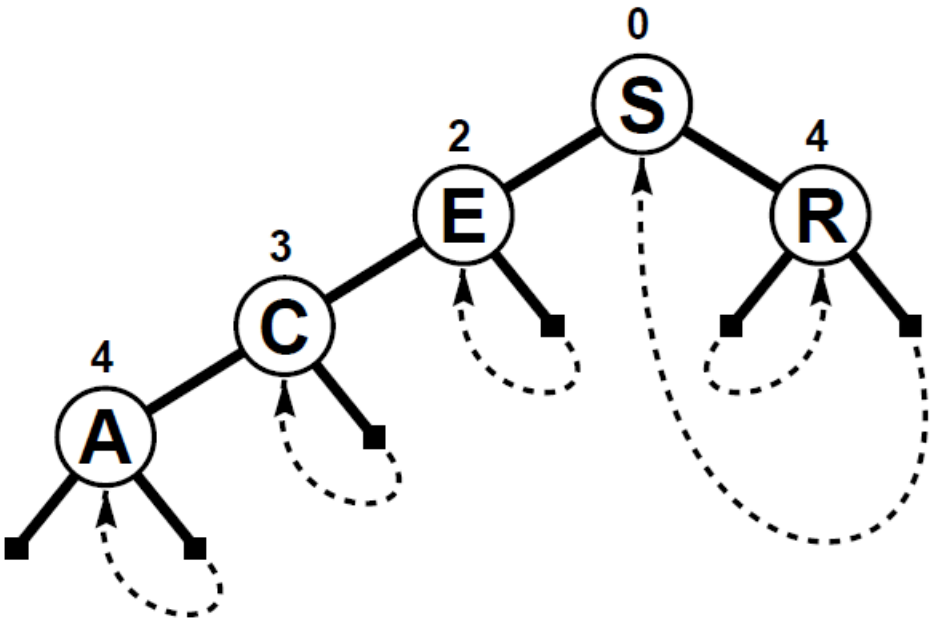
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries



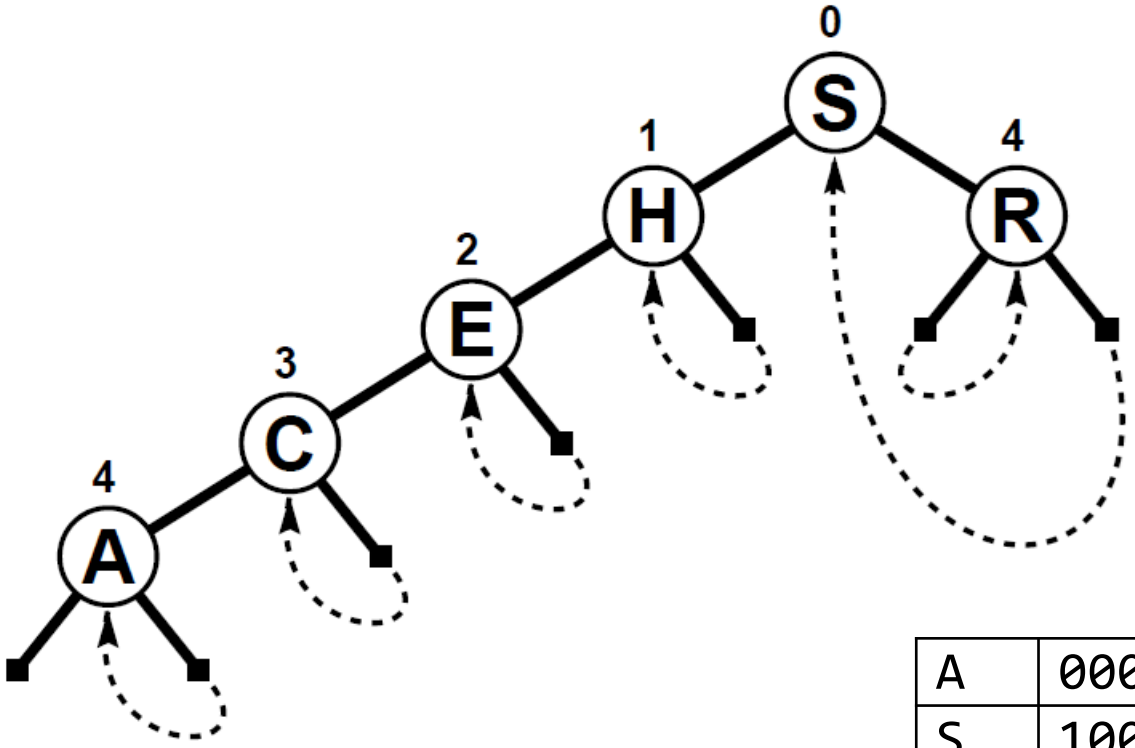
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries



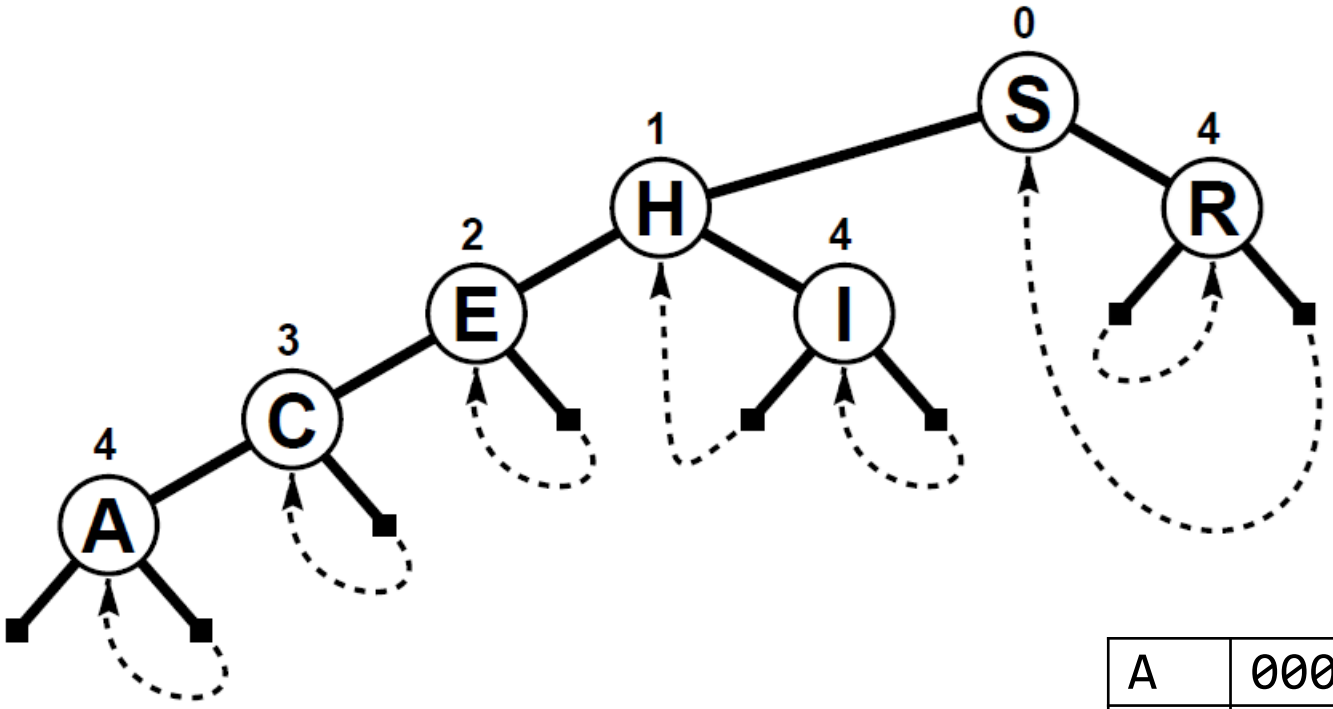
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries



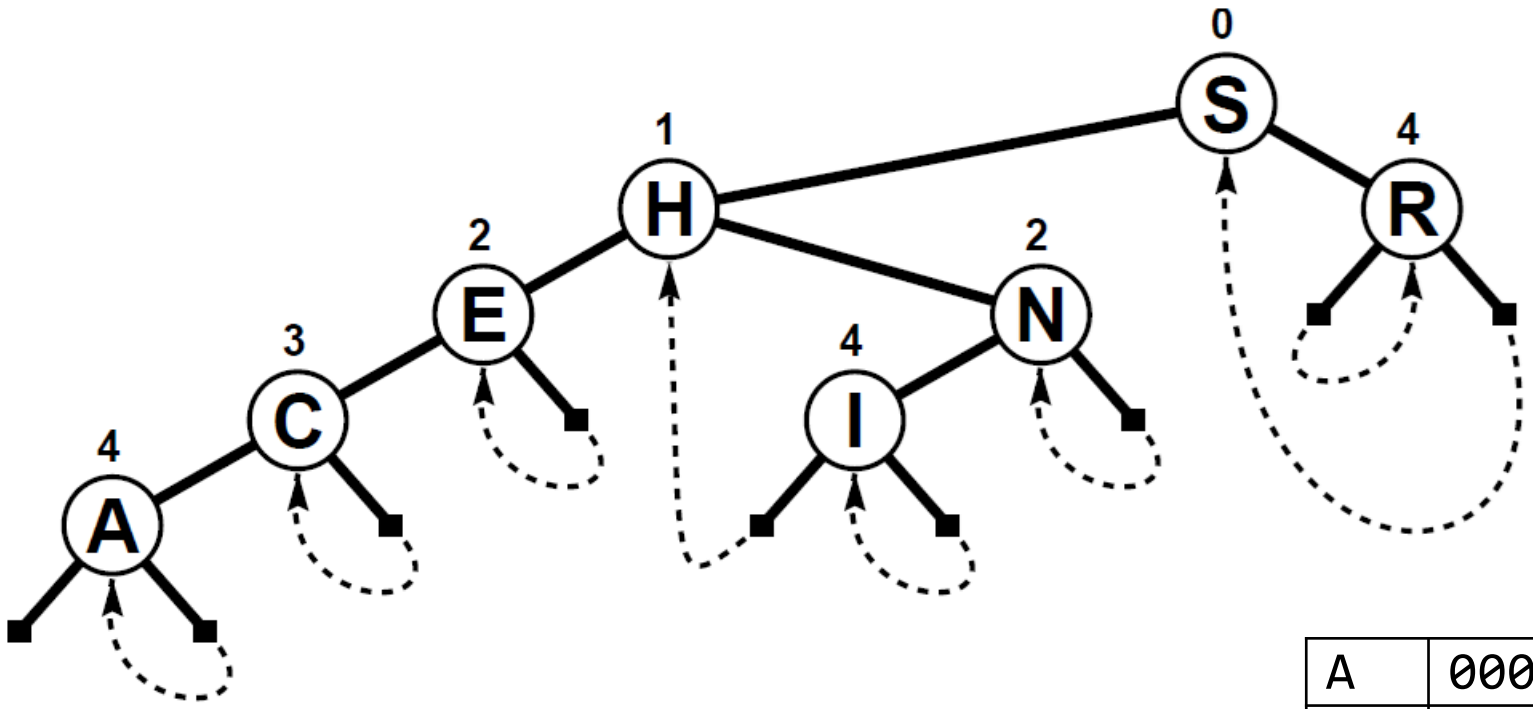
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries



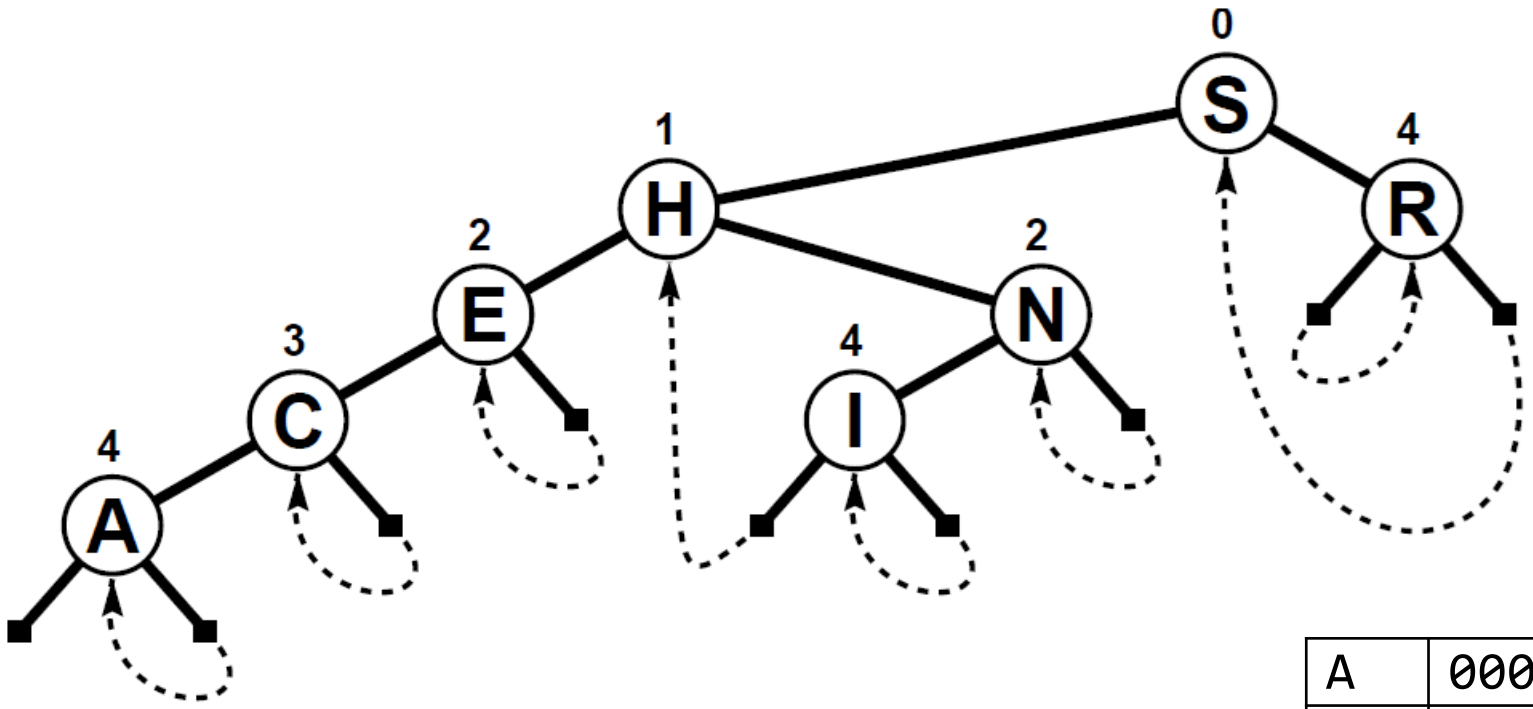
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries



A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

Patricia Tries



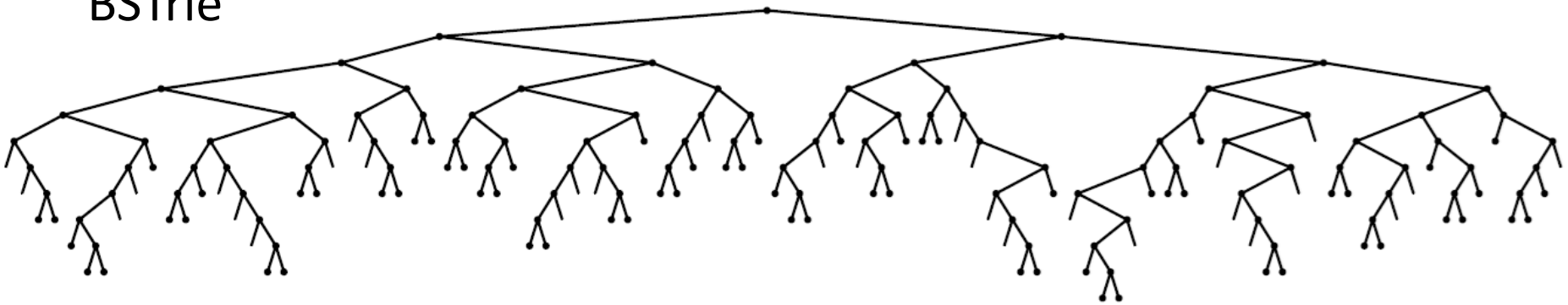
A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110

[Sedgewick] p.654

Patricia Tries vs BSTrie

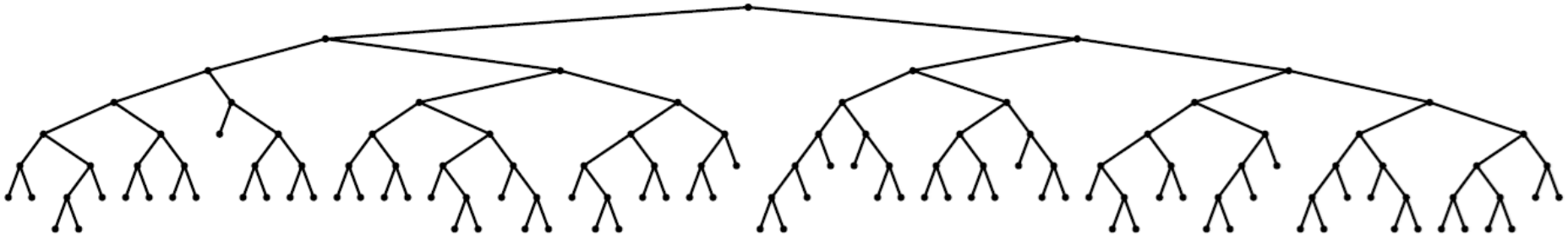
Generated from 200 random keys

BSTrie



[Sedgewick] p.648

Patricia Trie



[Sedgewick] p.657

Patricia Tries

Analysis

- Quintessential radix search method: manages to **identify the bits** that distinguish the search keys and builds them into a data structure with **no surplus nodes**
- Only **one full key comparison** in every search
- Search/insert requires about $\lg n$ bit comparisons on average and $2 \lg n$ bit comparisons in the worst-case in a tree built from n random b-bit keys which is independent of key length.
- Well suited for applications with variable-length keys that are potentially huge (e.g. string search)

Multiway Search Tries

So far, we have always compared one bit at a time. Why not compare **multiple bits** at a time?

- Examining r bits at a time

  speeds up search by a factor of r

  uses nodes with $R = 2^r$ links ("R-ary" or "R-way" tries)

  gives an excellent search time: $O(\log_R n)$ comparisons

- There is a catch

  increasing the number of links leads to **wasted space**

Multiway Search Tries

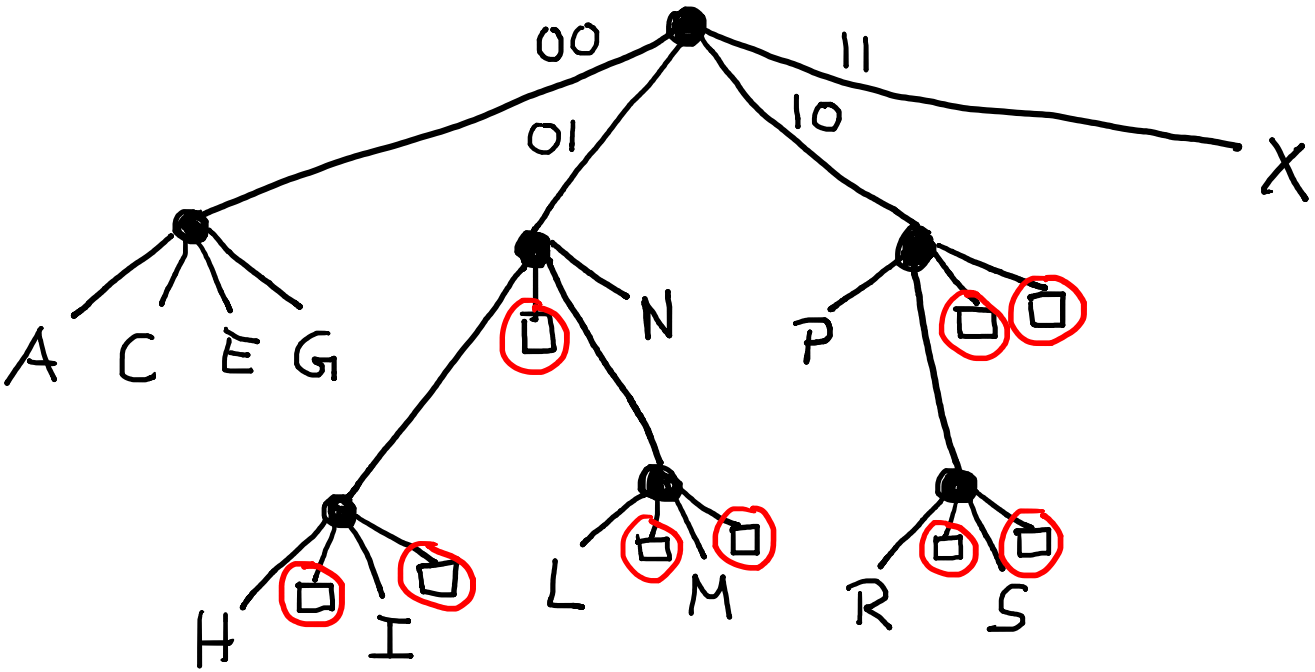
Definition

A **multiway search trie** is a multiway tree that has keys associated with each of its leaves, defined recursively as follows:

- the trie for an empty set of keys is a `NIL` link;
- the trie for a single key is a leaf containing that key;
- and the trie for a set of keys of cardinality greater than one is an internal node with links referring to tries for keys with each possible digit value, with the leading digit considered to be removed for the purpose of constructing the subtrees.

Example: 4-way Search Trie

$R = 2^2$, so we consider two bits at a time



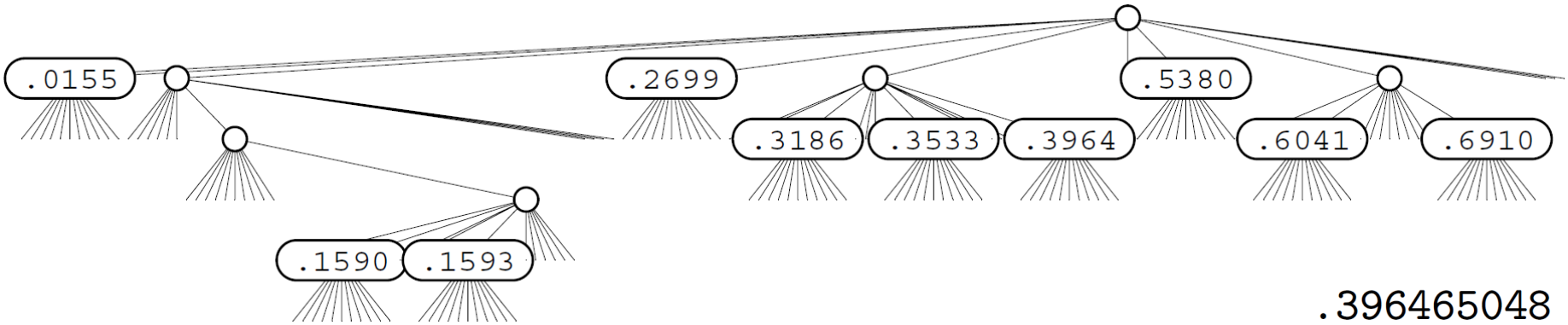
Many unused links!

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100

Example: 10-way Search Trie

$R = 10$

[Sedgewick] p.662



.396465048
.353336658
.318693642
.015583409
.159369371
.691004885
.899854354
.159072306
.604144269
.269971047
.538069659

This 10-way trie represents a sample set of decimal numbers

Existence Tries

Special purpose multiway tries that implements the concept of an existence table

- no associated data; the **keys are the data**
- think of it as a **representation of a set**
- efficient insert and search

Assumptions

- keys are distinct, no key is the prefix of another
- keys are of fixed length or have termination digit

Definition

- The **existence trie** corresponding to a **set of keys** is defined recursively as follows: the trie for an empty set of keys is a `NIL` link; the trie for a nonempty set is a node with links referring to the trie for **each possible key digit**, with the leading digit considered to be removed for the purpose of constructing the subtrees.

Existence Tries

Search: use the 'digits' in the key to guide down the trie; if we reach the link to END at the same time that we run out of key digits, we have a search hit; otherwise we have a search miss

```
MET-EXISTS(t,k,d)
 1: i = DIGIT(k,d)
 2: if t.next[i] == NIL
 3:     return FALSE
 4: if i == END
 5:     return TRUE
 6: else
 7:     return MET-EXISTS(t.next[i],k,d+1)
```

Existence Tries

Insert: search until a `NIL` link is reached, then add nodes for each of the remaining characters in the key

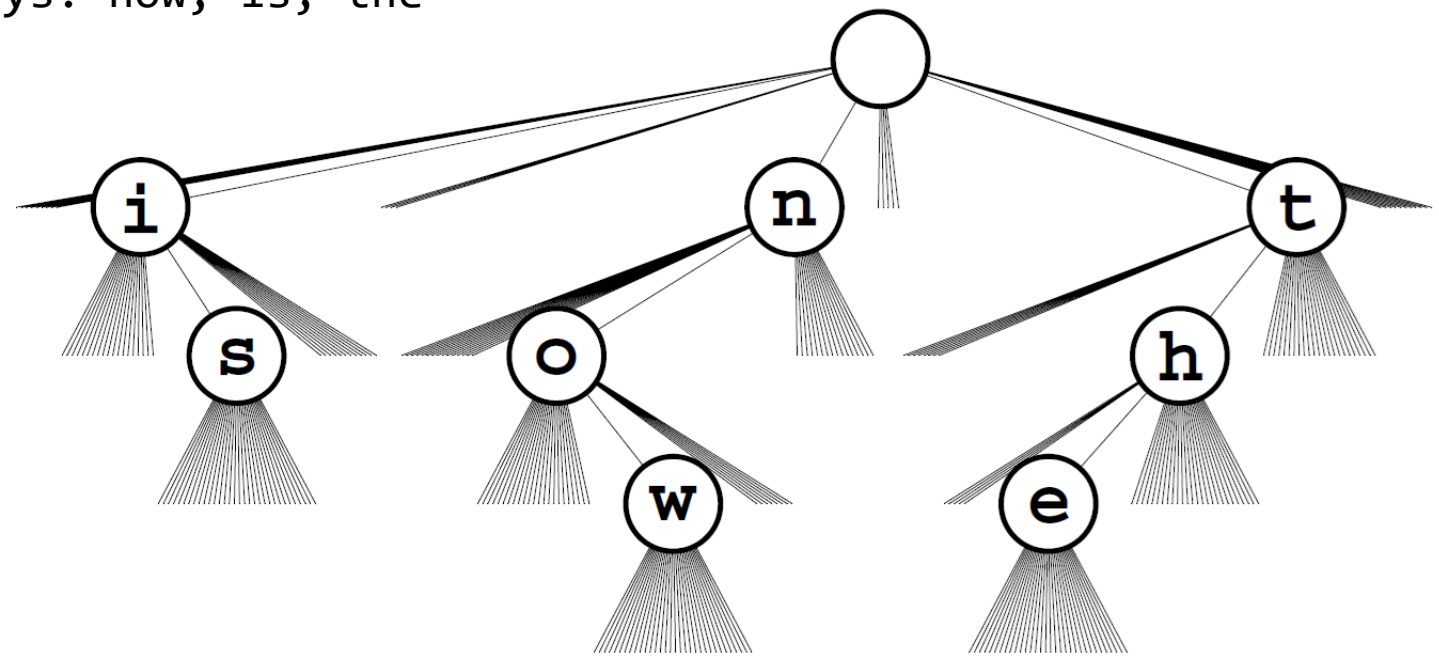
```
MET-INSERT(t,k,d)
 1: i = DIGIT(k,d)
 2: if t.next[i] == NIL
 3:     t.next[i] = NEWNODE()
 4: if i ≠ END
 5:     return MET-INSERT(t.next[i],k,d+1)
```

Example: 27-way Existence Trie

Given an alphabet Σ with $|\Sigma| = 26$ unique characters

the corresponding existence trie has 26+1 links at each node, one for each character plus one for a terminal symbol `END`

`Keys: now, is, the`

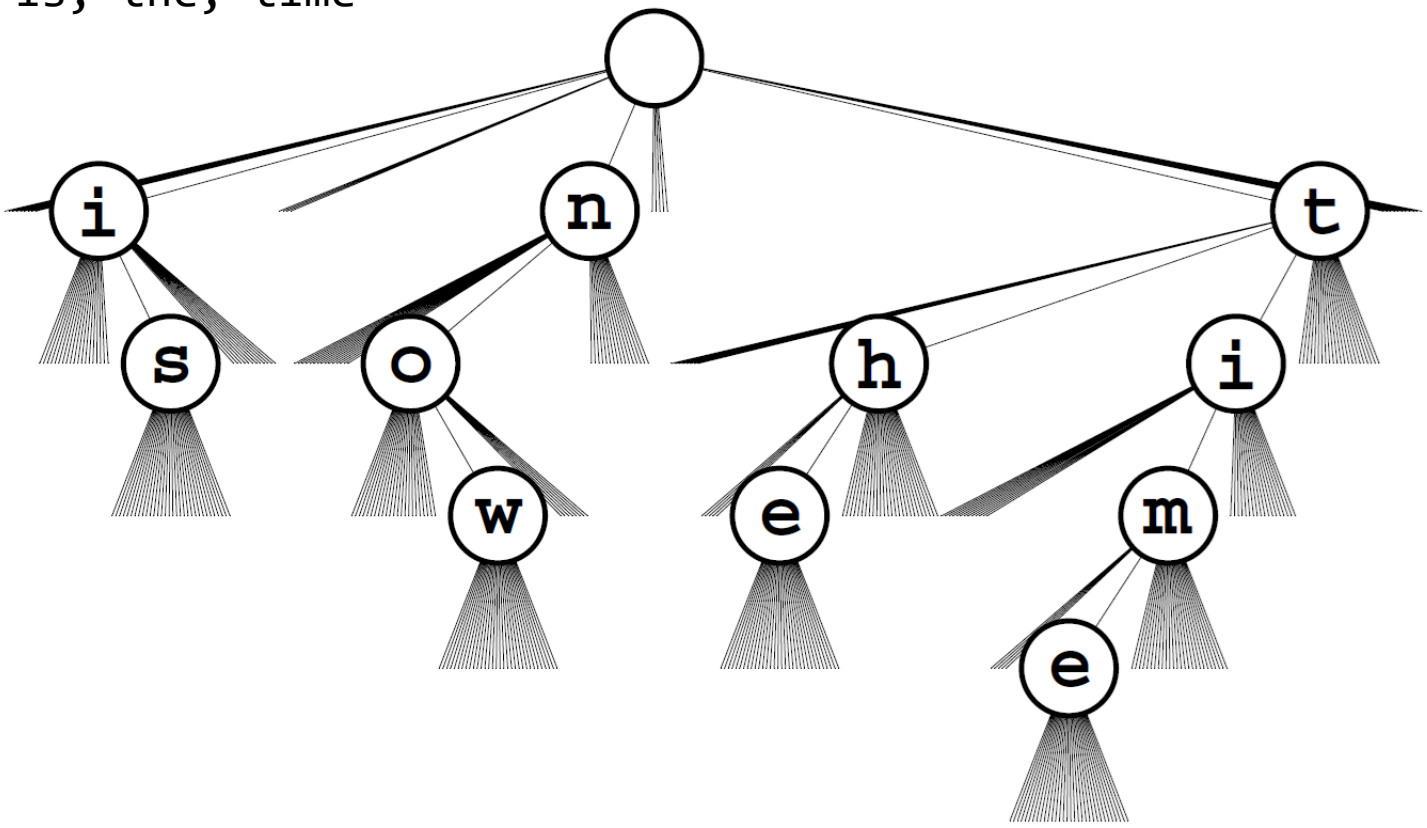


Example: 27-way Existence Trie

Given an alphabet Σ with $|\Sigma| = 26$ unique characters

the corresponding existence trie has 26+1 links at each node, one for each character plus one for a terminal symbol `END`

`Keys: now, is, the, time`

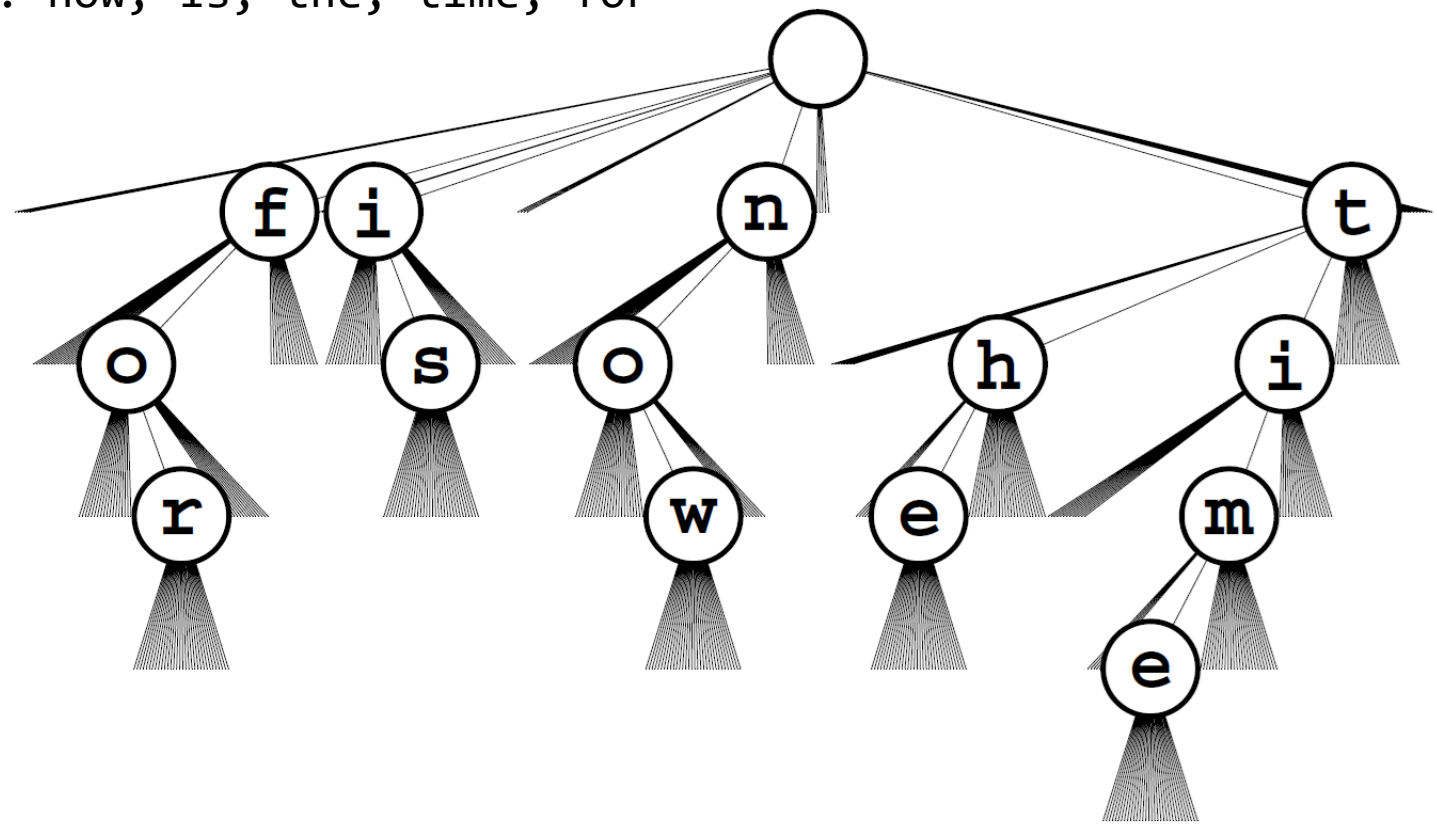


Example: 27-way Existence Trie

Given an alphabet Σ with $|\Sigma| = 26$ unique characters

the corresponding existence trie has 26+1 links at each node, one for each character plus one for a terminal symbol `END`

`Keys: now, is, the, time, for`



Some Observations for Multiway Search Tries

Analysis

- Search/insert requires about $\log_R n$ comparisons on average in a trie built from n random keys
- The number of links in an R-way trie built from n random keys is about $Rn/\ln R$

Suppose that we take the typical value of $R = 256$ and that we have n random 64-bit keys

Search will take $(\lg n)/8$ character comparisons (8 at most)

We will use fewer than $47n$ links

If we take $R = 65536$, we could cut down the search cost to 4 character comparisons, but would require over $5900n$ links

Ternary Search Tries

TSTs are a compact representation of tries

- Each node has **a character** and **three links**

  links correspond to '**less than**', '**equal to**', '**greater than**'

- Characters corresponding to keys are found when traversing the middle links

Search: compare the first character in the key with the root; if it is less, go left; if it is greater, go right; if it is equal take the middle link

  if we traverse the middle link of a node with terminal character END then we have a search hit

  otherwise we have a search miss

Insert: search, then add remaining characters

Example: Ternary Search Tries

Insert

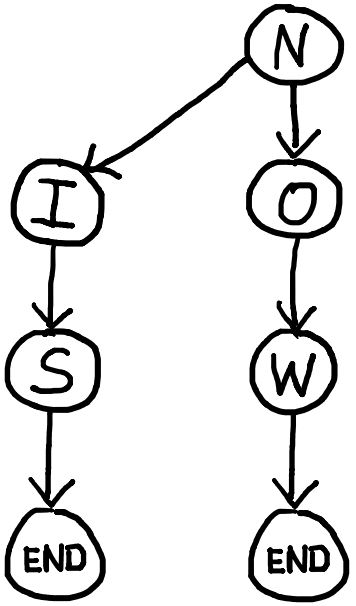
NOW

Example: Ternary Search Tries

Insert

NOW

IS



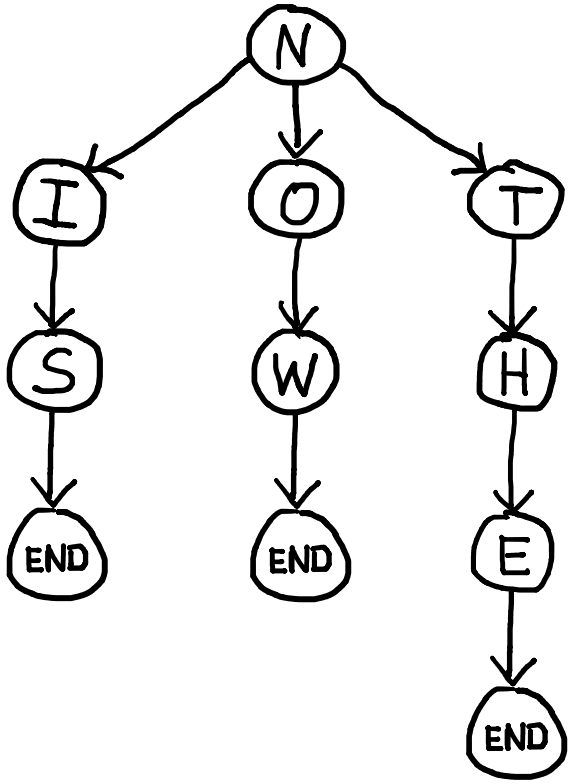
Example: Ternary Search Tries

Insert

NOW

IS

THE



Example: Ternary Search Tries

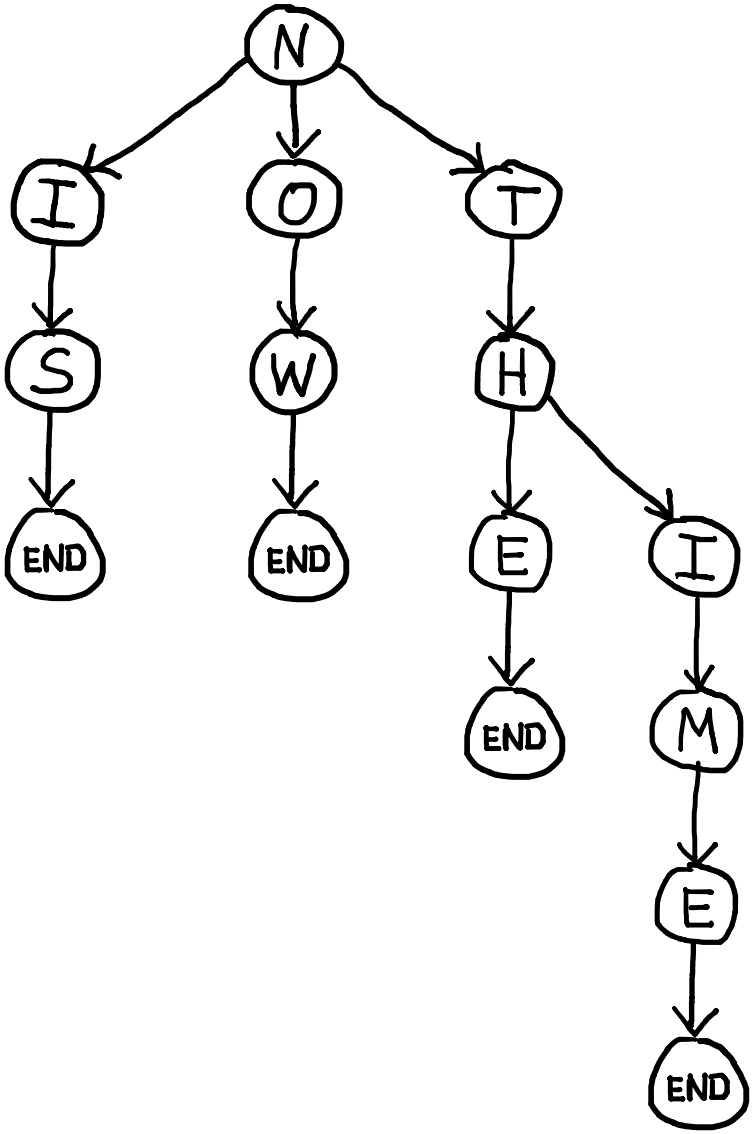
Insert

NOW

IS

THE

TIME



Example: Ternary Search Tries

Insert

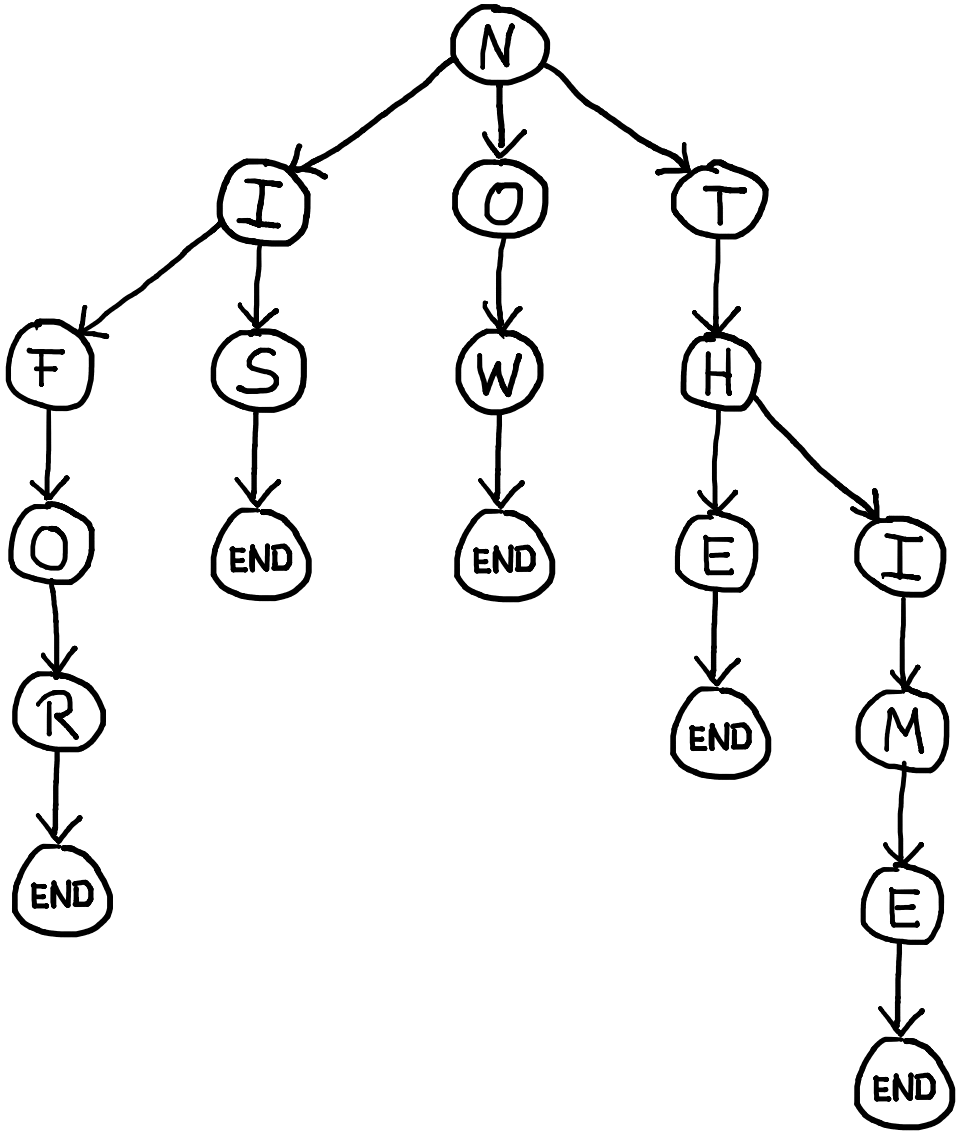
NOW

IS

THE

TIME

FOR



Example: Ternary Search Tries

Insert

NOW

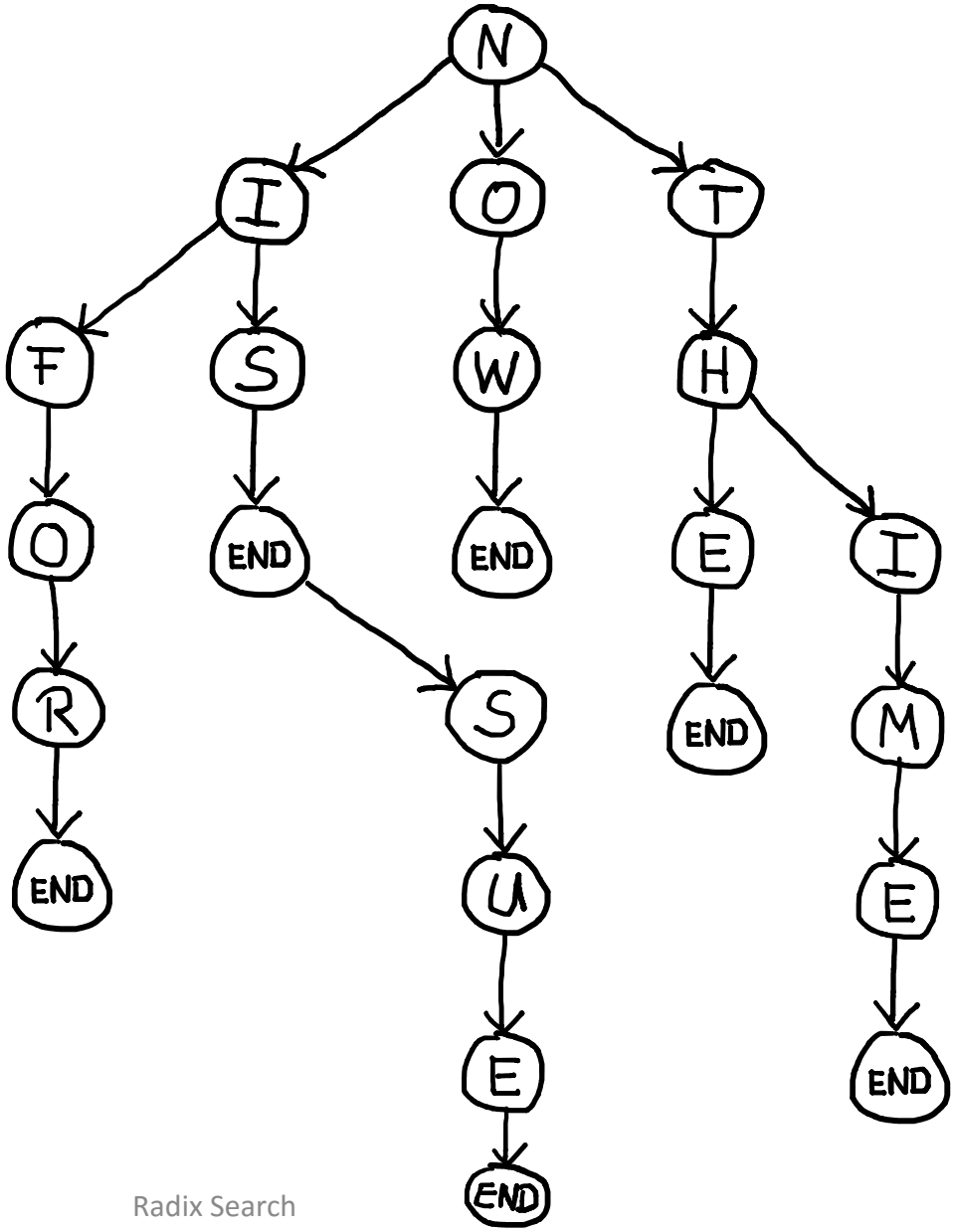
IS

THE

TIME

FOR

ISSUE



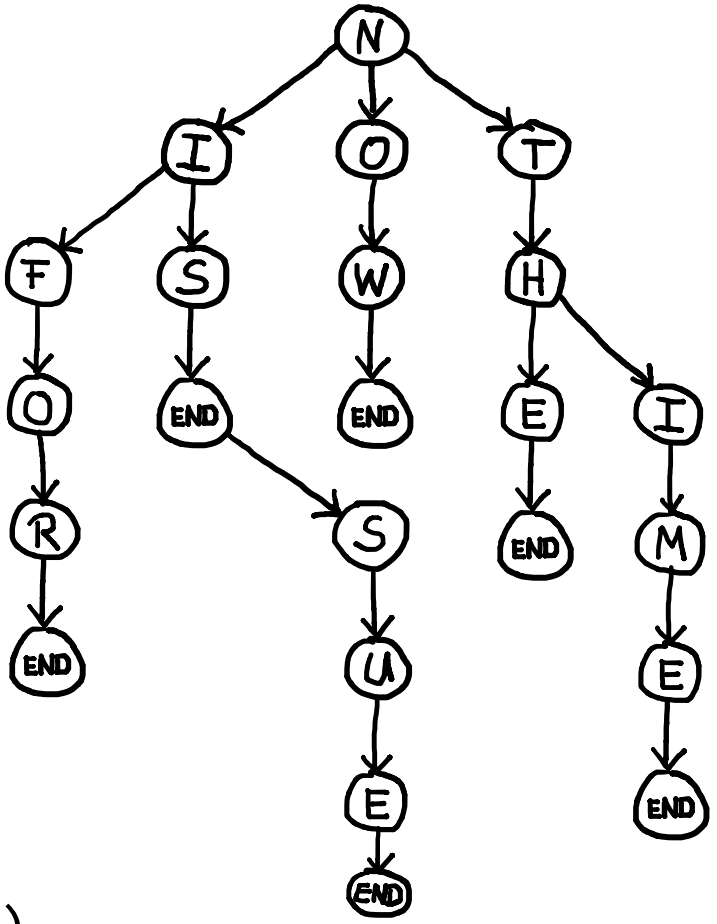
Note: relative value of END is arbitrary, but must be fixed by convention as either greatest or least value of the digits

Ternary Search Tries

```
TST-EXISTS(t,k,d)
 1: if t == NIL
 2:     return FALSE
 3: i = DIGIT(k,d)
 4: td = t.digit
 5: if i == END and td == END
 6:     return TRUE
 7: if i < td
 8:     return TST-EXISTS(t.left,k,d)
 9: if i > td
10:     return TST-EXISTS(t.right,k,d)
11: if i == td
12:     return TST-EXISTS(t.middle,k,d+1)
```

Ternary Search Tries

```
TST-EXISTS(t,k,d)
 1: if t == NIL
 2:     return FALSE
 3: i = DIGIT(k,d)
 4: td = t.digit
 5: if i == END and td == END
 6:     return TRUE
 7: if i < td
 8:     return TST-EXISTS(t.left,k,d)
 9: if i > td
10:     return TST-EXISTS(t.right,k,d)
11: if i == td
12:     return TST-EXISTS(t.middle,k,d+1)
```



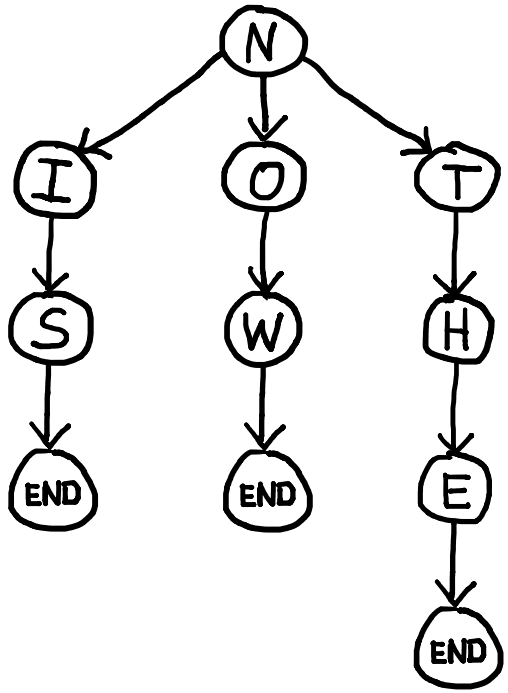
Ternary Search Tries

```
TST-INSERT(t,k,d)
 1: i = DIGIT(k,d)
 2: n = NIL
 3: if t == NIL
 4:     n = NEWNODE(i)
 5: else n = t
 6: nd = n.digit
 7: if i == END and nd == END
 8:     return n
 9: if i < nd
10:     n.left = TST-INSERT(n.left,k,d)
11: else if i > nd
12:     n.right = TST-INSERT(n.right,k,d)
13: else if i == nd
14:     n.middle = TST-INSERT(n.middle,k,d+1)
15: return n
```

Ternary Search Tries

```
TST-INSERT(t,k,d)
 1: i = DIGIT(k,d)
 2: n = NIL
 3: if t == NIL
 4:     n = NEWNODE(i)
 5: else n = t
 6: nd = n.digit
 7: if i == END and nd == END
 8:     return n
 9: if i < nd
10:     n.left = TST-INSERT(n.left,k,d)
11: else if i > nd
12:     n.right = TST-INSERT(n.right,k,d)
13: else if i == nd
14:     n.middle = TST-INSERT(n.middle,k,d+1)
15: return n
```

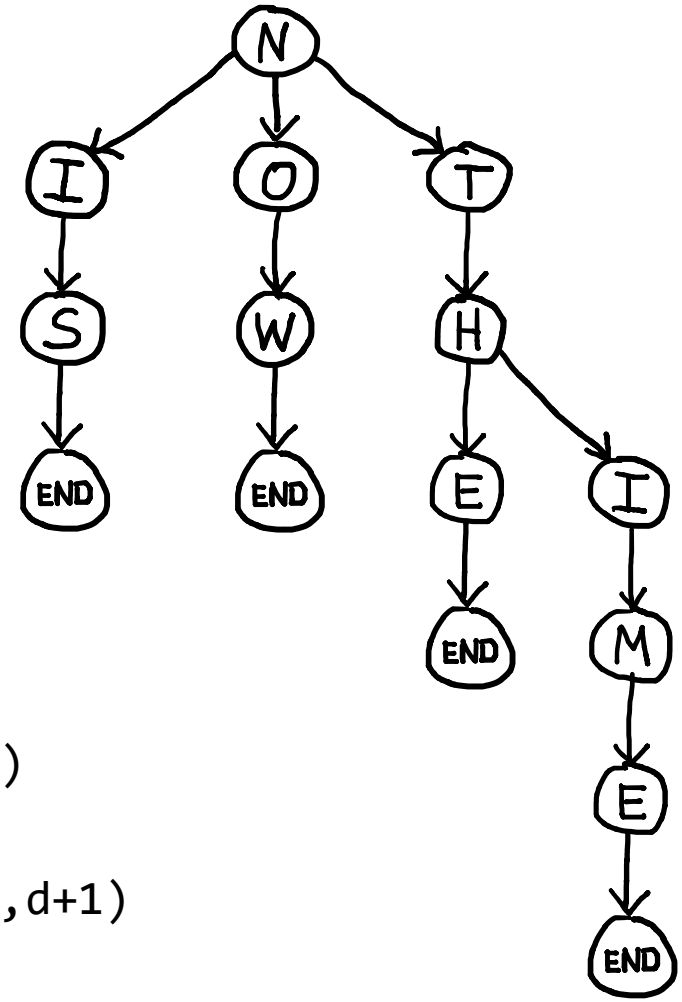
Insert TIME



Ternary Search Tries

```
TST-INSERT(t,k,d)
 1: i = DIGIT(k,d)
 2: n = NIL
 3: if t == NIL
 4:     n = NEWNODE(i)
 5: else n = t
 6: nd = n.digit
 7: if i == END and nd == END
 8:     return n
 9: if i < nd
10:     n.left = TST-INSERT(n.left,k,d)
11: else if i > nd
12:     n.right = TST-INSERT(n.right,k,d)
13: else if i == nd
14:     n.middle = TST-INSERT(n.middle,k,d+1)
15: return n
```

Insert TIME



Some Benefits of TSTs

- TSTs adapt gracefully to…
  - non-randomness in key structure

    e.g. consider a URL as a key
  - long keys that are non-fixed-length character strings

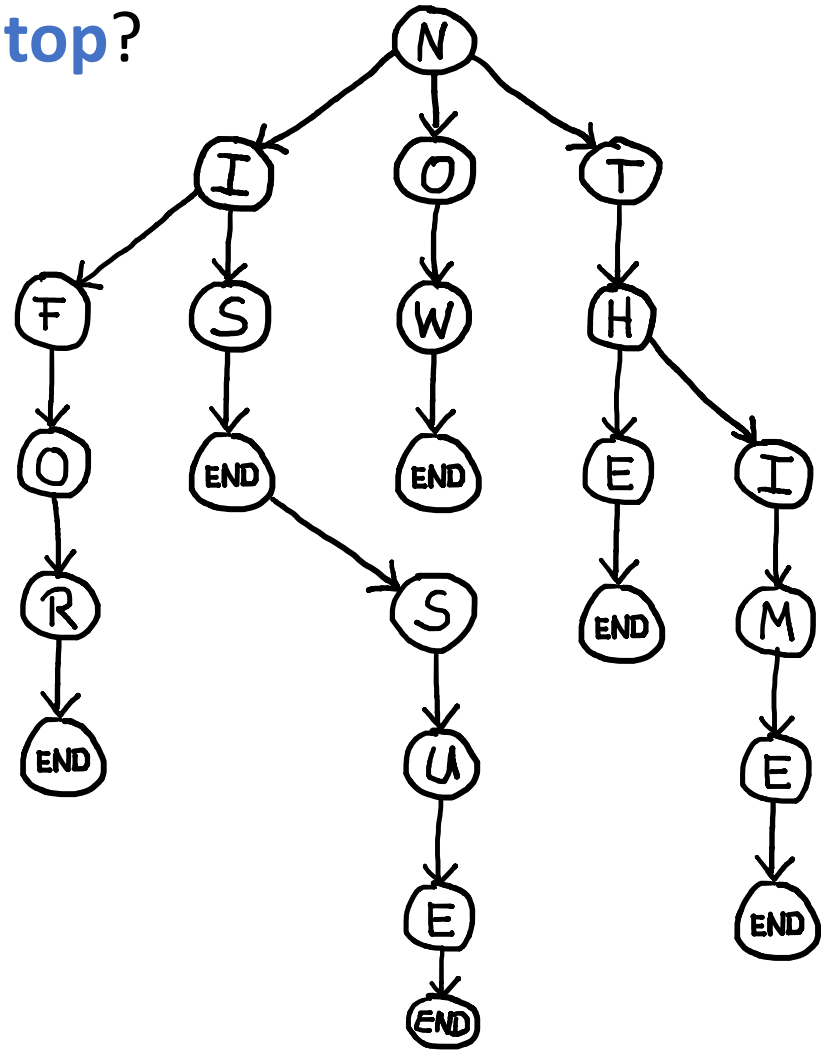
    a digit corresponds to a character in the string
  - large character sets (such as for Asian languages)

    in practice, many characters rarely used in a given key set
- Search misses in TSTs tend to be efficient
  - much less than the $\lg n$ comparisons of a BST
- TSTs support partial-match searches on strings

  e.g. ‘F*’ or ‘*T*’

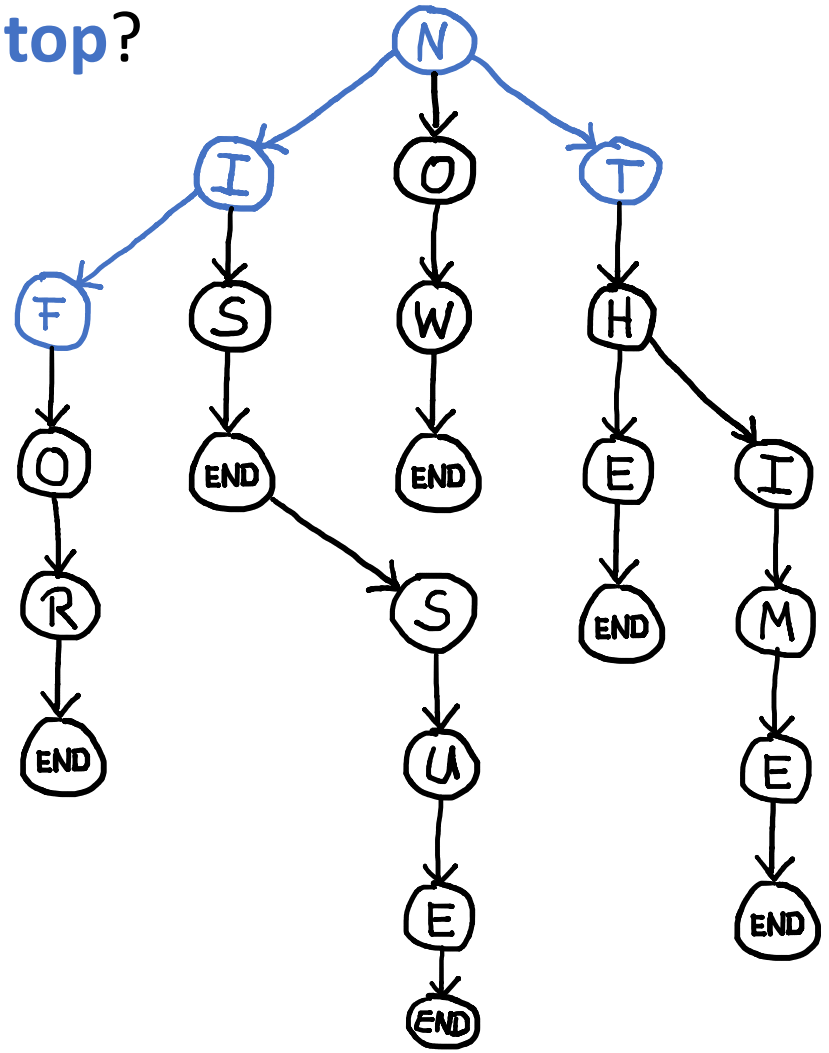
Some Observations for TSTs

What tends to happen at the **top**?



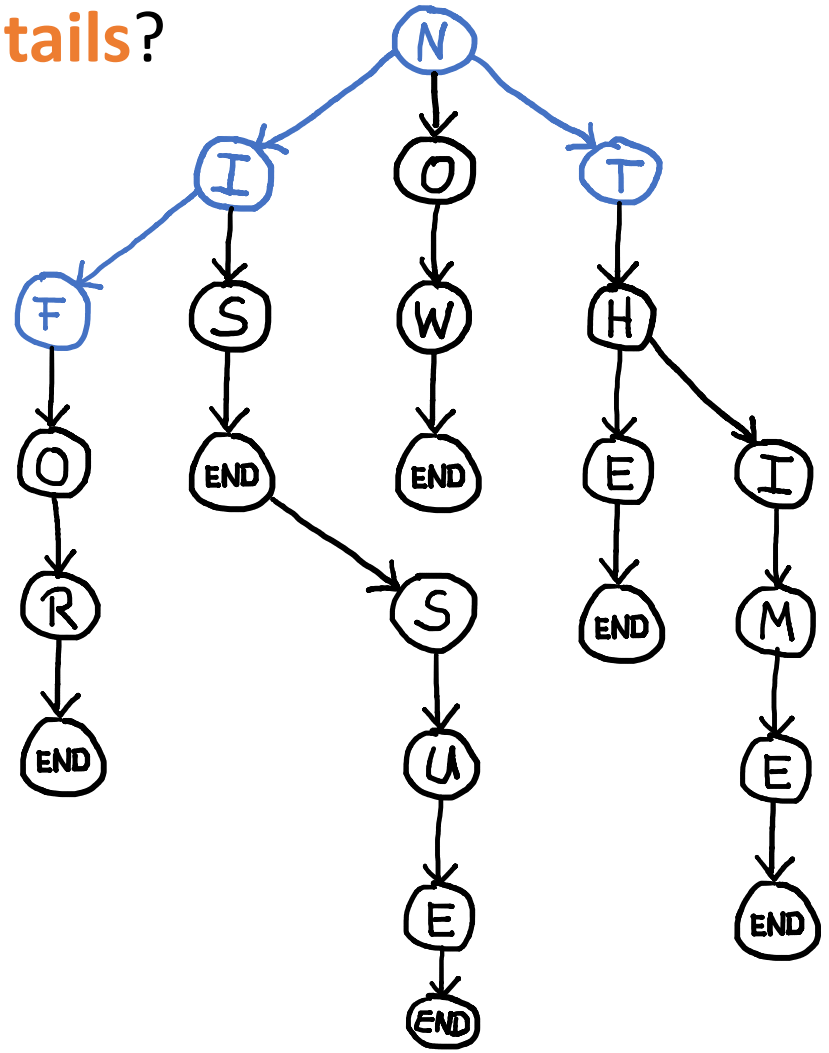
Some Observations for TSTs

What tends to happen at the **top**?



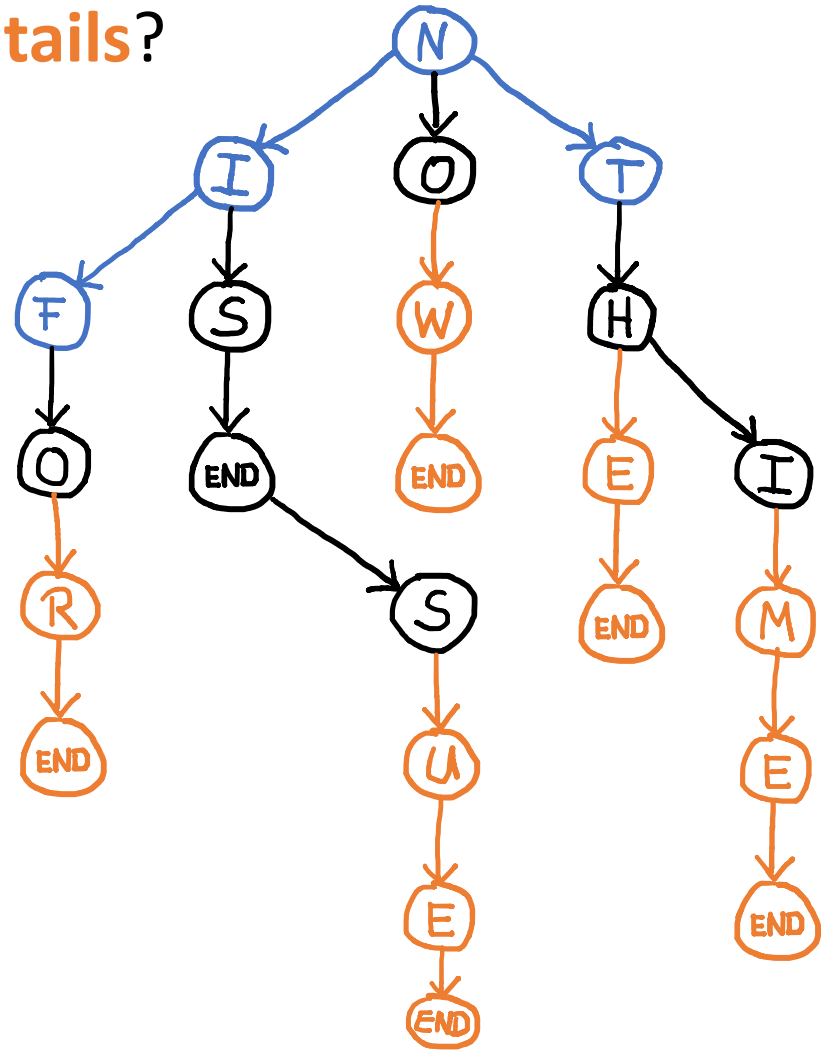
Some Observations for TSTs

What tends to happen at the **tails**?



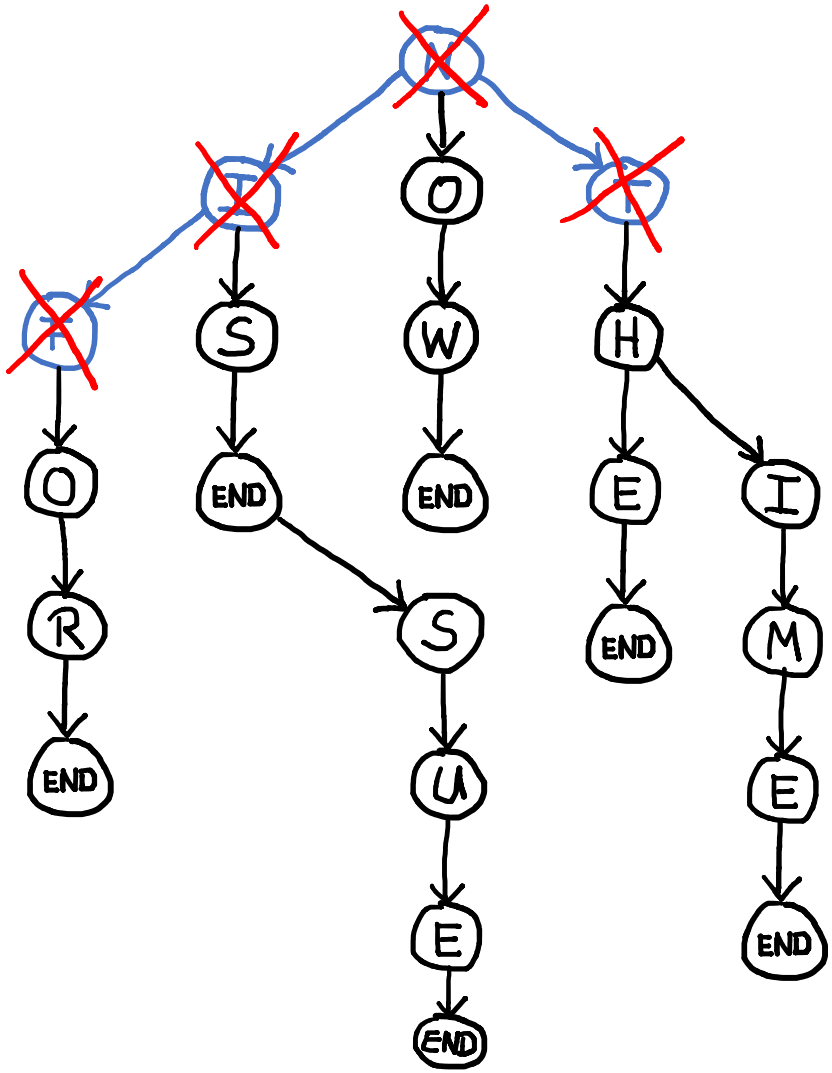
Some Observations for TSTs

What tends to happen at the **tails**?



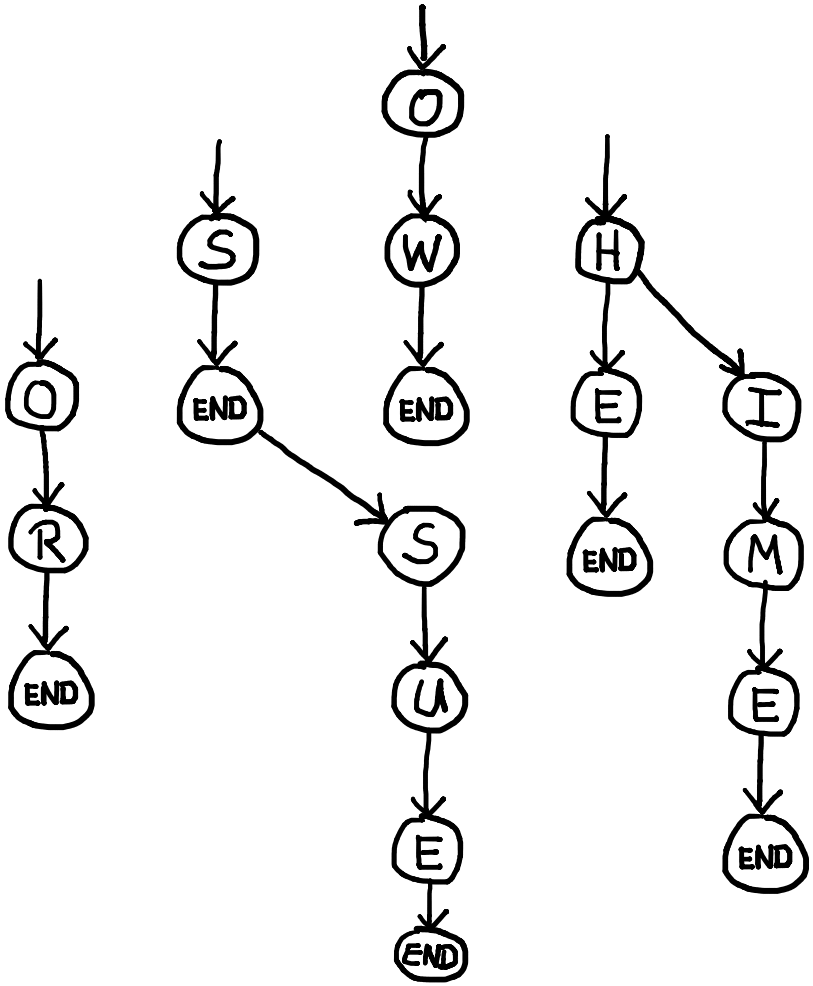
Optimising TSTs

First idea: wide head



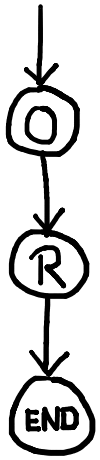
Optimising TSTs

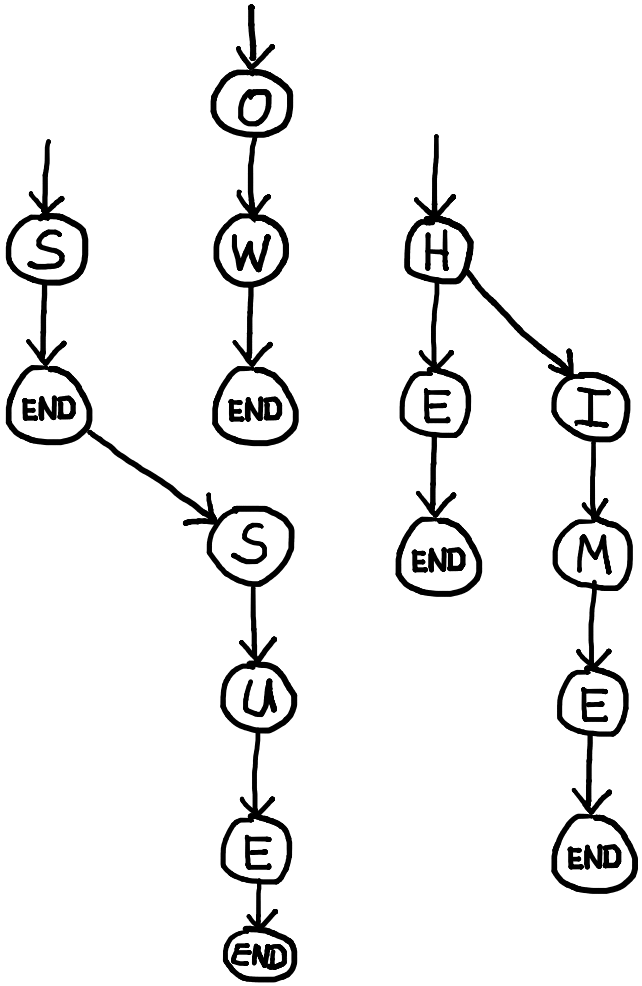
First idea: wide head



Optimising TSTs

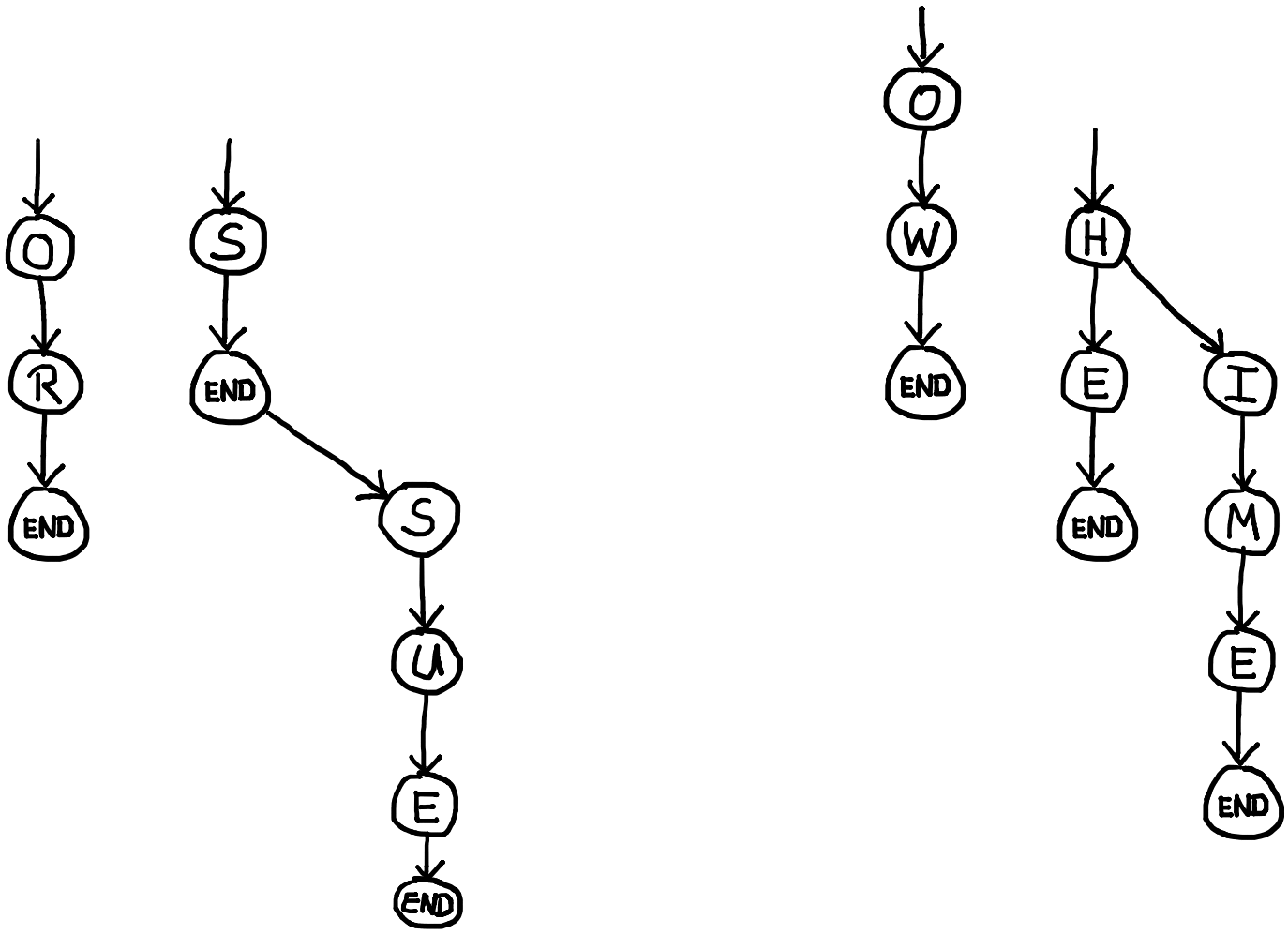
First idea: wide head





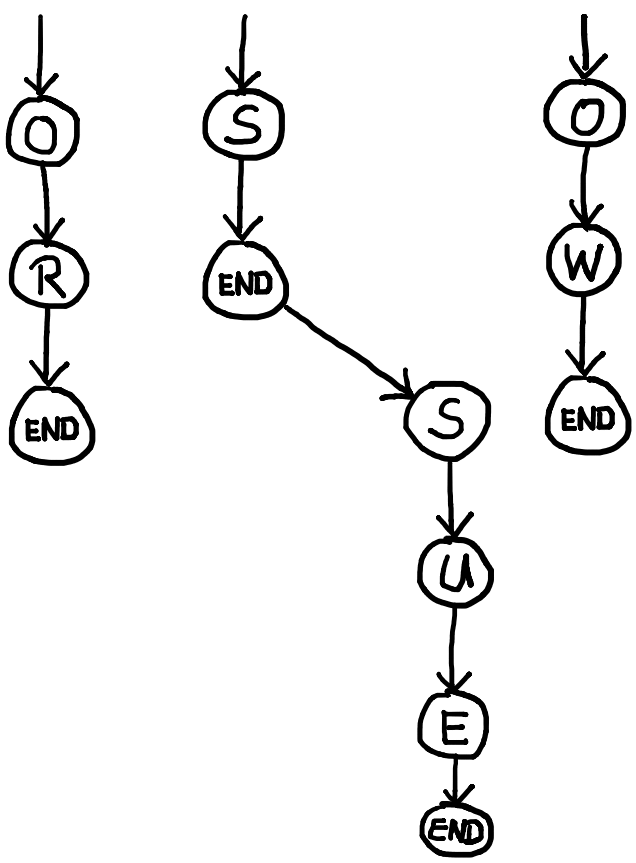
Optimising TSTs

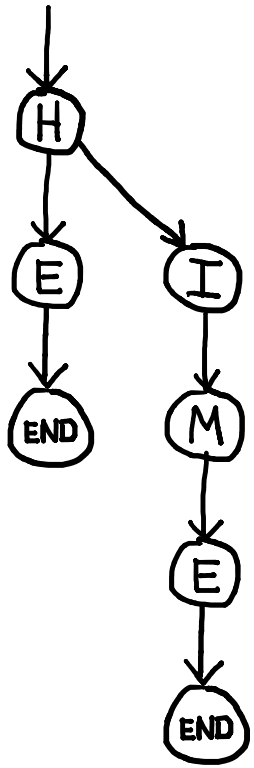
First idea: wide head



Optimising TSTs

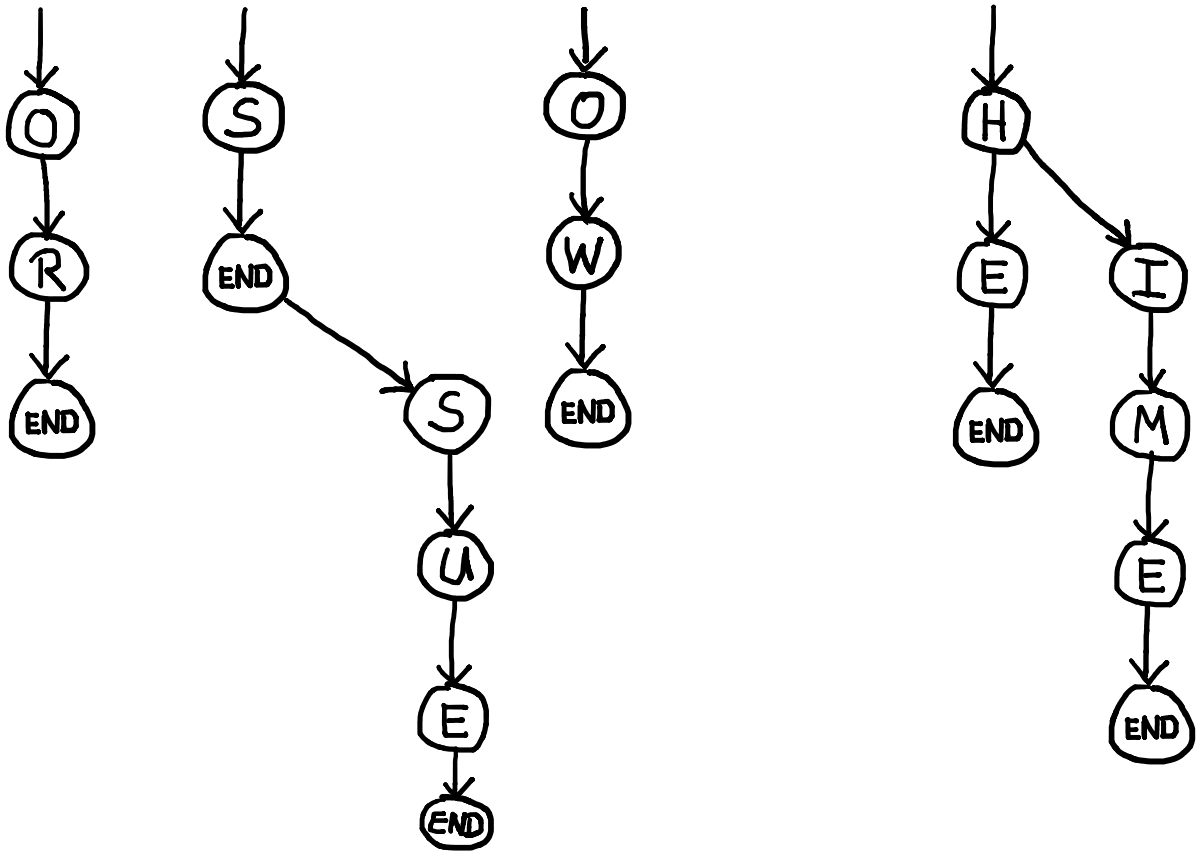
First idea: wide head





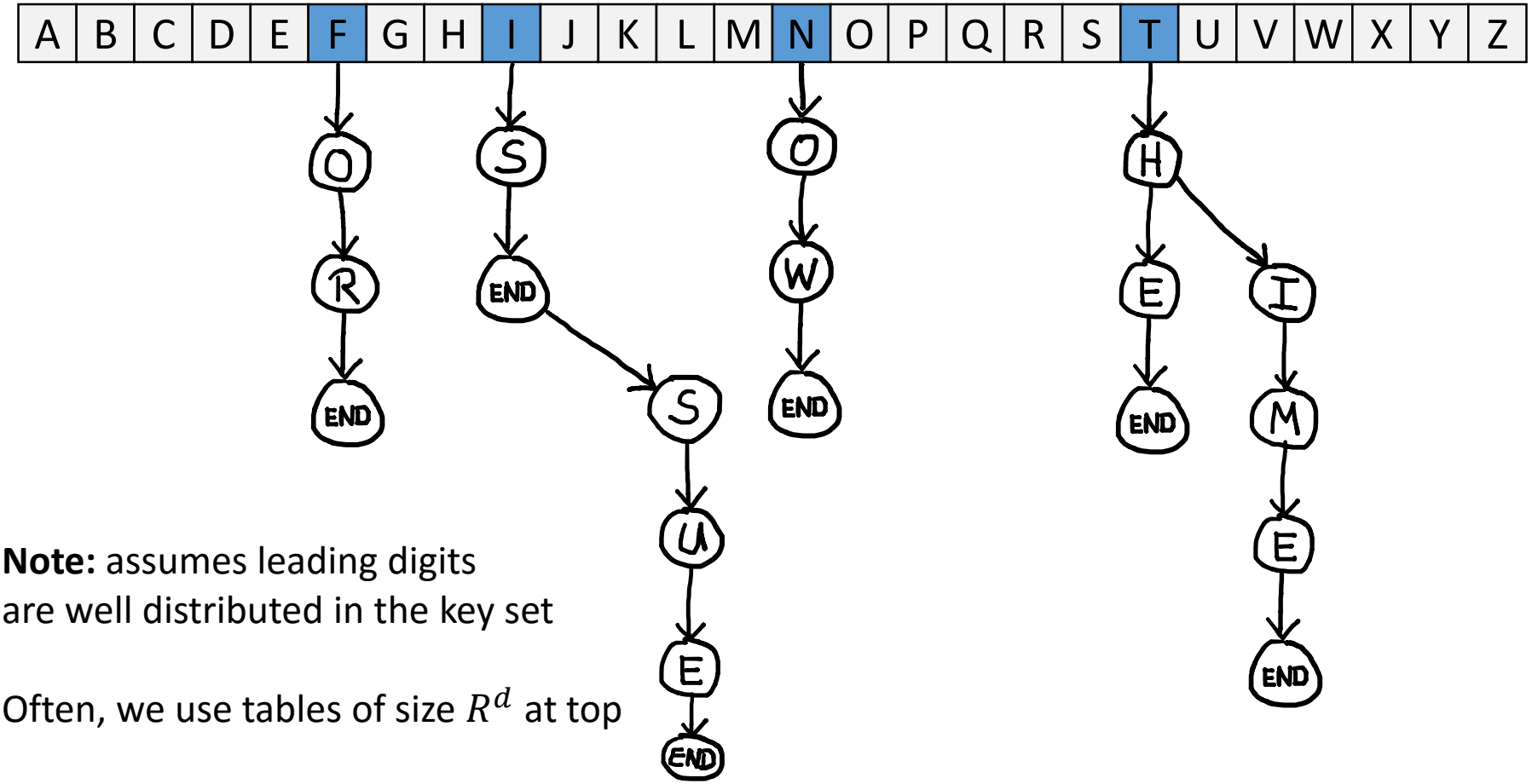
Optimising TSTs

First idea: wide head



Optimising TSTs

First idea: wide head, create a table of R TSTs



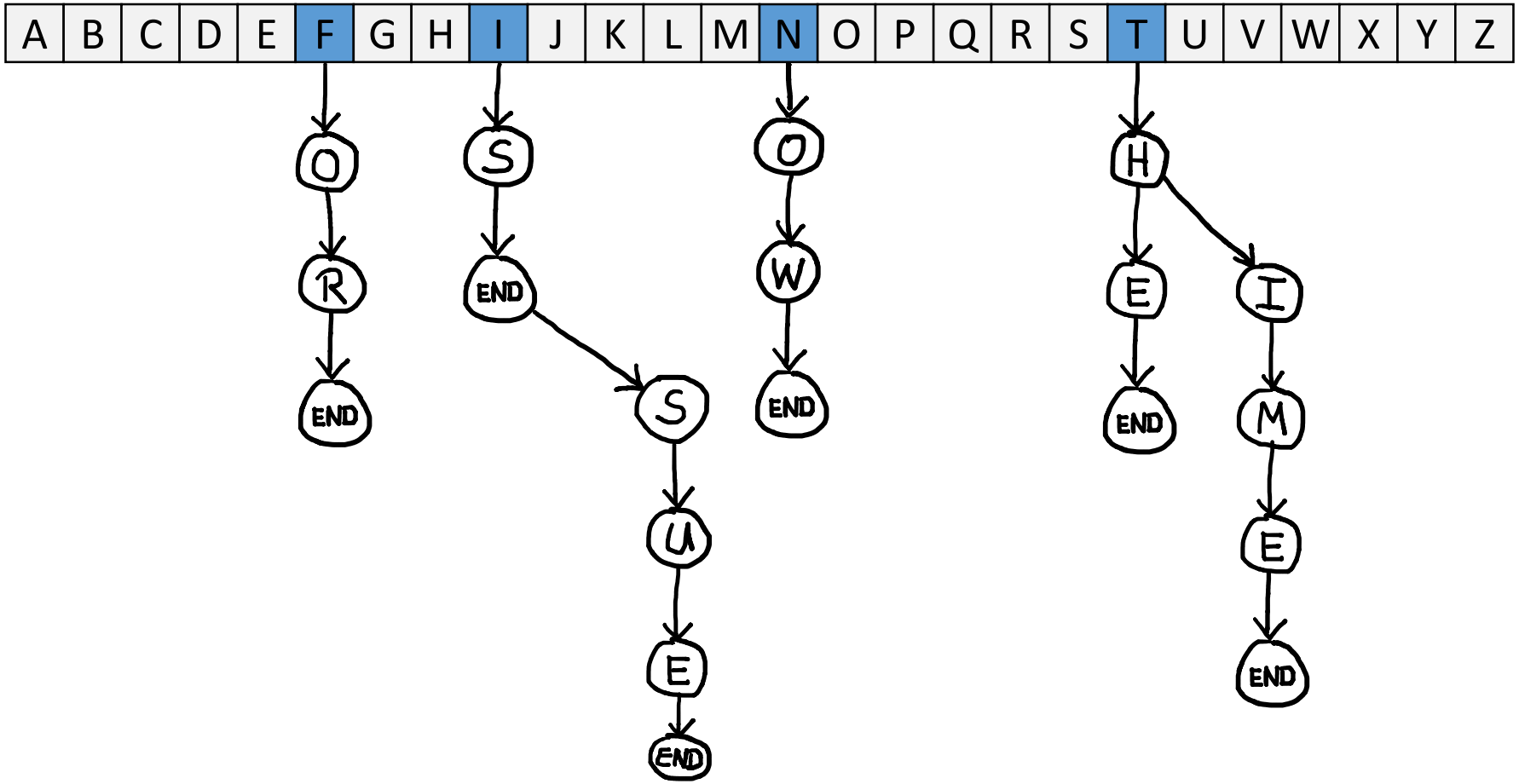
Note: assumes leading digits are well distributed in the key set

Often, we use tables of size R^d at top

Hybrid between Existence Trie and TST

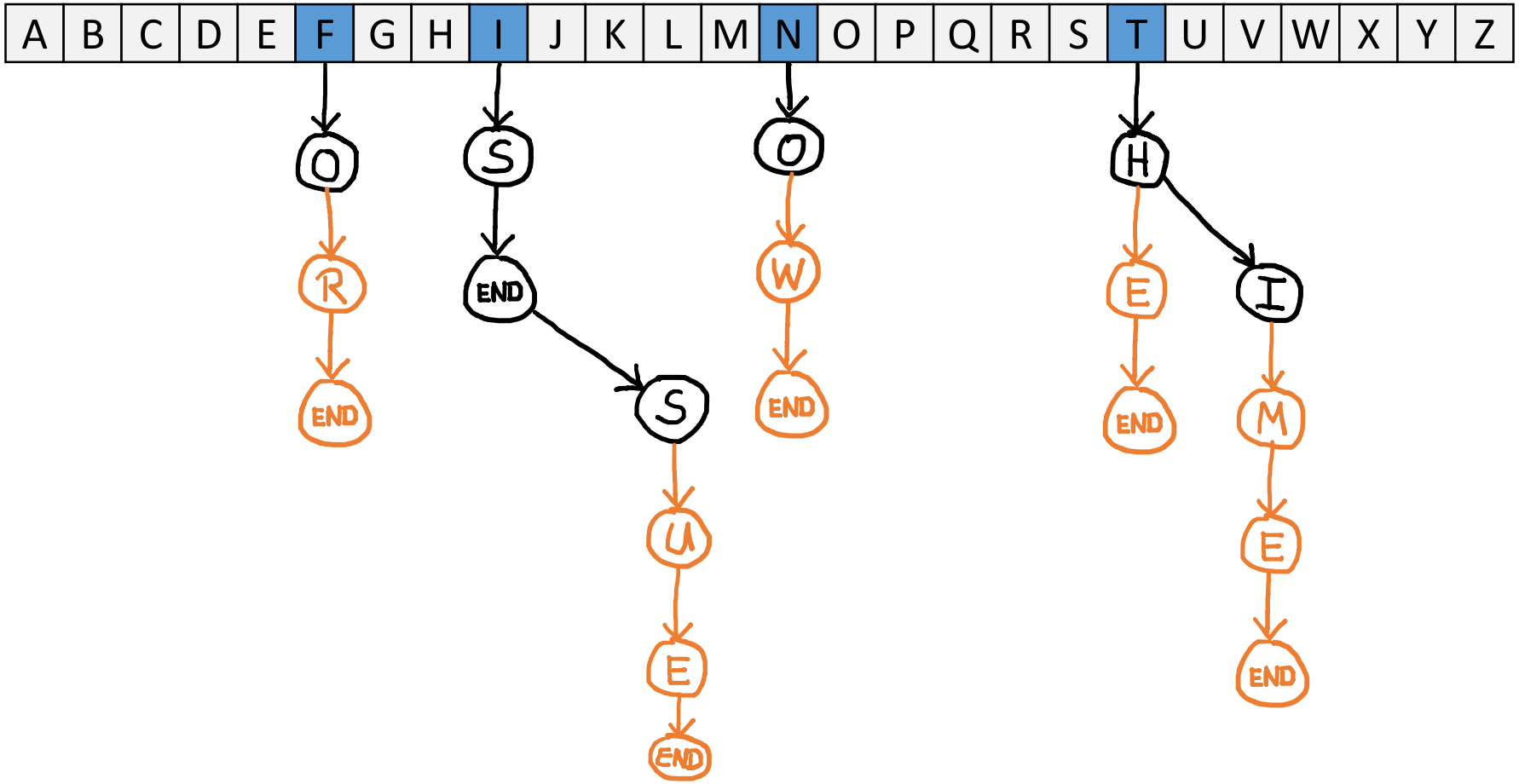
Optimising TSTs

Second idea: compact tails



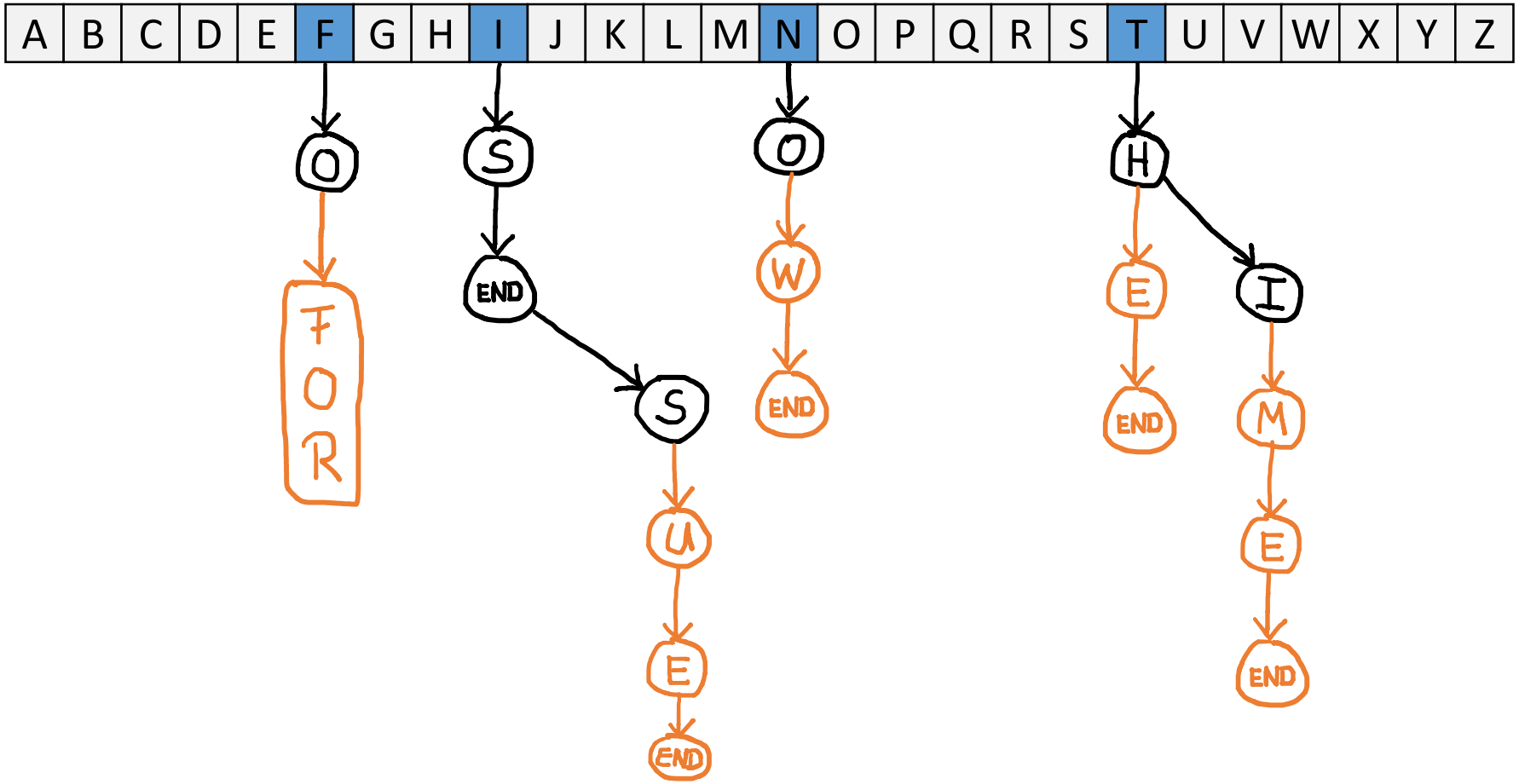
Optimising TSTs

Second idea: compact tails, store keys at leaves



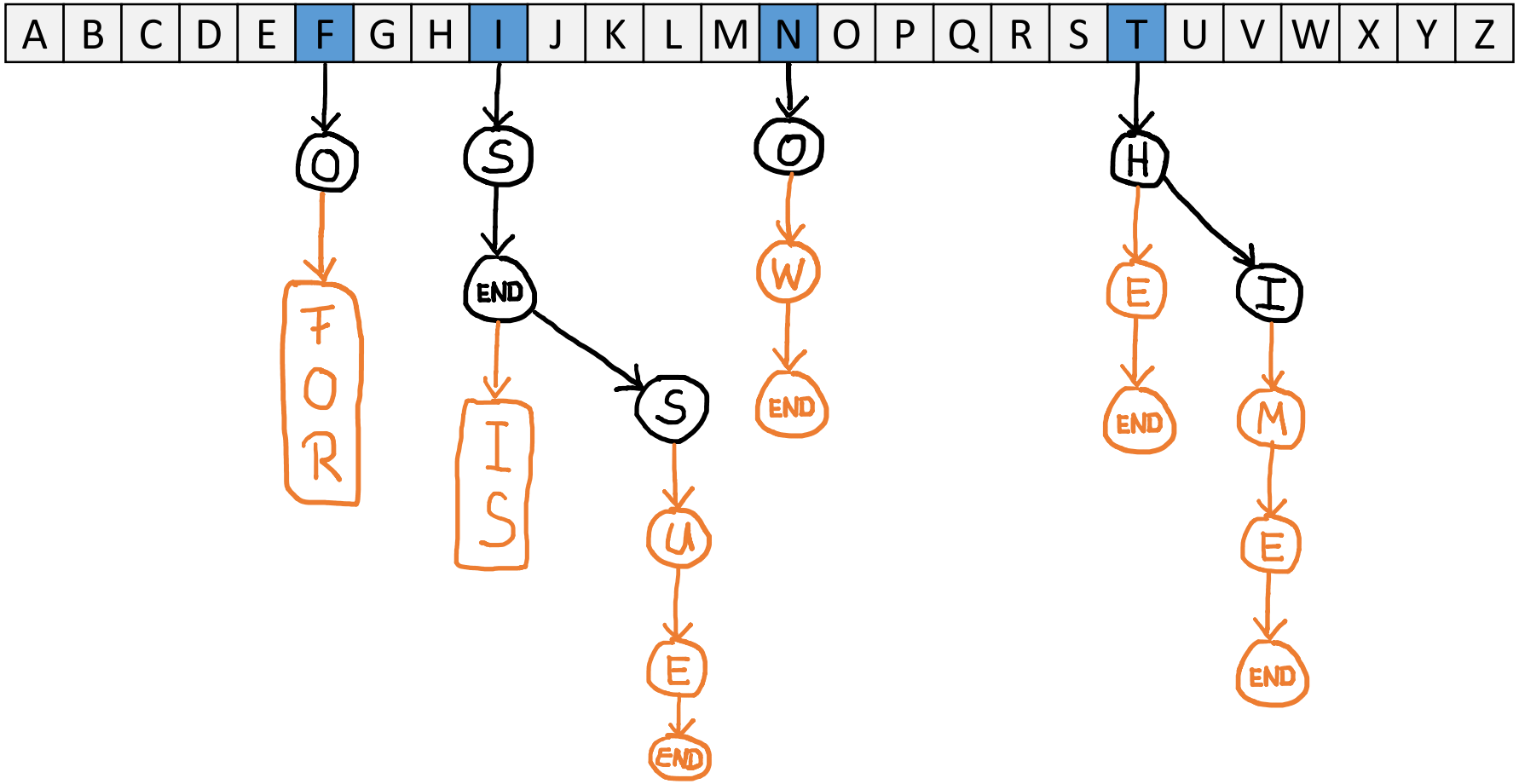
Optimising TSTs

Second idea: compact tails, store keys at leaves



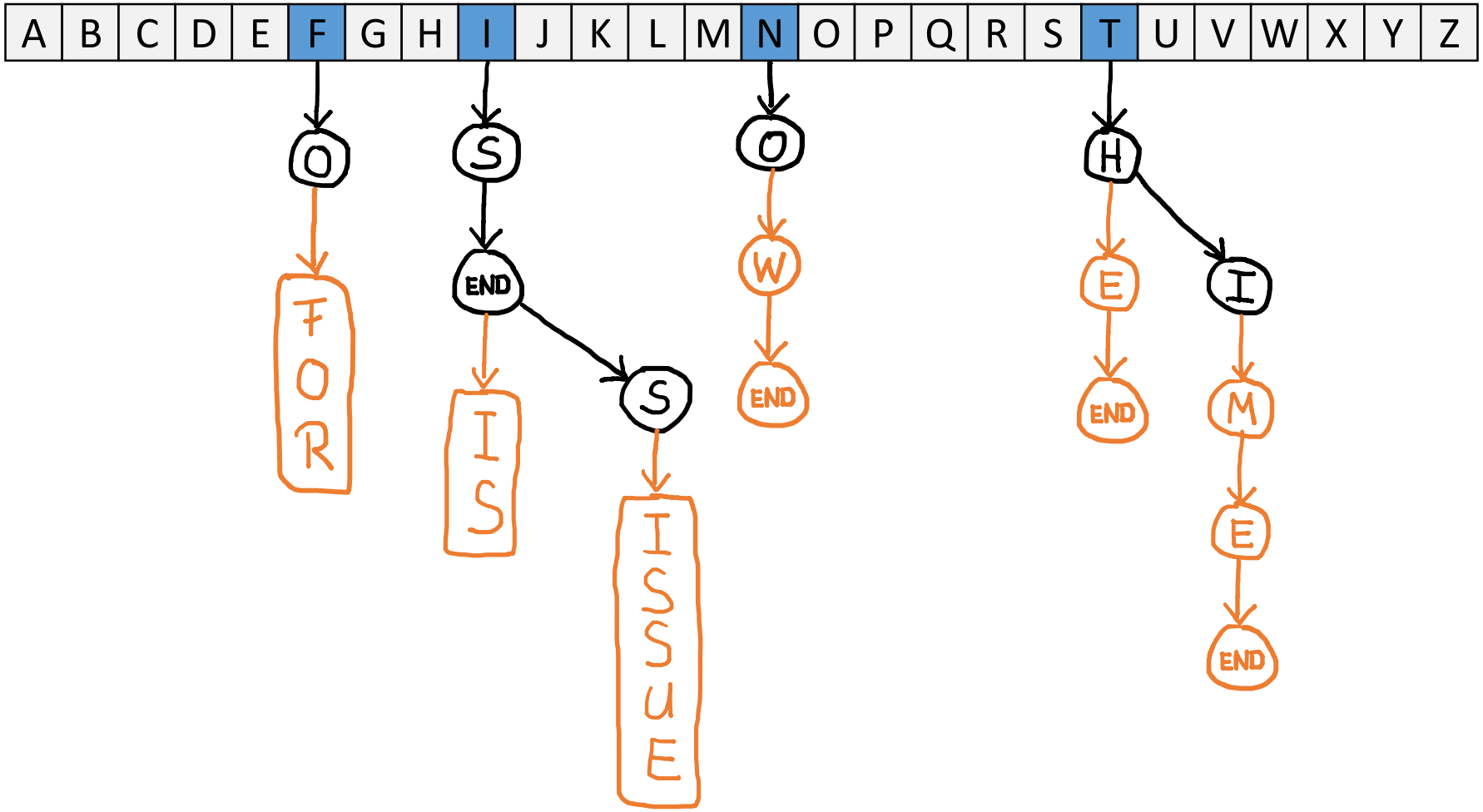
Optimising TSTs

Second idea: compact tails, store keys at leaves



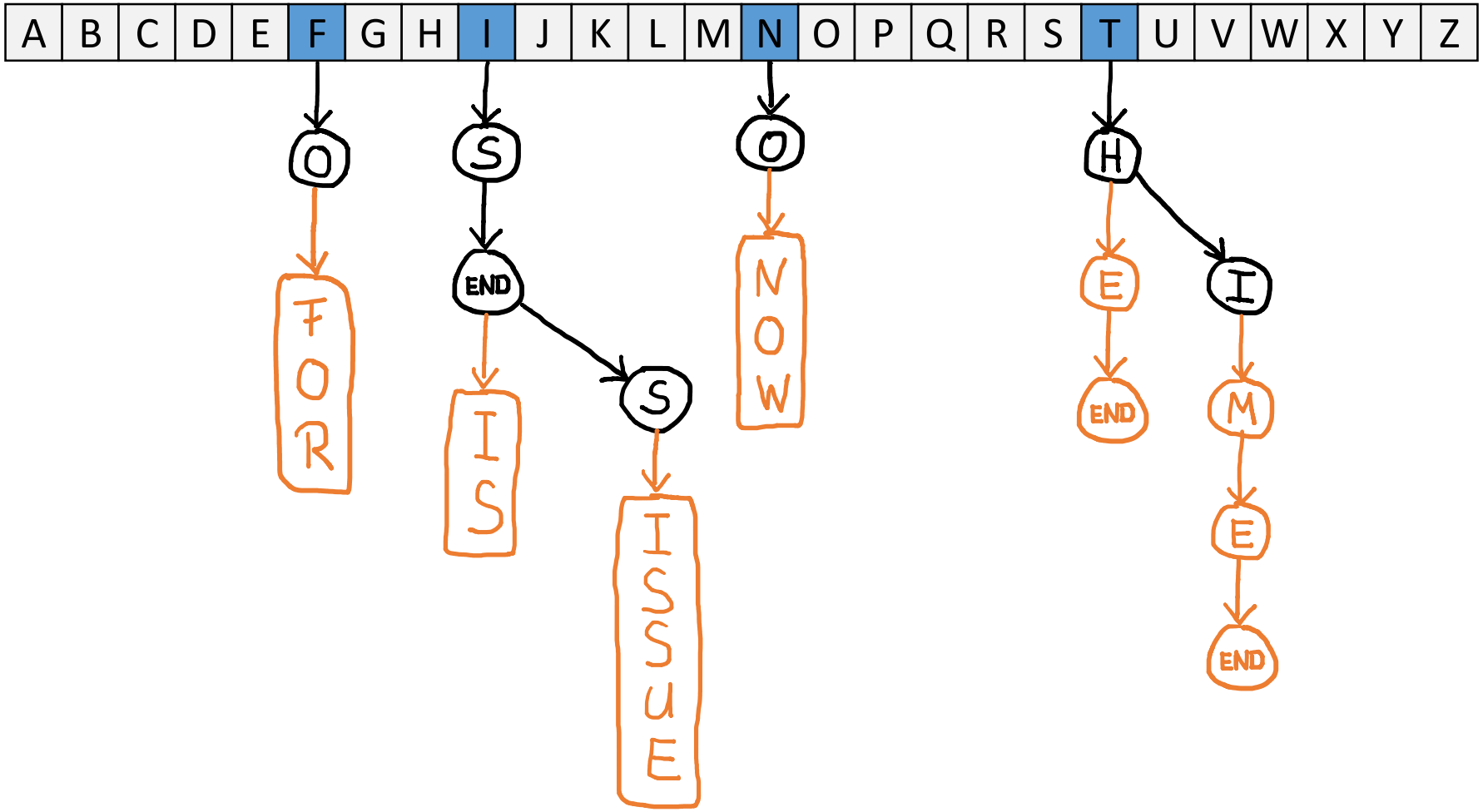
Optimising TSTs

Second idea: compact tails, store keys at leaves



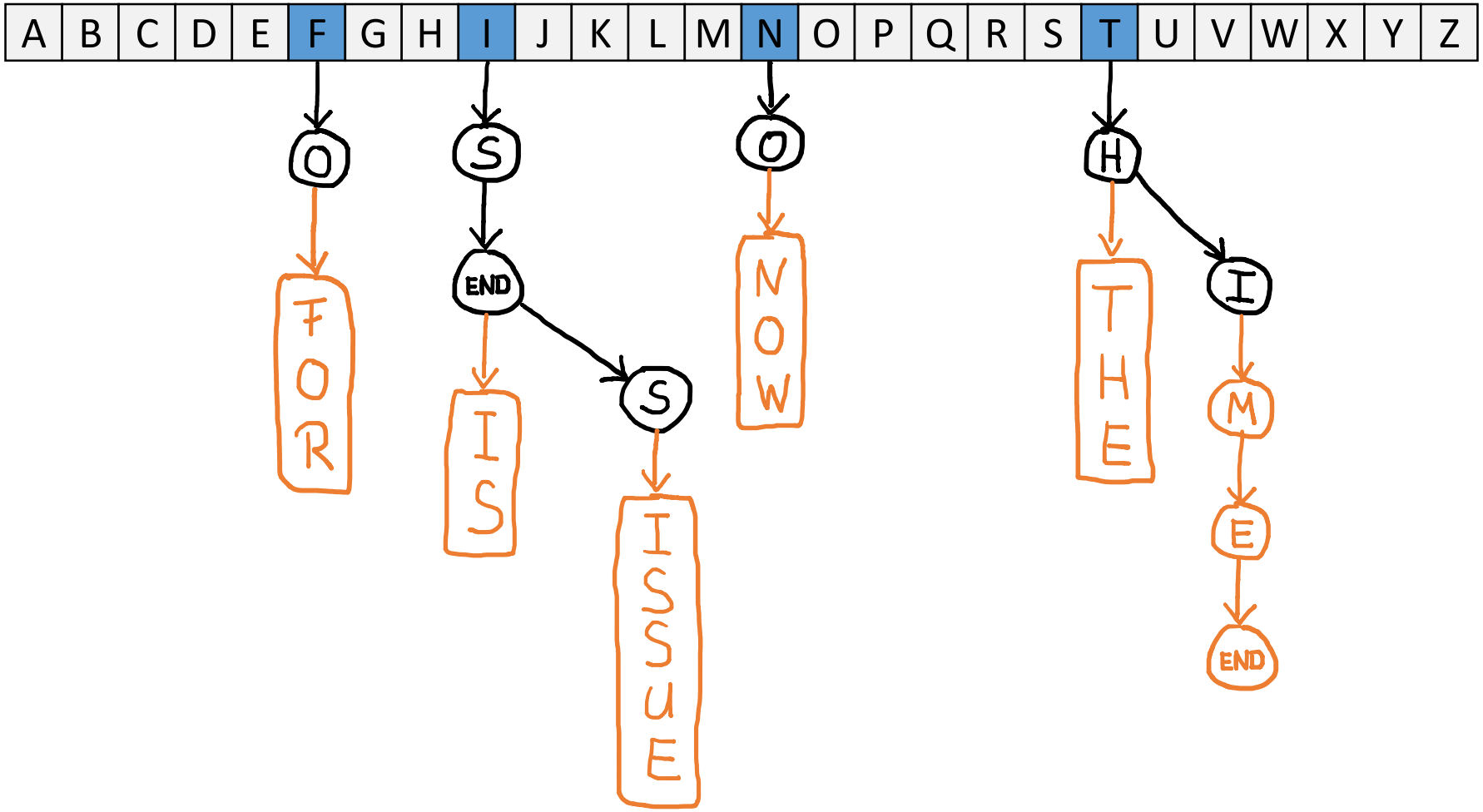
Optimising TSTs

Second idea: compact tails, store keys at leaves



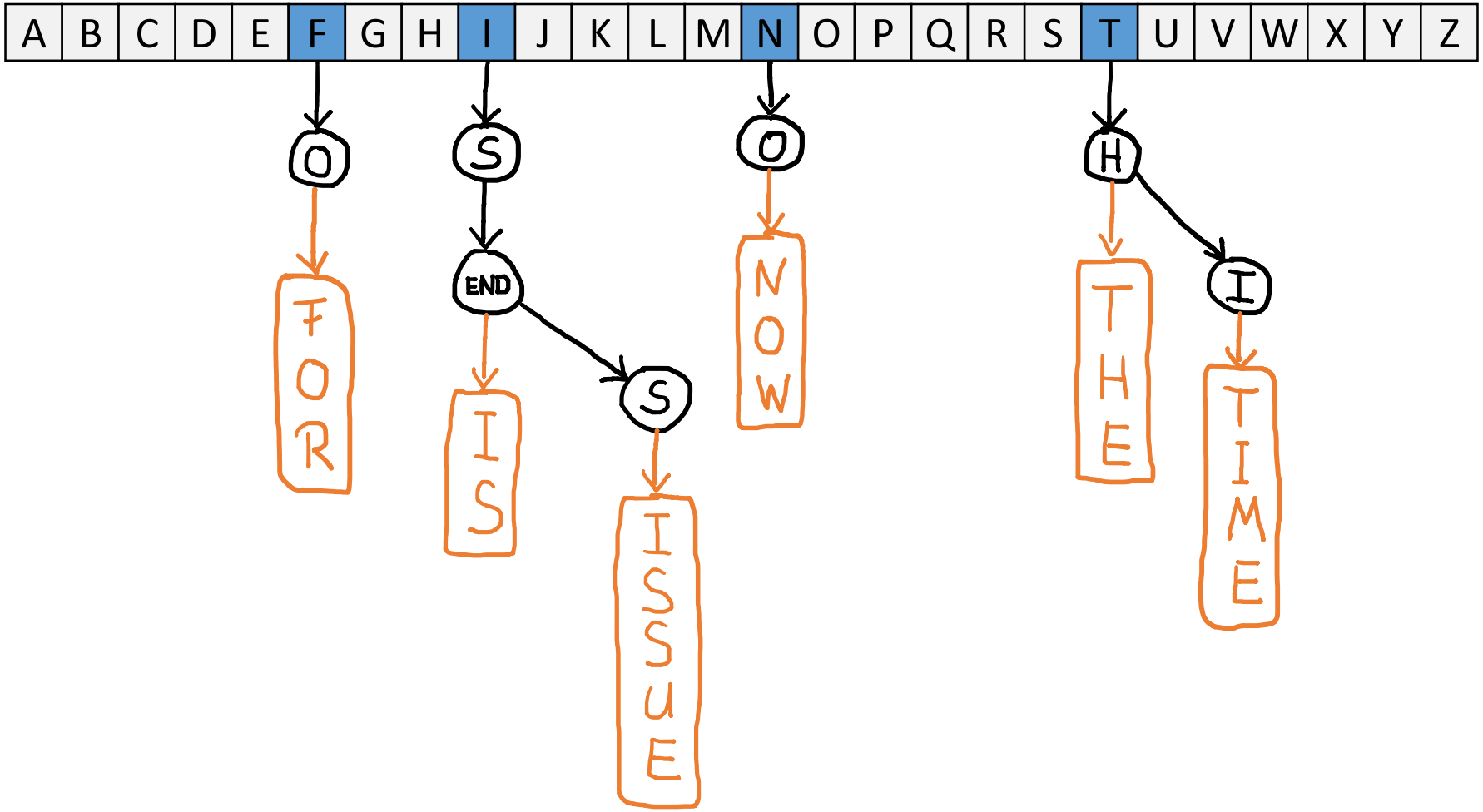
Optimising TSTs

Second idea: compact tails, store keys at leaves



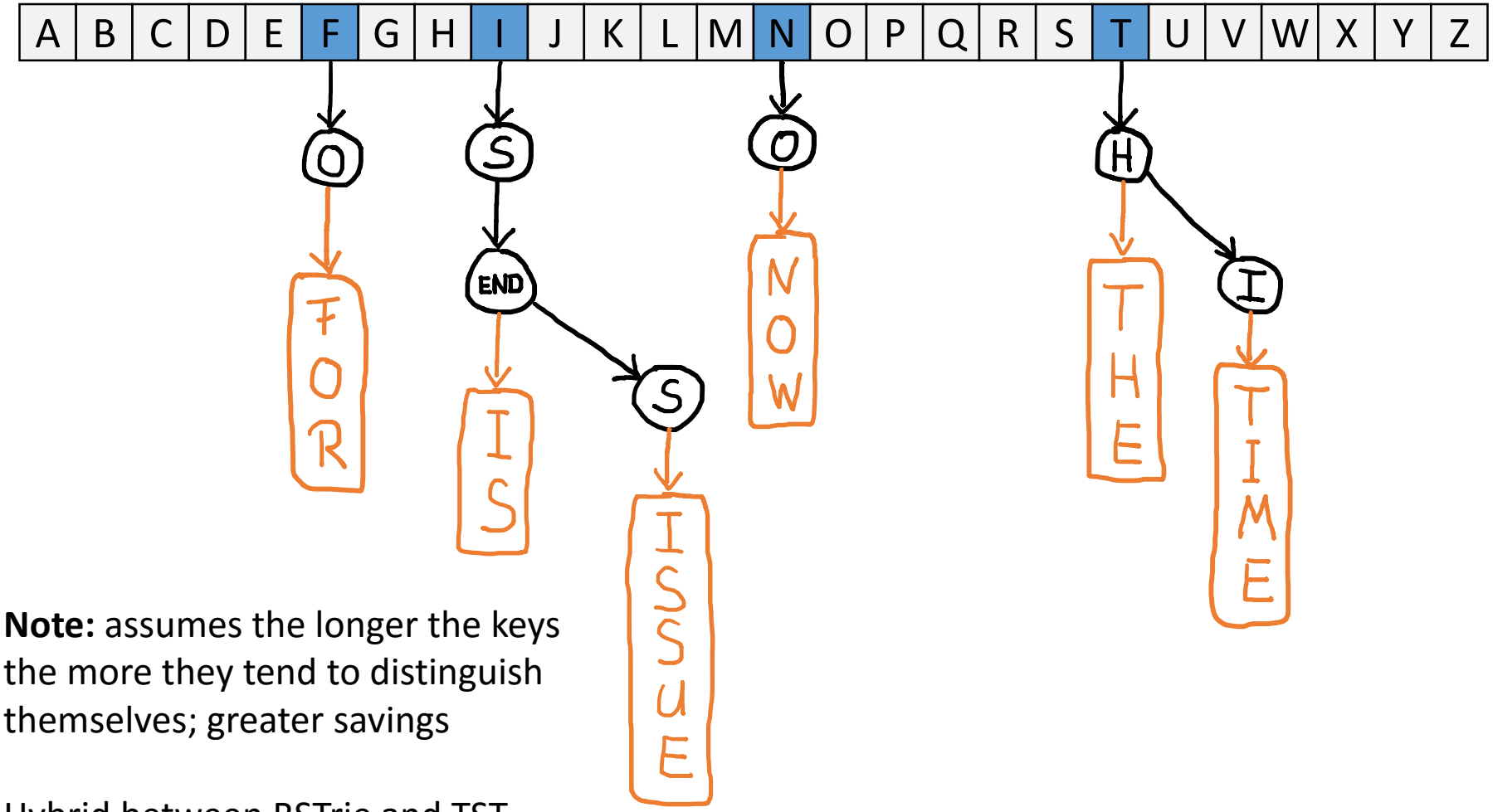
Optimising TSTs

Second idea: compact tails, store keys at leaves



Optimising TSTs

Second idea: compact tails, store keys at leaves

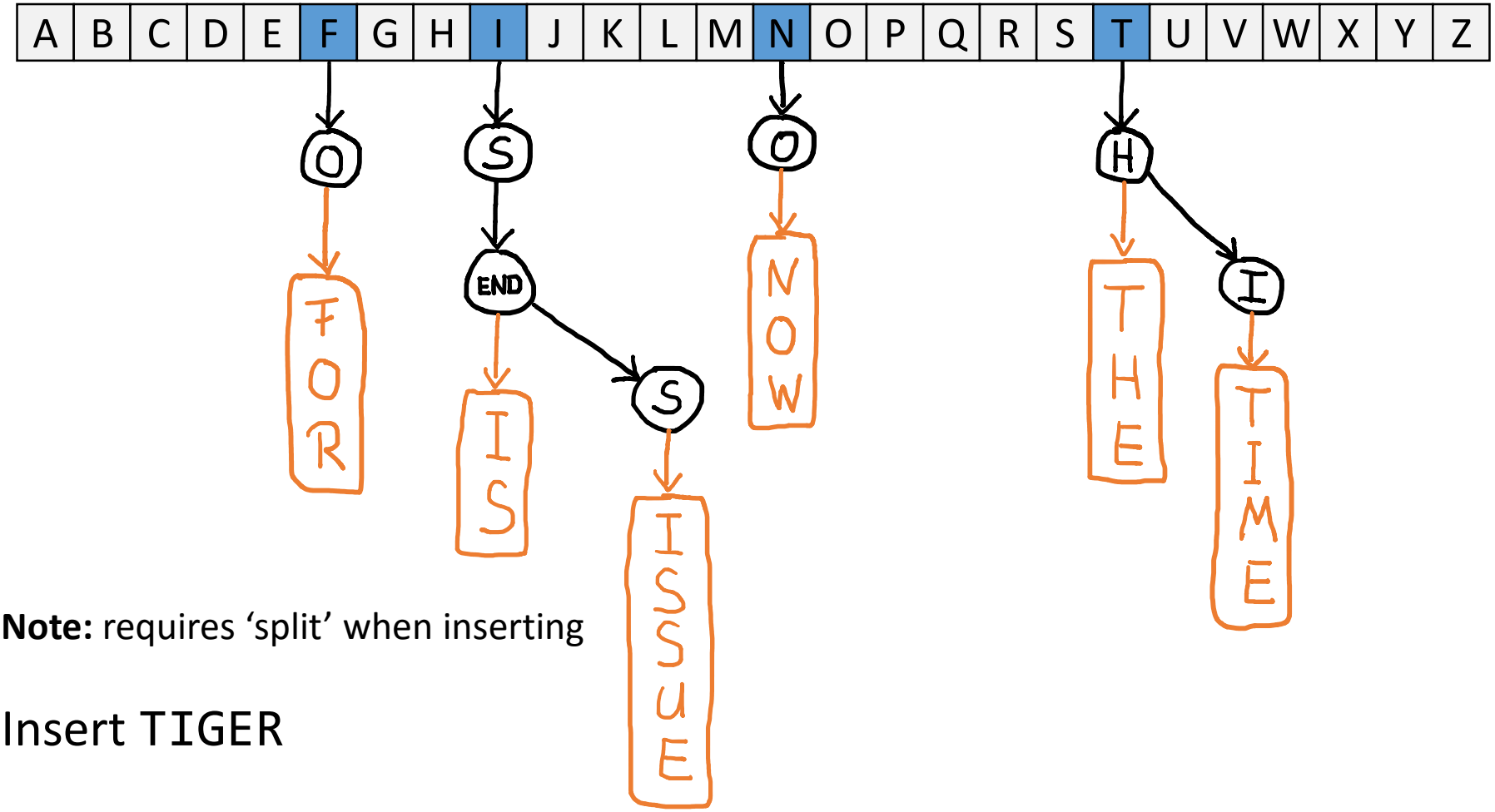


Note: assumes the longer the keys the more they tend to distinguish themselves; greater savings

Hybrid between BSTrie and TST

Optimising TSTs

Second idea: compact tails, store keys at leaves

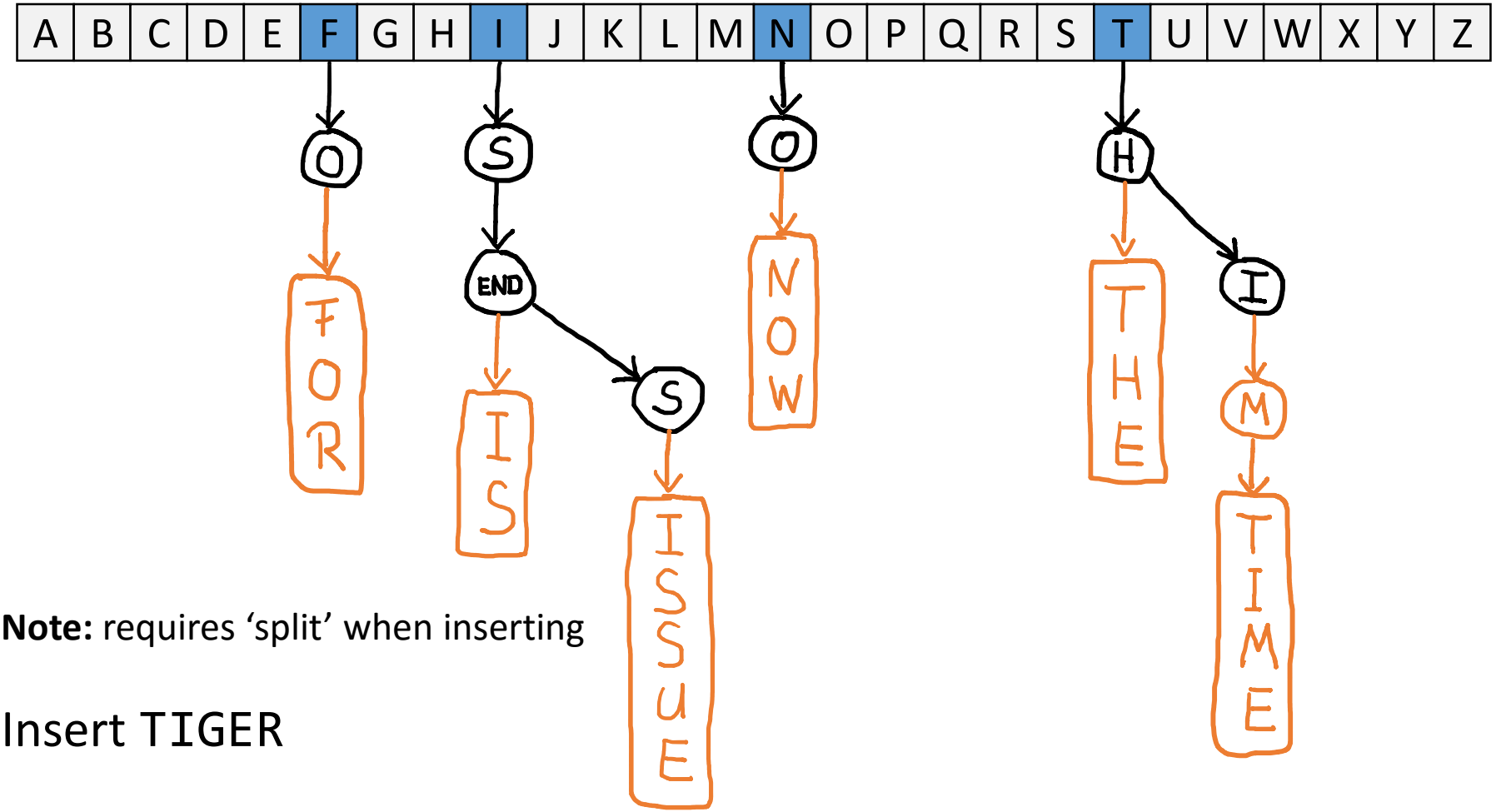


Note: requires 'split' when inserting

Insert TIGER

Optimising TSTs

Second idea: compact tails, store keys at leaves

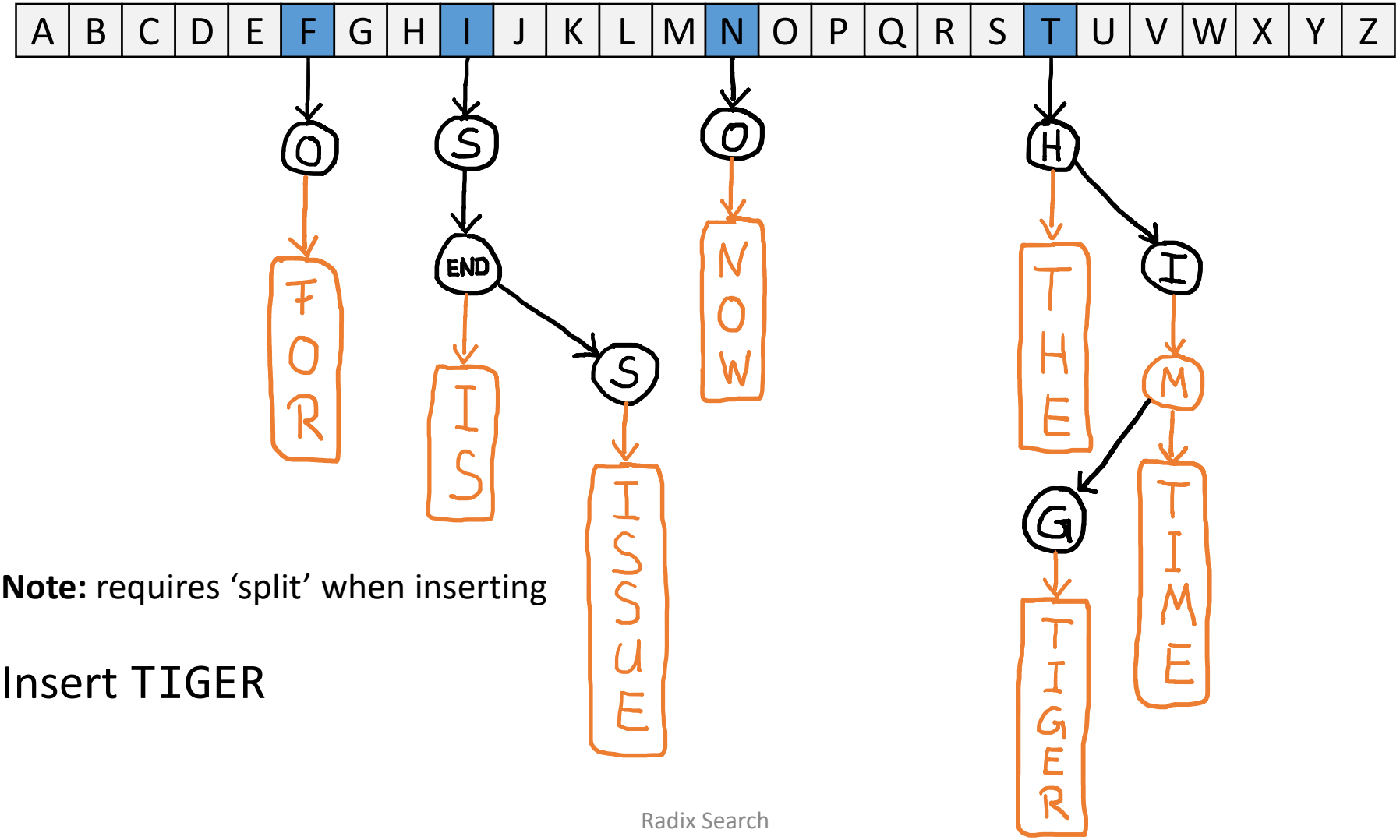


Note: requires 'split' when inserting

Insert TIGER

Optimising TSTs

Second idea: compact tails, store keys at leaves



Note: requires ‘split’ when inserting

Insert TIGER

Analysis of TSTs

- A search or insert in a TST with keys at the leaves and R^d-way branching at the root requires approximately $\ln n - d \ln R$ character accesses for n keys that are random character strings
- The number of links required is R^d (for the root node) plus a small constant multiple of n

Example:

consider 1 billion random keys of strings with $R = 256$, and we use a table of size $R^2 = 65536$ at the top

a typical search will require about $\ln 10^9 - 2 \ln 256 \approx 9.6$ character comparisons

Thought Exercise

How might a search engine use a TST?

- Consider “suggestion completion”

  given some characters, suggest some possible searches to perform

  two parts to the problem

  1. searching for completions
  2. ranking the completions for relevance

- The TST is used to implement a **suffix tree**, which is a form of string index

Suffix Trees

Each position in a text is treated as the beginning of a string key running to the end of the text

- typically, we start at word boundaries

  Ready are you? What know you of ready? For eight hundred years have I trained Jedi. My own counsel will I keep on who is to be trained. A Jedi must have the deepest commitment, the most serious mind. This one a long time have I watched.

- building up and entering the keys one by one

Key 1: *Ready are you? What know you of ready? For eight…*

Key 2: *are you? What know you of ready? For eight hundred…*

Key 3: *you? What know you of ready? For eight hundred years…*

Key 4: *What know you of ready? For eight hundred years have…*

Space and Time of Suffix Trees

- Space is not a significant issue

  the suffix tree is an index into a shared text

- TST-based algorithms are highly suitable because their running time does not depend on the key length, but rather on the number of digits (i.e., characters) needed to distinguish among the keys

Empirical Analysis

Relative timings for random sequences of n keys formed from long, semi-structured strings

	construction				search misses			
n	B	H	T	T*	B	H	T	T*
1250	4	4	5	5	2	2	2	1
2500	8	7	10	9	5	5	3	2
5000	19	16	21	20	10	8	6	4
12500	48	48	54	97	29	27	15	14
25000	118	99	188	156	67	59	36	30
50000	230	191	333	255	137	113	70	65

B Binary Search Tree
H Hashing with separate chaining
T TST
T* TST with R^2-way branch at root

[Sedgewick] p.671

Conclusions

- Radix search is a powerful technique for efficient search

  closely related to radix sort

- Many variants of radix search algorithms

  BSTries, Patricia Tries, Multiway Tries, Existence Tries, TSTs

- The optimal choice highly depends on the application

References

Books

- [Sedgewick] **Algorithms in Java, Parts 1-4**
  R. Sedgewick. Addison-Wesley. 2002 (3rd Edition)

Online

- http://algs4.cs.princeton.edu/lectures/
- https://www.coursera.org/courses?query=algorithms