Representing and Searching Sets of Strings

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

April 27, 2016

Outline

- Radix search
- Ternary search tries

Sets of Strings

- Several very important applications

Sets of Strings

- Several very important applications

  E.g.,

  - dictionary (of words)
  - symbol table in a compiler
  - all kinds of key-based index
  - ...

Symbol Table

- Operations

- Operations
  - insert(Key)
  - delete(Key)
  - search(Key)
  - min()
  - max()

Dictionary

- Operations

Dictionary

- Operations
  - insert(Key)
  - search(Key)

Dictionary

- Operations
  - insert(Key)
  - search(Key)
- No *delete* operation
- Built once and searched many times

Binary Search

```
BINARYSEARCH(A, K)
first = 1
last = length(A)
while first ≤ last
        x = ⌈(first + last)/2⌉
        if A[x] == K
                return TRUE
        elseif first == last
                return FALSE
        elseif A[x] > K
                last = x − 1
        else fisrt = x + 1
return FALSE
```

```
TREE-SEARCH(T, K)
x = T.root
while x ≠ NIL and K ≠ x.key
        if K < x.key
                x = x.left
        else x = x.right
if x ≠ NIL
        return TRUE
else return FALSE
```

- Complexity?

Binary Search

K ***is a string!***

```
BINARYSEARCH(A, K)
first = 1
last = length(A)
while first ≤ last
    x = ⌈(first + last)/2⌉
    if A[x] == K
        return TRUE
    elseif first == last
        return FALSE
    elseif A[x] > K
        last = x − 1
    else fisrt = x + 1
return FALSE
```

```
TREE-SEARCH(T, K)
x = T.root
while x ≠ NIL and K ≠ x.key
    if K < x.key
        x = x.left
    else x = x.right
if x ≠ NIL
    return TRUE
else return FALSE
```

- Complexity?

Binary Search

K ***is a string!***

```
BINARYSEARCH(A, K)
first = 1
last = length(A)
while first ≤ last
    x = ⌈(first + last)/2⌉
    if A[x] == K
        return TRUE
    elseif first == last
        return FALSE
    elseif A[x] > K
        last = x − 1
    else fisrt = x + 1
return FALSE
```

```
TREE-SEARCH(T, K)
x = T.root
while x ≠ NIL and K ≠ x.key
    if K < x.key
        x = x.left
    else x = x.right
if x ≠ NIL
    return TRUE
else return FALSE
```

- Complexity?
  - we must account for the ***complexity of string comparisons***

String Comparison

- Assuming a string is an array of bytes, the condition $A[x] == K$ (line 5 of **BINARYSEARCH**) becomes **STRINGEQUALS**$(A[x], K)$

String Comparison

- Assuming a string is an array of bytes, the condition $A[x] == K$ (line 5 of **BinarySearch**) becomes **StringEquals**$(A[x], K)$

```
StringEquals(S1, S2)
if length(S1) ≠ length(S2)
    return FALSE
for i = 1 to length(S1)
    if S1[i] ≠ S2[i]
        return FALSE
return TRUE
```

String Comparison

- Assuming a string is an array of bytes, the condition $A[x] == K$ (line 5 of **BinarySearch**) becomes **StringEquals**$(A[x], K)$

```
StringEquals(S1, S2)
if length(S1) ≠ length(S2)
        return FALSE
for i = 1 to length(S1)
        if S1[i] ≠ S2[i]
                return FALSE
return TRUE
```

- The complexity of **StringEquals**(S_1, S_2) is $O(m)$, where m is the max string size

String Comparison

- Assuming a string is an array of bytes, the condition $A[x] == K$ (line 5 of **BinarySearch**) becomes **StringEquals**$(A[x], K)$

```
StringEquals(S1, S2)
if length(S1) ≠ length(S2)
        return FALSE
for i = 1 to length(S1)
        if S1[i] ≠ S2[i]
                return FALSE
return TRUE
```

- The complexity of **StringEquals**(S_1, S_2) is $O(m)$, where m is the max string size

- So, the complexity of **BinarySearch**(A, K) is $O(m \log n)$

What About a Hash Table

```
CHAINED-HASH-SEARCH(T, K)
L = T[h(K)]
return LIST-SEARCH(L, K)
```

```
HASH-SEARCH(T, K)
for i = 1 to length(T)
        j = h(K, i)
        if T[j] == K
                return TRUE
        if T[j] == NIL
                return FALSE
return FALSE
```

What About a Hash Table

```
CHAINED-HASH-SEARCH(T, K)
    L = T[h(K)]
    return LIST-SEARCH(L, K)
```

```
HASH-SEARCH(T, K)
    for i = 1 to length(T)
        j = h(K, i)
        if T[j] == K
            return TRUE
        if T[j] == NIL
            return FALSE
    return FALSE
```

- Complexity?

What About a Hash Table

```
CHAINED-HASH-SEARCH(T, K)
L = T[h(K)]
return LIST-SEARCH(L, K)
```

```
HASH-SEARCH(T, K)
for i = 1 to length(T)
        j = h(K, i)
        if T[j] == K
                return TRUE
        if T[j] == NIL
                return FALSE
return FALSE
```

- Complexity?
  - here, too, we must account for the string comparisons

What About a Hash Table

```
CHAINED-HASH-SEARCH(T, K)
L = T[h(K)]
return LIST-SEARCH(L, K)
```

```
HASH-SEARCH(T, K)
for i = 1 to length(T)
        j = h(K, i)
        if T[j] == K
                return TRUE
        if T[j] == NIL
                return FALSE
return FALSE
```

- Complexity?
  - here, too, we must account for the string comparisons
  - and for the hash functions

Observation

Observation

- When we start **BinarySearch**(A, K)
  - $A[x]$ is probably far away from K
  - so, **StringEquals**$(A[x], K)$ is likely to return quickly

Observation

- When we start **BinarySearch**(A, K)
  - $A[x]$ is probably far away from K
  - so, **StringEquals**$(A[x], K)$ is likely to return quickly
- Later in **BinarySearch**(A, K)
  - $A[x]$ gets closer and closer to K
  - so, **StringEquals**$(A[x], K)$ is likely to iterate for nearly m steps

Observation

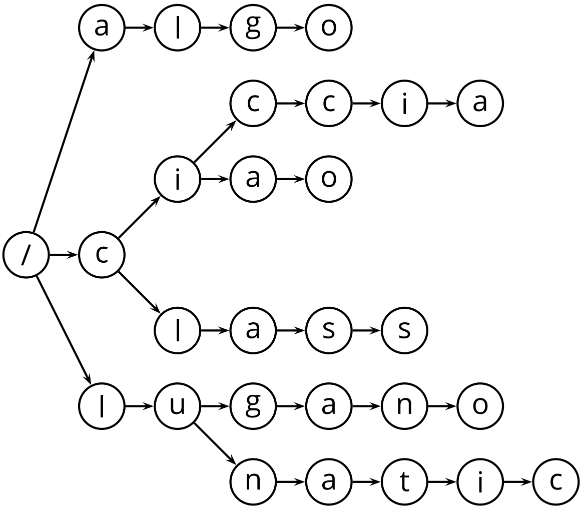
- When we start **BinarySearch**(A, K)
  - $A[x]$ is probably far away from K
  - so, **StringEquals**$(A[x], K)$ is likely to return quickly
- Later in **BinarySearch**(A, K)
  - $A[x]$ gets closer and closer to K
  - so, **StringEquals**$(A[x], K)$ is likely to iterate for nearly m steps
  - problem is, **StringEquals**$(A[x], K)$ is likely to go through the same prefix of K many times
- So, since $m = \Theta(\log N)$, and **BinarySearch**(A, K) uses $\Theta(\log N)$ comparisons each one running in $O(m)$:

$$T(N, m) = O(\log^2 N)$$

- Idea: a data structure where ***common prefixes*** are shared

A New Data Structure

- Idea: a data structure where ***common prefixes*** are shared



A "Trie"

- Data structure useful for information re*trie*val (pronounced "try" to distinguish it from a tree. . .)

A “Trie”

- Data structure useful for information re*trie*val (pronounced “try” to distinguish it from a tree. . .)
- Every node holds one character

A “Trie”

- Data structure useful for information re*trie*val (pronounced “try” to distinguish it from a tree. . .)

- Every node holds one character

- Keys with the same prefix share a branch of the tree

A "Trie"

- Data structure useful for information re*trie*val (pronounced "try" to distinguish it from a tree. . .)
- Every node holds one character
- Keys with the same prefix share a branch of the tree
- Keys are stored at (or just represented by) leaf nodes

A "Trie"

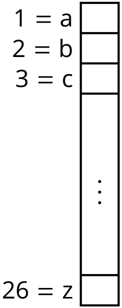
- Data structure useful for information re*trie*val (pronounced "try" to distinguish it from a tree. . .)
- Every node holds one character
- Keys with the same prefix share a branch of the tree
- Keys are stored at (or just represented by) leaf nodes
- *Question:* how do we represent nodes and links?

A “Trie”

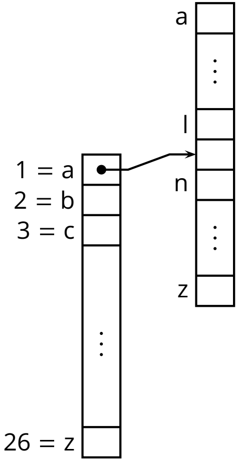
- Data structure useful for information re*trie*val (pronounced “try” to distinguish it from a tree. . .)
- Every node holds one character
- Keys with the same prefix share a branch of the tree
- Keys are stored at (or just represented by) leaf nodes
- *Question:* how do we represent nodes and links?
  - one way would be to hold $|\Sigma|$ links
  - one for each character of the given alphabet Σ

Radix Trie

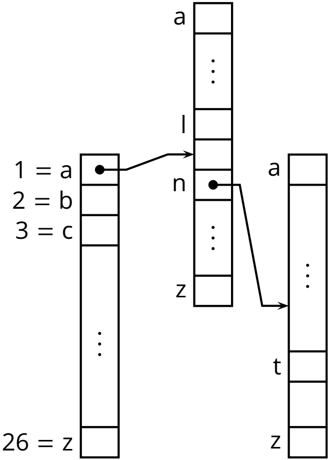
Radix Trie



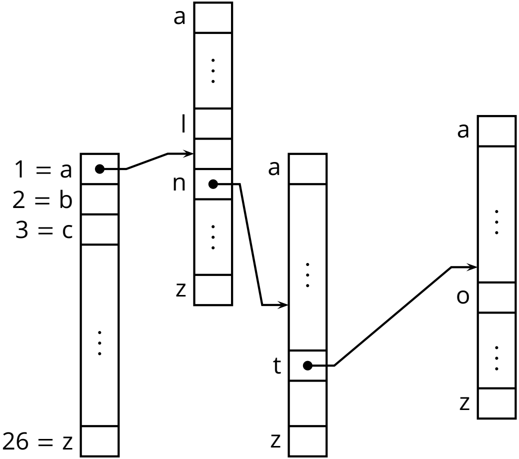
Radix Trie



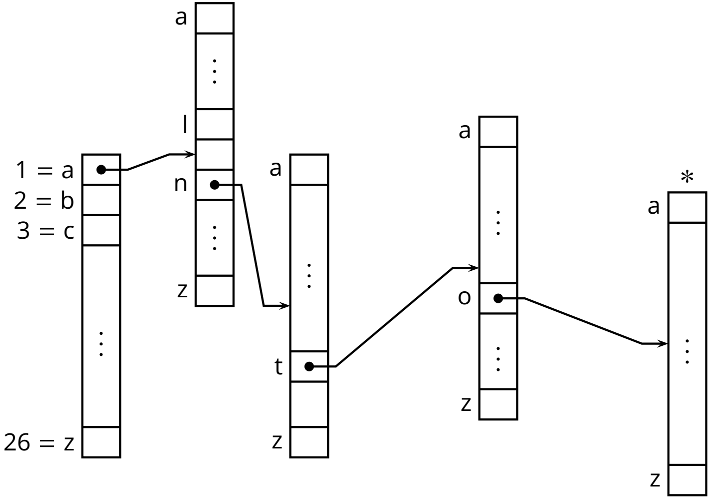
Radix Trie



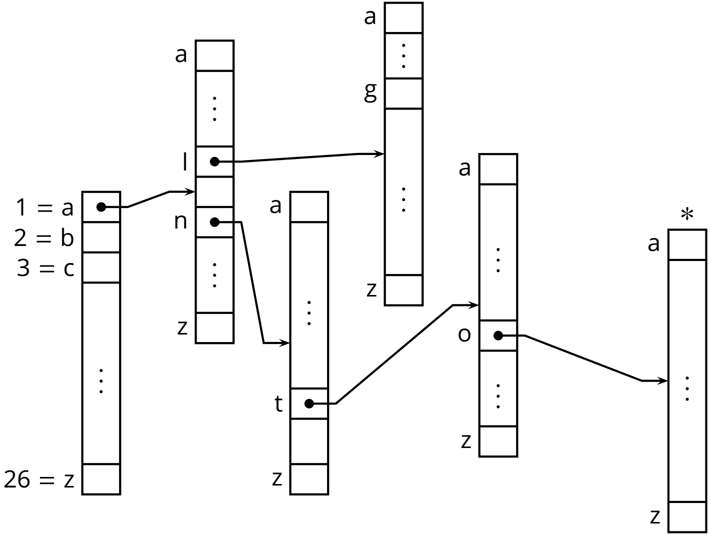
Radix Trie



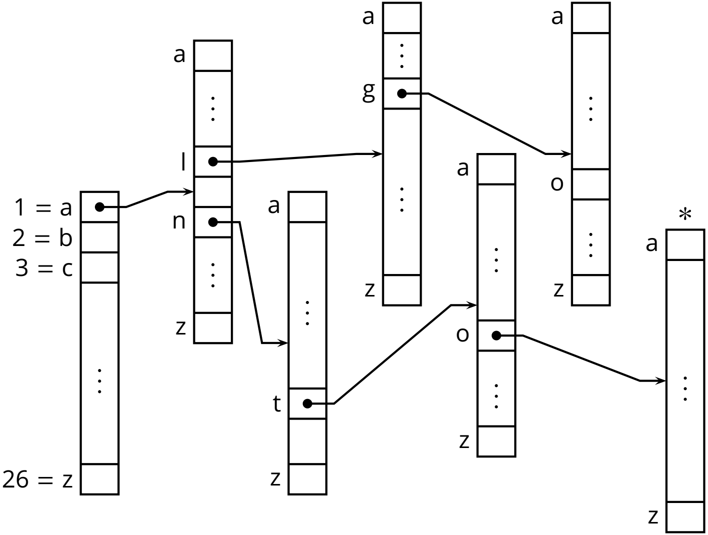
Radix Trie



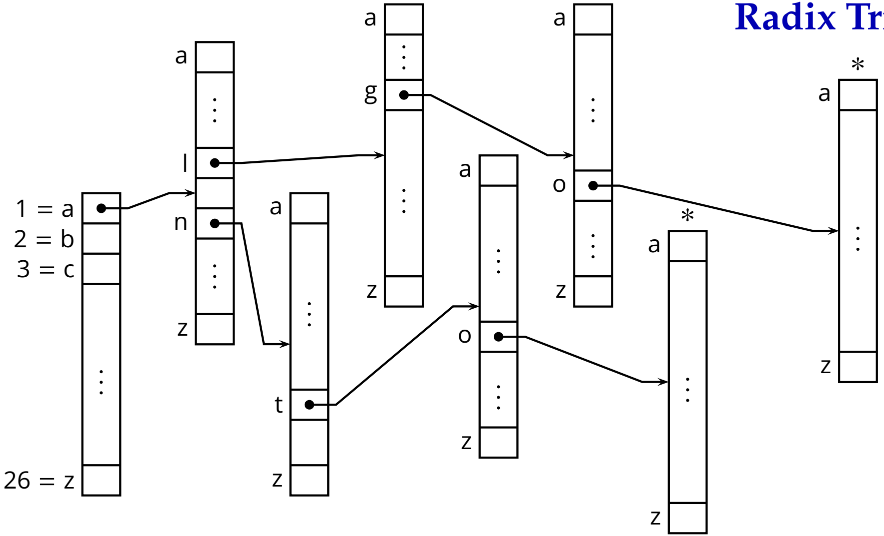
Radix Trie



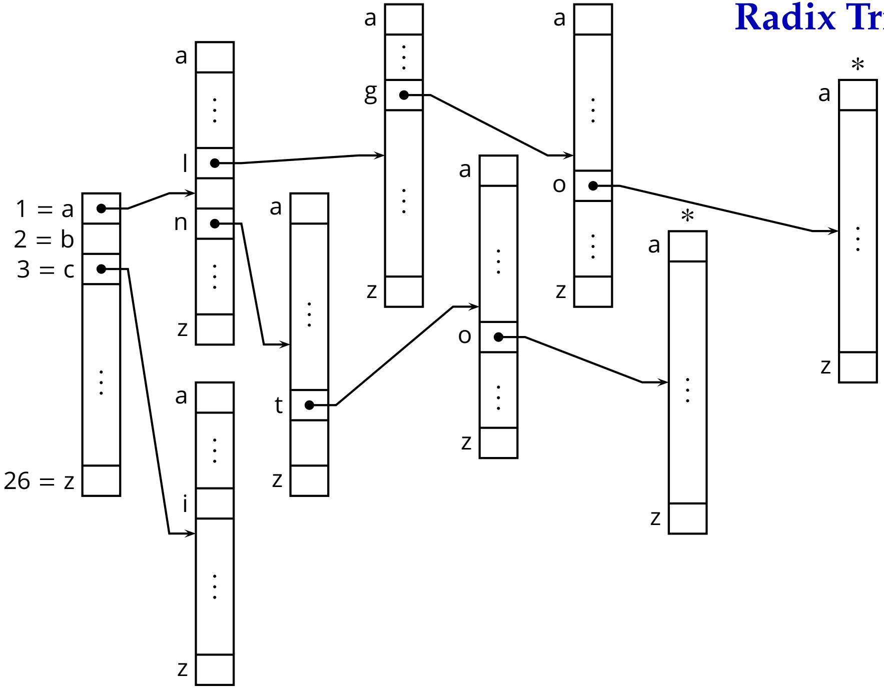
Radix Trie



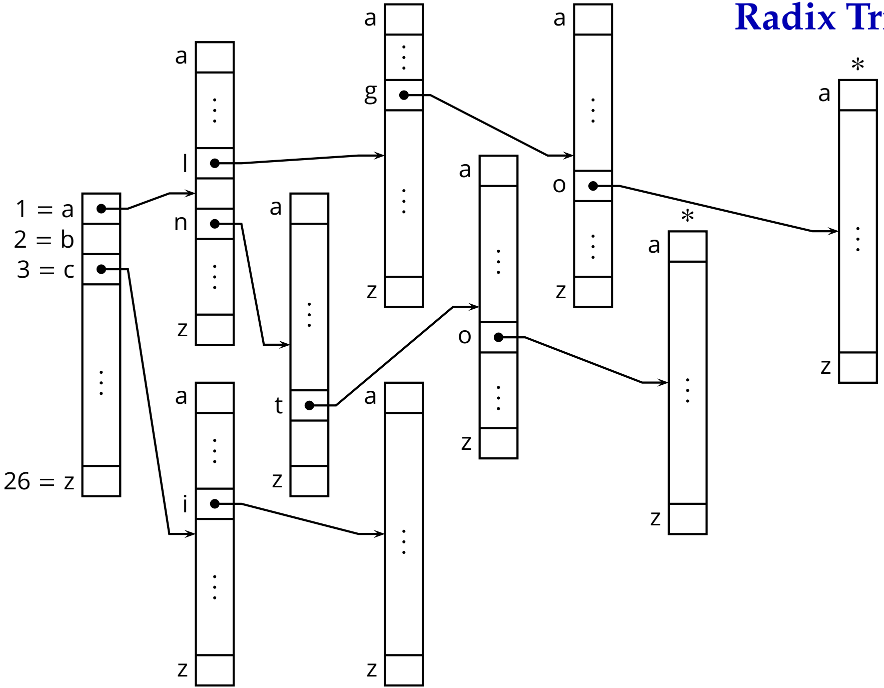
Radix Trie



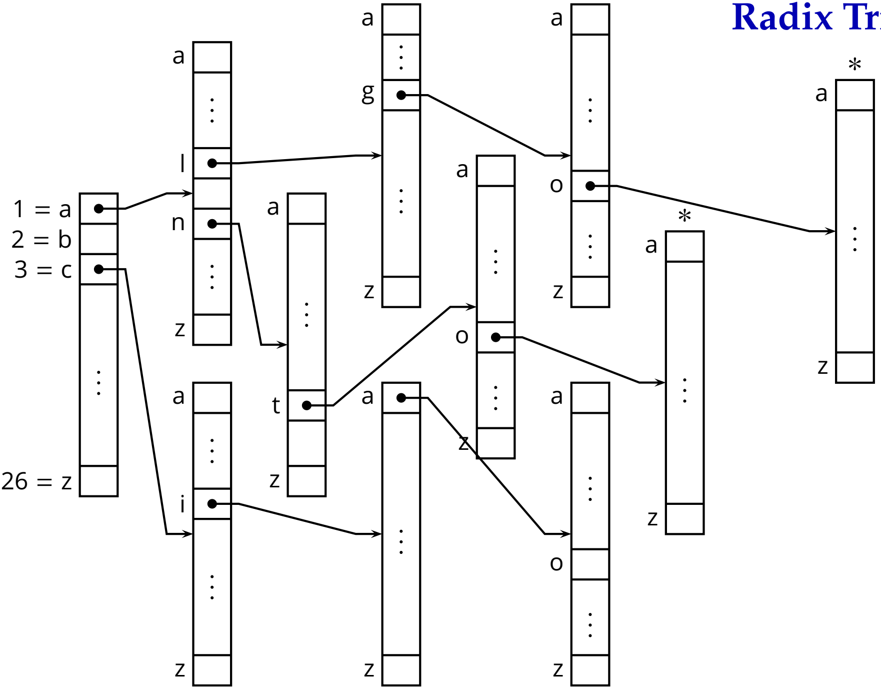
Radix Trie



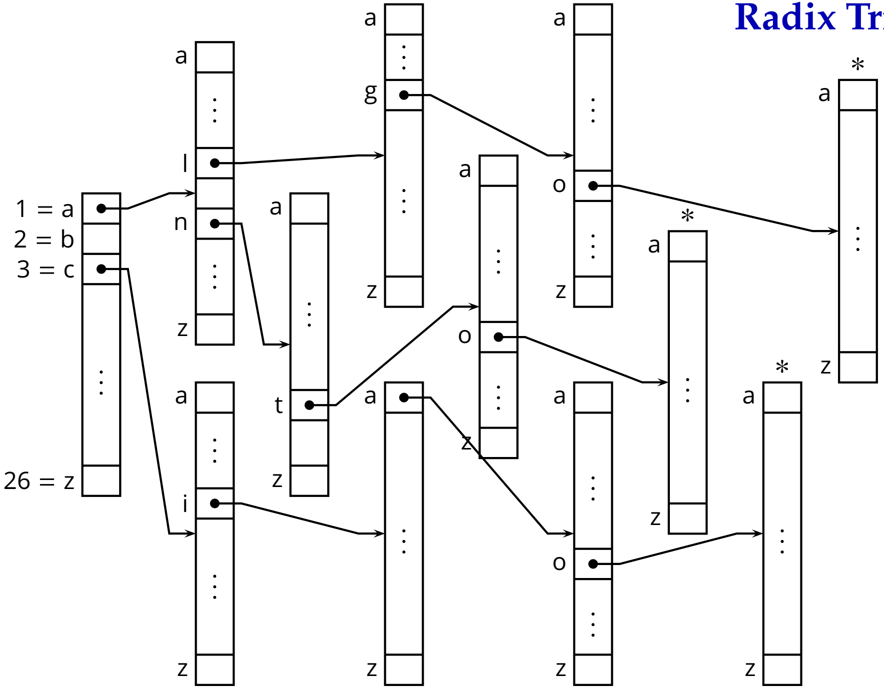
Radix Trie



Radix Trie



Radix Trie



- Every element x has an array of links $x.links$
  - e.g., in “radix-256,” an element represents a *byte* in a string (of bytes)
- Every element x has a $x.value$ that is TRUE if that prefix corresponds to a string in the dictionary
  - this is to distinguish an entire word from a prefix

- Every element x has an array of links $x.links$
  - e.g., in "radix-256," an element represents a *byte* in a string (of bytes)

- Every element x has a $x.value$ that is TRUE if that prefix corresponds to a string in the dictionary
  - this is to distinguish an entire word from a prefix

```
RADIXSEARCH(Root, K)
n = Root
for i = 1 to length(K)
        if n.links[K[i]] == NIL
                return FALSE
        else n = n.links[K[i]]
return n.value
```

Complexities of Radix Search

- What is the complexity of Radix Search with a dictionary of N strings of up to m characters?

Complexities of Radix Search

- What is the complexity of Radix Search with a dictionary of *N* strings of up to *m* characters?

$$\boxed{T(N, m) = \Theta(m)}$$

Complexities of Radix Search

- What is the complexity of Radix Search with a dictionary of N strings of up to m characters?

$$T(N, m) = \Theta(m)$$

- What is the ***space complexity?***

Complexities of Radix Search

- What is the complexity of Radix Search with a dictionary of N strings of up to m characters?

$$T(N, m) = \Theta(m)$$

- What is the ***space complexity?***
  - first approximation:

$$S(N, m) = O(|\Sigma| mN)$$

Complexities of Radix Search

- What is the complexity of Radix Search with a dictionary of N strings of up to m characters?

$$\boxed{T(N, m) = \Theta(m)}$$

- What is the ***space complexity?***
  - first approximation:

$$S(N, m) = O(|\Sigma| mN) \qquad S(N, m) = \Omega(|\Sigma| \log_{|\Sigma|} N)$$

Complexities of Radix Search

- What is the complexity of Radix Search with a dictionary of N strings of up to m characters?

$$\boxed{T(N,m) = \Theta(m)}$$

- What is the ***space complexity?***
  - first approximation:

$$S(N,m) = O(|\Sigma| mN) \qquad S(N,m) = \Omega(|\Sigma| \log_{|\Sigma|} N)$$

  - a better characterization (*Exercise:* figure this out!):

$$S(N,m) = \Theta\left(|\Sigma|\left[\frac{N-1}{|\Sigma|-1} + N\left(m - \frac{\log N}{\log|\Sigma|}\right)\right]\right)$$

Idea

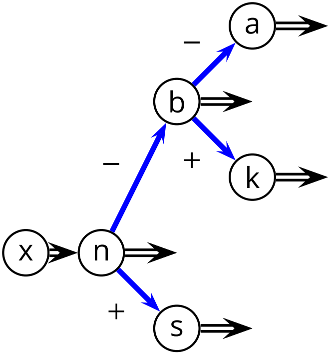
- We do not represent a full array of links

Idea

- We do not represent a full array of links
- Instead, we represent a small binary “index” of the existing links

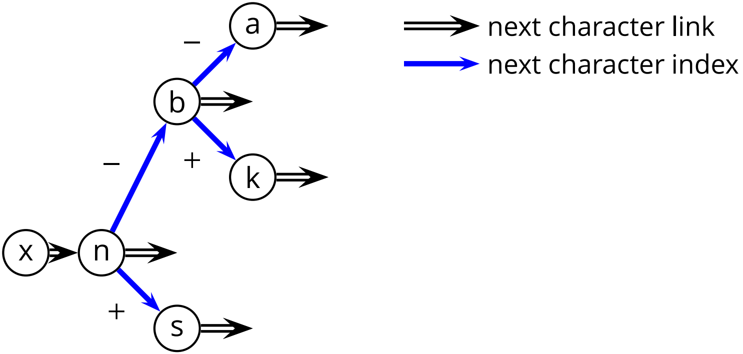
- We do not represent a full array of links
- Instead, we represent a small binary "index" of the existing links

  E.g., prefixes "xa", "xb", "xn", "xk", and "xs" might be represented as follows



- We do not represent a full array of links
- Instead, we represent a small binary “index” of the existing links

  E.g., prefixes “xa”, “xb”, “xn”, “xk”, and “xs” might be represented as follows



Ternary Search Trie

- *n.character* is the character at node *n*; i.e., the last character in the *prefix* represented by *n*

Ternary Search Trie

- *n.character* is the character at node *n*; i.e., the last character in the *prefix* represented by *n*

- *n.value* is the *value* to which *n* maps to; if the TST is a dictionary, then *n.value* is *true* iff the prefix represented by *n* is a key in the dictionary

- *n.character* is the character at node *n*; i.e., the last character in the *prefix* represented by *n*

- *n.value* is the *value* to which *n* maps to; if the TST is a dictionary, then *n.value* is *true* iff the prefix represented by *n* is a key in the dictionary

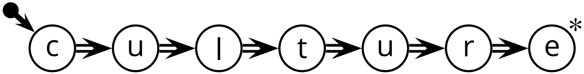
- A node *n* has three links
  - *n.lower* links to a node representing a “lower” character at the same position
  - *n.higher* links to a node representing a “higher” character at the same position
  - *n.equal* links to a node representing a character in the next position

Example

“culture”

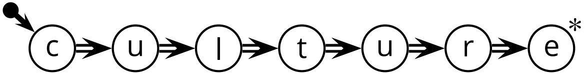
Example

"culture"



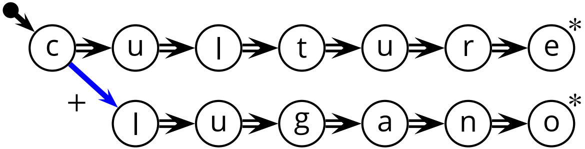
Example

“lugano”



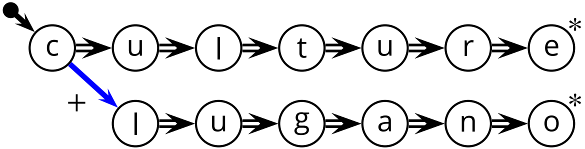
Example

"lugano"



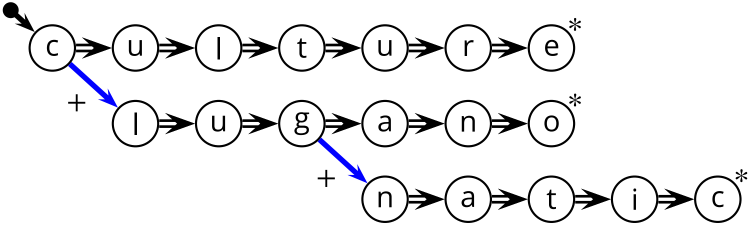
Example

"lunatic"



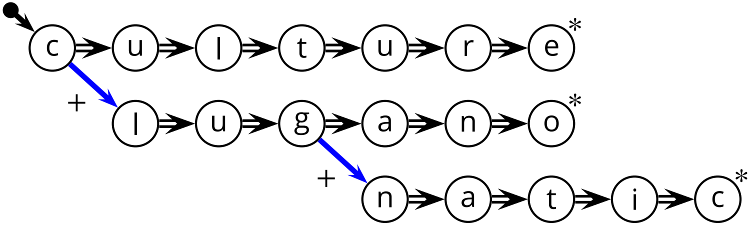
Example

"lunatic"



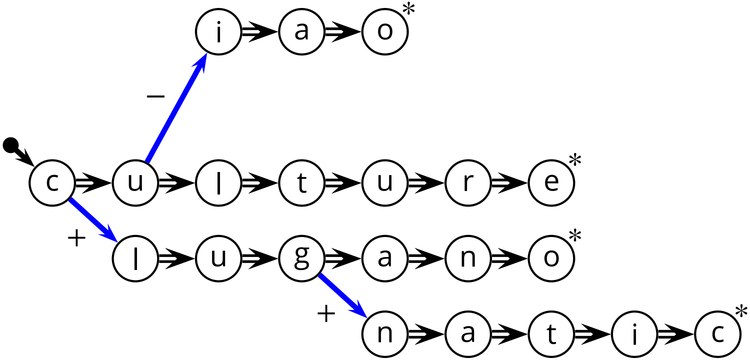
Example

"ciao"



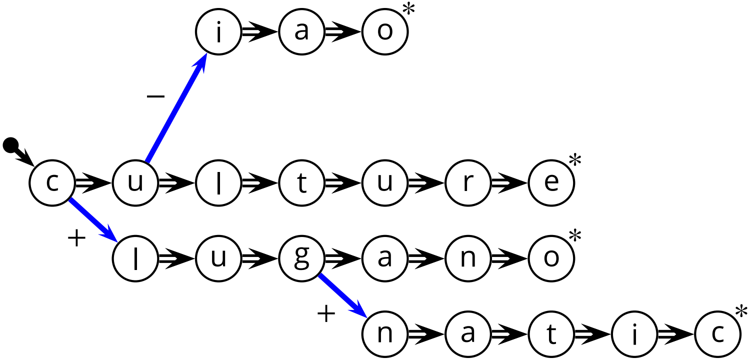
Example

“ciao”



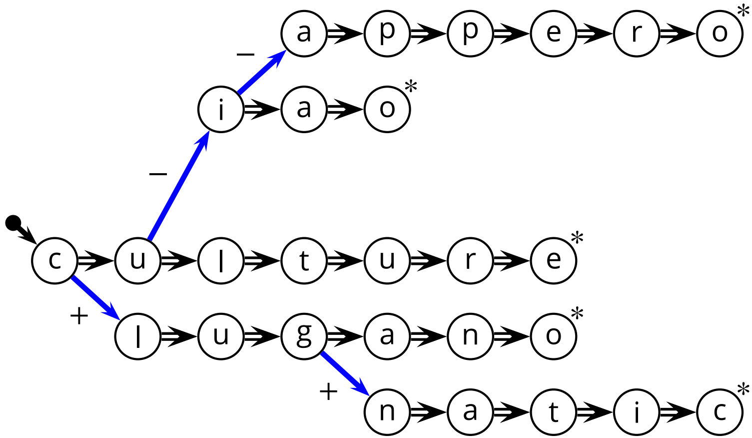
Example

“cappero”



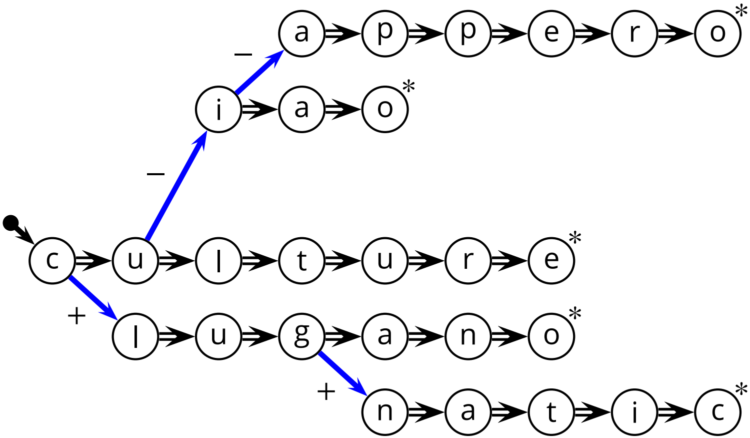
Example

"cappero"



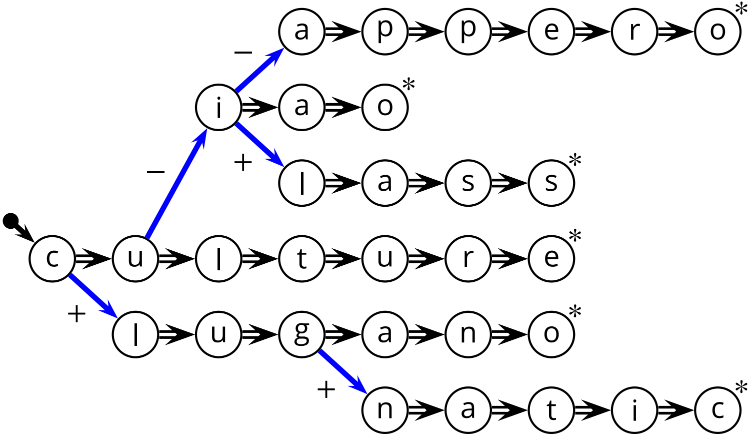
Example

“class”



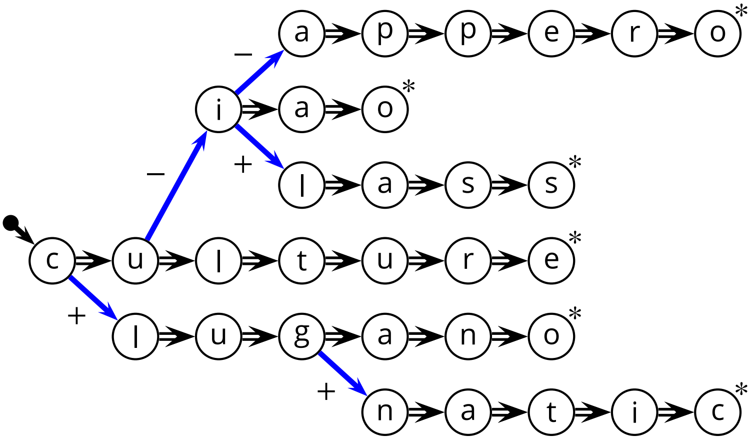
Example

"class"



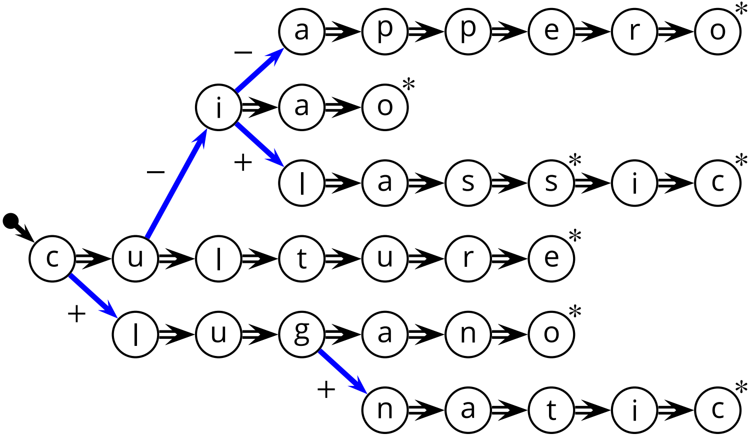
Example

"classic"



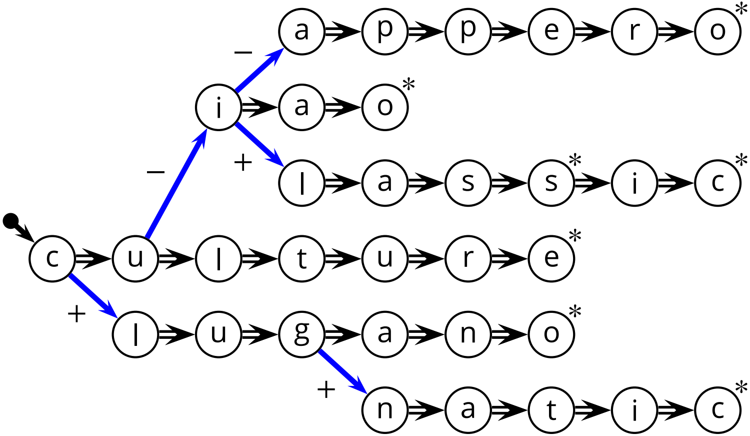
Example

“classic”



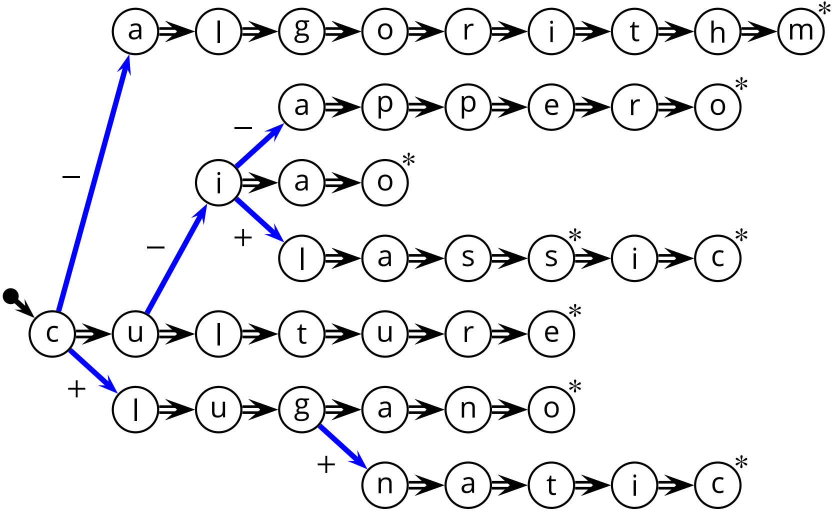
Example

"algorithm"



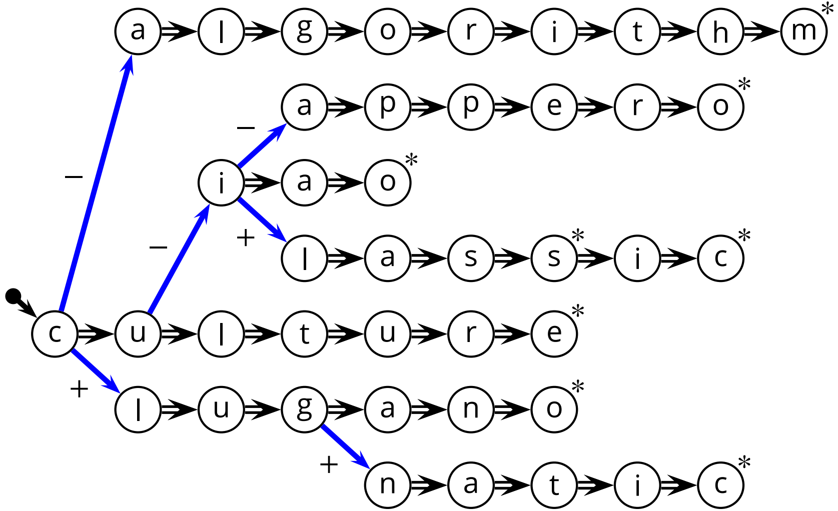
Example

“algorithm”



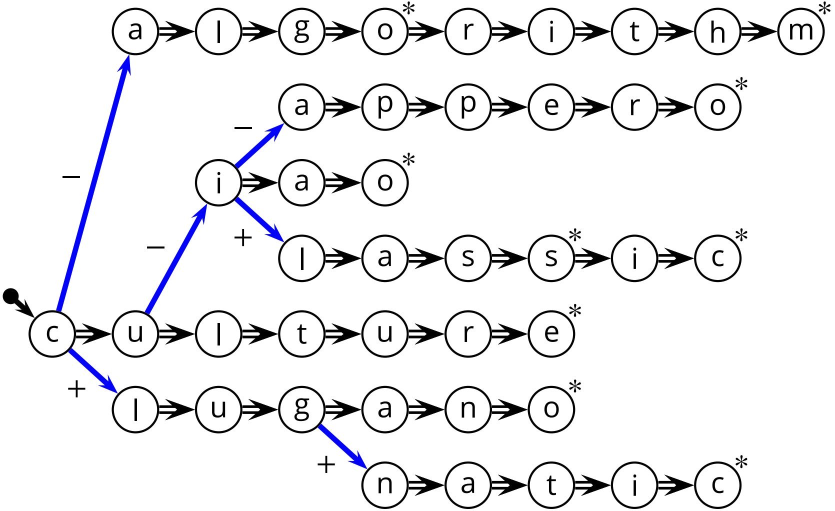
Example

"algo"



Example

"algo"



TST Search

TST Search

```
TSTSearch(T, K)
for i = 1 to |K|
    if i > 1
        T = T.equal
    while T ≠ NIL and K[i] ≠ T.character
        if K[i] < T.character
            T = T.lower
        else T = T.higher
    if T == NIL
        return FALSE
return n.value
```

TST Search

```
TSTSearch(T, K)
for i = 1 to |K|
    if i > 1
        T = T.equal
    while T ≠ NIL and K[i] ≠ T.character
        if K[i] < T.character
            T = T.lower
        else T = T.higher
    if T == NIL
        return FALSE
return n.value
```

- Is it correct?

TST Search

```
TSTSearch(T, K)
for i = 1 to |K|
    if i > 1
        T = T.equal
    while T ≠ NIL and K[i] ≠ T.character
        if K[i] < T.character
            T = T.lower
        else T = T.higher
    if T == NIL
        return FALSE
return n.value
```

- Is it correct? Not completely! (**Exercise:** fix it.)
- Complexity?

TST Search

```
TSTSearch(T, K)
for i = 1 to |K|
    if i > 1
        T = T.equal
    while T ≠ NIL and K[i] ≠ T.character
        if K[i] < T.character
            T = T.lower
        else T = T.higher
    if T == NIL
        return FALSE
return n.value
```

- Is it correct? Not completely! (**Exercise:** fix it.)
- Complexity? Non-trivial…

TST Insertion

TST Insertion

- Recursion starts with $root = \textbf{TSTInsert}(root, K, 1)$

```
TSTInsert(T, K, i)
if T == NIL
    T = NewNode(K[i])
if K[i] < T.character
    T.lower = TSTInsert(T.lower, K, i)
elseif K[i] > T.character
    T.higher = TSTInsert(T.higher, K, i)
elseif K[i] == T.character
    if i < |K|
        T.equal = TSTInsert(T.equal, K, i + 1)
    else T.value = TRUE
return T
```